ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Distributed Computing

Principles of

Distributed Computing

Roger Wattenhofer

wattenhofer@ethz.ch

Spring 2016

Contents

Introduction

What is Distributed Computing?

In the last few decades, we have experienced an unprecedented growth in the area of distributed systems and networks. Distributed computing now encompasses many of the activities occurring in today's computer and communications world. Indeed, distributed computing appears in quite diverse application areas: The Internet, wireless communication, cloud or parallel computing, multi-core systems, mobile networks, but also an ant colony, a brain, or even the human society can be modeled as distributed systems.

These applications have in common that many processors or entities (often called nodes) are active in the system at any moment. The nodes have certain degrees of freedom: they have their own hard- and software. Nevertheless, the nodes may share common resources and information, and, in order to solve a problem that concerns several—or maybe even all—nodes, coordination is necessary.

Despite these commonalities, a human brain is of course very different from a quadcore processor. Due to such differences, many different models and parameters are studied in the area of distributed computing. In some systems the nodes operate synchronously, in other systems they operate asynchronously. There are simple homogeneous systems, and heterogeneous systems where different types of nodes, potentially with different capabilities, objectives etc., need to interact. There are different communication techniques: nodes may communicate by exchanging messages, or by means of shared memory. Occasionally the communication infrastructure is tailor-made for an application, sometimes one has to work with any given infrastructure. The nodes in a system often work together to solve a global task, occasionally the nodes are autonomous agents that have their own agenda and compete for common resources. Sometimes the nodes can be assumed to work correctly, at times they may exhibit failures. In contrast to a single-node system, distributed systems may still function correctly despite failures as other nodes can take over the work of the failed nodes. There are different kinds of failures that can be considered: nodes may just crash, or they might exhibit an arbitrary, erroneous behavior, maybe even to a degree where it cannot be distinguished from malicious (also known as Byzantine) behavior. It is also possible that the nodes follow the rules indeed, however they tweak the parameters to get the most out of the system; in other words, the nodes act selfishly.

Apparently, there are many models (and even more combinations of models) that can be studied. We will not discuss them in detail now, but simply define

them when we use them. Towards the end of the course a general picture should emerge, hopefully!

Course Overview

This course introduces the basic principles of distributed computing, highlighting common themes and techniques. In particular, we study some of the fundamental issues underlying the design of distributed systems:

- Communication: Communication does not come for free; often communication cost dominates the cost of local processing or storage. Sometimes we even assume that everything but communication is free.
- Coordination: How can you coordinate a distributed system so that it performs some task efficiently? How much overhead is inevitable?
- Fault-tolerance: A major advantage of a distributed system is that even in the presence of failures the system as a whole may survive.
- Locality: Networks keep growing. Luckily, global information is not always needed to solve a task, often it is sufficient if nodes talk to their neighbors. In this course, we will address whether a local solution is possible.
- Parallelism: How fast can you solve a task if you increase your computational power, e.g., by increasing the number of nodes that can share the workload? How much parallelism is possible for a given problem?
- Symmetry breaking: Sometimes some nodes need to be selected to orchestrate computation or communication. This is achieved by a technique called symmetry breaking.
- Synchronization: How can you implement a synchronous algorithm in an asynchronous environment?
- Uncertainty: If we need to agree on a single term that fittingly describes this course, it is probably "uncertainty". As the whole system is distributed, the nodes cannot know what other nodes are doing at this exact moment, and the nodes are required to solve the tasks at hand despite the lack of global knowledge.

Finally, there are also a few areas that we will not cover in this course, mostly because these topics have become so important that they deserve their own courses. Examples for such topics are distributed programming or security/cryptography.

In summary, in this class we explore essential algorithmic ideas and lower bound techniques, basically the "pearls" of distributed computing and network algorithms. We will cover a fresh topic every week.

Have fun!

Chapter Notes

Many excellent text books have been written on the subject. The book closest to this course is by David Peleg [Pel00], as it shares about half of the material. A main focus of Peleg's book are network partitions, covers, decompositions, and spanners – an interesting area that we will only touch in this course. There exist a multitude of other text books that overlap with one or two chapters of this course, e.g., [Lei92, Bar96, Lyn96, Tel01, AW04, HKP+05, CLRS09, Suo12]. Another related course is by James Aspnes [Asp] and one by Jukka Suomela [Suo14].

Some chapters of this course have been developed in collaboration with (former) Ph.D. students, see chapter notes for details. Many students have helped to improve exercises and script. Thanks go to Philipp Brandes, Raphael Eidenbenz, Roland Flury, Klaus-Tycho Förster, Stephan Holzer, Barbara Keller, Fabian Kuhn, Christoph Lenzen, Thomas Locher, Remo Meier, Thomas Moscibroda, Regina O'Dell, Yvonne-Anne Pignolet, Jochen Seidel, Stefan Schmid, Johannes Schneider, Jara Uitto, Pascal von Rickenbach (in alphabetical order).

Bibliography

[Asp] James Aspnes. Notes on Theory of Distributed Systems.

[AW04] Hagit Attiya and Jennifer Welch. *Distributed Computing: Fundamentals, Simulations and Advanced Topics (2nd edition)*. John Wiley Interscience, March 2004.

[Bar96] Valmir C. Barbosa. *An introduction to distributed algorithms*. MIT Press, Cambridge, MA, USA, 1996.

[CLRS09] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms (3. ed.)*. MIT Press, 2009.

[HKP+05] Juraj Hromkovic, Ralf Klasing, Andrzej Pelc, Peter Ruzicka, and Walter Unger. *Dissemination of Information in Communication Networks - Broadcasting, Gossiping, Leader Election, and Fault-Tolerance*. Texts in Theoretical Computer Science. An EATCS Series. Springer, 2005.

[Lei92] F. Thomson Leighton. *Introduction to parallel algorithms and architectures: array, trees, hypercubes*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1992.

[Lyn96] Nancy A. Lynch. *Distributed Algorithms*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1996.

[Pel00] David Peleg. *Distributed Computing: a Locality-Sensitive Approach*. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 2000.

[Suo12] Jukka Suomela. Deterministic Distributed Algorithms, 2012.

[Suo14] Jukka Suomela. Distributed algorithms. Online textbook, 2014.

[Tel01] Gerard Tel. *Introduction to Distributed Algorithms*. Cambridge University Press, New York, NY, USA, 2nd edition, 2001.

Chapter 1

Vertex Coloring

Vertex coloring is an infamous graph theory problem. It is also a useful toy example to see the style of this course already in the first lecture. Vertex coloring does have quite a few practical applications, for example in the area of wireless networks where coloring is the foundation of so-called TDMA MAC protocols. Generally speaking, vertex coloring is used as a means to break symmetries, one of the main themes in distributed computing. In this chapter we will not really talk about vertex coloring applications, but treat the problem abstractly. At the end of the class you probably learned the fastest algorithm ever! Let us start with some simple definitions and observations.

1.1 Problem & Model

Problem 1.1 (Vertex Coloring)**.** *Given an undirected graph $G = (V, E)$, assign a color c_v to each vertex $v \in V$ such that the following holds: $e = (v, w) \in E \Rightarrow c_v \neq c_w$.*

Remarks:

- Throughout this course, we use the terms *vertex* and *node* interchangeably.
- The application often asks us to use few colors! In a TDMA MAC protocol, for example, less colors immediately imply higher throughput. However, in distributed computing we are often happy with a solution which is suboptimal. There is a tradeoff between the optimality of a solution (efficacy), and the work/time needed to compute the solution (efficiency).

Assumption 1.3 (Node Identifiers)**.** *Each node has a unique identifier, e.g., its IP address. We usually assume that each identifier consists of only* $\log n$ *bits if the system has n nodes.*

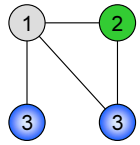


Figure 1.2: 3-colorable graph with a valid coloring.

Remarks:

- Sometimes we might even assume that the nodes exactly have identifiers $1, \ldots, n$.
- It is easy to see that node identifiers (as defined in Assumption 1.3) solve the coloring problem 1.1, but using n colors is not exciting. How many colors are needed is a well-studied problem:

Definition 1.4 (Chromatic Number)**.** *Given an undirected Graph $G = (V, E)$, the* chromatic number $\chi(G)$ *is the minimum number of colors to solve Problem 1.1.*

To get a better understanding of the vertex coloring problem, let us first look at a simple non-distributed ("centralized") vertex coloring algorithm:

Algorithm 1.5 Greedy Sequential

1: **while** there is an uncolored vertex v **do**
2: color v with the minimal color (number) that does not conflict with the already colored neighbors
3: **end while**

Definition 1.6 (Degree)**.** *The number of neighbors of a vertex v, denoted by $\delta(v)$, is called the* degree *of v. The maximum degree vertex in a graph G defines the graph degree $\Delta(G) = \Delta$.*

Theorem 1.7. *Algorithm 1.5 is correct and terminates in n "steps". The algorithm uses at most $\Delta + 1$ colors.*

Proof: Since each node has at most Δ neighbors, there is always at least one color free in the range $\{1, \ldots, \Delta + 1\}$.

Remarks:

- In Definition 1.11 we will see what is meant by "step".
- Sometimes $\chi(G) \ll \Delta + 1$.

Definition 1.8 (Synchronous Distributed Algorithm)**.** *In a synchronous distributed algorithm, nodes operate in synchronous rounds. In each round, each node executes the following steps:*

1. *Send messages to neighbors in graph (of reasonable size).*

2. *Receive messages (that were sent by neighbors in step 1 of the same round).*
3. *Do some local computation (of reasonable complexity).*

Remarks:

- Any other step ordering is fine.
- What does "reasonable" mean in this context? We are somewhat flexible here, and different model variants exist. Generally, we will deal with algorithms that only do very simple computations (a comparison, an addition, etc.). Exponential-time computation is usually considered cheating in this context. Similarly, sending a message with a node ID, or a value is considered okay, whereas sending really long messages is fishy. We will have more exact definitions later, when we need them.
- We can build a distributed version of Algorithm 1.5:

Algorithm 1.9 Reduce

```
1: Assume that initially all nodes have IDs
2: Each node v executes the following code:
3: node v sends its ID to all neighbors
4: node v receives IDs of neighbors
5: while node v has an uncolored neighbor with higher ID do
6:     node v sends "undecided" to all neighbors
7:     node v receives new decisions from neighbors
8: end while
9: node v chooses the smallest admissible free color
10: node v informs all its neighbors about its choice
```

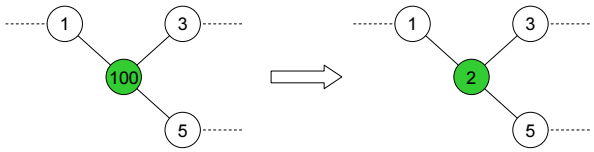


Figure 1.10: Vertex 100 receives the lowest possible color.

Definition 1.11 (Time Complexity)**.** *For synchronous algorithms (as defined in 1.8) the* time complexity *is the number of rounds until the algorithm terminates. The algorithm terminates when the last node terminates.*

Theorem 1.12. *Algorithm 1.9 is correct and has time complexity n. The algorithm uses at most $\Delta + 1$ colors.*

Proof. Nodes choose colors that are different from their neighbors, and no two neighbors choose concurrently. In each round at least one node chooses a color, so we are done after at most n rounds. □

Remarks:

- In the worst case, this algorithm is still not better than sequential.
- Moreover, it seems difficult to come up with a fast algorithm.
- Maybe it's better to first study a simple special case, a tree, and then go from there.

1.2 Coloring Trees

Lemma 1.13. $\chi(Tree) \leq 2$

Proof. Call some node the root of the tree. If the distance of a node to the root is odd (even), color it 1 (0). An odd node has only even neighbors and vice versa. □

Remarks:

- If we assume that each node knows its parent (root has no parent) and children in a tree, this constructive proof gives a very simple algorithm:

Algorithm 1.14 Slow Tree Coloring

```
1: Color the root 0, root sends 0 to its children
2: Each node v concurrently executes the following code:
3: if node v receives a message c_p (from parent) then
4:     node v chooses color c_v = 1 − c_p
5:     node v sends c_v to its children (all neighbors except parent)
6: end if
```

Theorem 1.15. *Algorithm 1.14 is correct. If each node knows its parent and its children, the time complexity is the tree height which is bounded by the diameter of the tree.*

Remarks:

- How can we determine a root in a tree if it is not already given? We will figure that out later.
- The time complexity of the algorithm is the height of the tree.
- Nice trees, e.g., balanced binary trees, have logarithmic height, that is we have a logarithmic time complexity.
- However, if the tree has a degenerated topology, the time complexity may again be up to n, the number of nodes.
- This algorithm is not very exciting. Can we do better than logarithmic?

Here is the idea of the algorithm: We start with color labels that have $\log n$ bits. In each round we compute a new label with exponentially smaller size than the previous label, still guaranteeing to have a valid vertex coloring! The algorithm terminates in $\log^* n$ time. Log-Star?! That's the number of logarithms (to the base 2) you need to take to get down to 2. Formally:

Definition 1.16 (Log-Star)**.**
$\forall x \leq 2 :\ \log^* x := 1 \quad \forall x > 2 :\ \log^* x := 1 + \log^*(\log x)$

Remarks:

- Log-star is an amazingly slowly growing function. Log-star of all the atoms in the observable universe (estimated to be 10^{80}) is 5. So log-star increases indeed very slowly! There are functions which grow even more slowly, such as the inverse Ackermann function, however, the inverse Ackermann function of all the atoms is already 4.

Algorithm 1.17 "6-Color"

```
1: Assume that initially the nodes have IDs of size log n bits
2: The root assigns itself the label 0
3: Each other node v executes the following code
4: send own color c_v to all children
5: repeat
6:    receive color c_p from parent
7:    interpret c_v and c_p as bit-strings
8:    let i be the index of the smallest bit where c_v and c_p differ
9:    the new label is i (as bitstring) followed by the i^th bit of c_v
10:   send c_v to all children
11: until c_w ∈ {0,...,5} for all nodes w
```

Example:

Algorithm 1.17 executed on the following part of a tree:

Grand-parent	0010110000	$\rightarrow$	10010	$\rightarrow$	...
Parent	1010010000	$\rightarrow$	01010	$\rightarrow$	111
Child	0110010000	$\rightarrow$	10001	$\rightarrow$	001

Theorem 1.18. *Algorithm 1.17 terminates in $\log^* n + k$ time, where k is a constant independent of n.*

Proof. We need to show that parent p and child c always have different colors. Initially, this is true, since all nodes start out with their unique ID. In a round, let i be the smallest index where child c has a different bit from parent p. If parent p differs in a different index bit $j \neq i$ from its own parent, parent and child will compute different colors in that round. On the other hand, if $j = i$, the symmetry is broken by p having a different bit at index i.

Regarding runtime, note that the size of the largest color shrinks dramatically in each round, apart from the symmetry-breaking bit, exactly as a logarithmic function. With some (tedious and boring) machinery, one can show

that indeed every node will have a color in the range $\{0, \ldots, 5\}$ in $\log^* n + k$ rounds. □

Remarks:

- Let us have a closer look at the end game of the algorithm. Colors $11*$ (in binary notation, i.e., 6 or 7 in decimal notation) will not be chosen, because the node will then do another round. This gives a total of 6 colors (i.e., colors 0,..., 5).
- What about that last line of the loop? How do the nodes know that all nodes now have a color in the range $\{0, \ldots, 5\}$? The answer to this question is surprisingly complex. One may hardwire the number of rounds into the until statement, such that all nodes execute the loop for exactly the same number of rounds. However, in order to do so, all nodes need to know n, the number of nodes, which is ugly. There are (non-trivial) solutions where nodes do not need to know n, see exercises.
- Can one reduce the number of colors? Note that Algorithm 1.9 does not work (since the degree of a node can be much higher than 6)! For fewer colors we need to have siblings monochromatic!

Algorithm 1.19 Shift Down

```
1: Each other node v concurrently executes the following code:
2: Recolor v with the color of parent
3: Root chooses a new (different) color from {0,1,2}
```

Lemma 1.20. *Algorithm 1.19 preserves coloring legality; also siblings are monochromatic.*

Now Algorithm 1.9 can be used to reduce the number of used colors from 6 to 3.

Algorithm 1.21 Six-2-Three

```
1: Each node v concurrently executes the following code:
2: for x = 5,4,3 do
3:    Perform subroutine Shift down (Algorithm 1.19)
4:    if c_v = x then
5:       choose the smallest admissible new color c_v ∈ {0,1,2}
6:    end if
7: end for
```

Theorem 1.23. *Algorithms 1.17 and 1.21 color a tree with three colors in time $\mathcal{O}(\log^* n)$.*

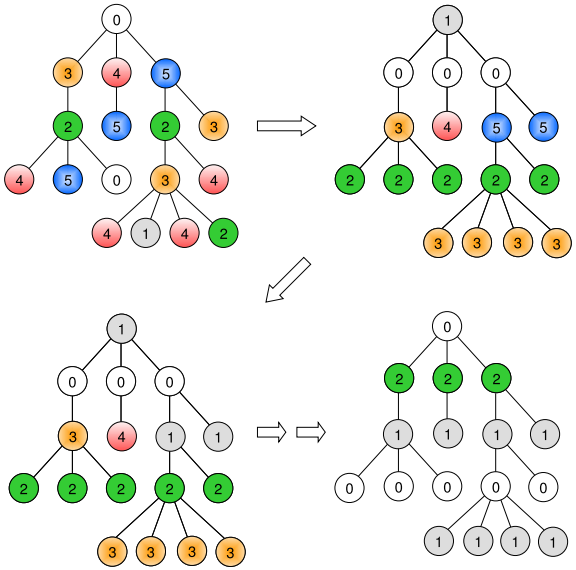


Figure 1.22: Possible execution of Algorithm 1.21.

Remarks:

- The term $\mathcal{O}()$ used in Theorem 1.18 is called "big O" and is often used in distributed computing. Roughly speaking, $\mathcal{O}(f)$ means "in the order of f, ignoring constant factors and smaller additive terms." More formally, for two functions f and g, it holds that $f \in O(g)$ if there are constants x_0 and c so that $|f(x)| \leq c|g(x)|$ for all $x \geq x_0$. For an elaborate discussion on the big O notation we refer to other introductory math or computer science classes, or Wikipedia.
- A fast tree-coloring with only 2 colors is more than exponentially more expensive than coloring with 3 colors. In a tree degenerated to a list, nodes far away need to figure out whether they are an even or odd number of hops away from each other in order to get a 2-coloring. To do that one has to send a message to these nodes. This costs time linear in the number of nodes.
- The idea of this algorithm can be generalized, e.g., to a ring topology. Also a general graph with constant degree Δ can be colored with $\Delta + 1$ colors in $\mathcal{O}(\log^* n)$ time. The idea is as follows: In each step, a node compares its label to each of its neighbors, constructing a logarithmic difference-tag as in Algorithm 1.17. Then the new label is the concatenation of all the difference-tags. For constant degree Δ, this gives a 3Δ-label in $\mathcal{O}(\log^* n)$ steps. Algorithm 1.9 then reduces the number of colors to $\Delta+1$ in $2^{3\Delta}$ (this is still a constant for constant Δ!) steps.

- Unfortunately, coloring a general graph is not yet possible with this technique. We will see another technique for that in Chapter 7. With this technique it is possible to color a general graph with $\Delta + 1$ colors in $\mathcal{O}(\log n)$ time.
- A lower bound shows that many of these log-star algorithms are asymptotically (up to constant factors) optimal. We will see that later.

Chapter Notes

The basic technique of the log-star algorithm is by Cole and Vishkin [CV86]. A tight bound of $\frac{1}{2}\log^* n$ was proven recently [RS15]. The technique can be generalized and extended, e.g., to a ring topology or to graphs with constant degree [GP87, GPS88, KMW05]. Using it as a subroutine, one can solve many problems in log-star time. For instance, one can color so-called growth bounded graphs (a model which includes many natural graph classes, for instance unit disk graphs) asymptotically optimally in $\mathcal{O}(\log^* n)$ time [SW08]. Actually, Schneider et al. show that many classic combinatorial problems beyond coloring can be solved in log-star time in growth bounded and other restricted graphs.

In a later chapter we learn a $\Omega(\log^* n)$ lower bound for coloring and related problems [Lin92]. Linial's paper also contains a number of other results on coloring, e.g., that any algorithm for coloring d-regular trees of radius r that run in time at most $2r/3$ require at least $\Omega(\sqrt{d})$ colors.

For general graphs, later we will learn fast coloring algorithms that use a maximal independent sets as a base. Since coloring exhibits a trade-off between efficacy and efficiency, many different results for general graphs exist, e.g., [PS96, KSOS06, BE09, Kuh09, SW10, BE11b, KP11, BE11a, BEPS12, PS13, CPS14, BEK14].

Some parts of this chapter are also discussed in Chapter 7 of [Pel00], e.g., the proof of Theorem 1.18.

Bibliography

[BE09] Leonid Barenboim and Michael Elkin. Distributed (delta+1)-coloring in linear (in delta) time. In *41st ACM Symposium On Theory of Computing (STOC)*, 2009.

[BE11a] Leonid Barenboim and Michael Elkin. Combinatorial Algorithms for Distributed Graph Coloring. In *25th International Symposium on DIStributed Computing*, 2011.

[BE11b] Leonid Barenboim and Michael Elkin. Deterministic Distributed Vertex Coloring in Polylogarithmic Time. *J. ACM*, 58(5):23, 2011.

[BEK14] Leonid Barenboim, Michael Elkin, and Fabian Kuhn. Distributed (delta+1)-coloring in linear (in delta) time. *SIAM J. Comput.*, 43(1):72–95, 2014.

[BEPS12] Leonid Barenboim, Michael Elkin, Seth Pettie, and Johannes Schneider. The locality of distributed symmetry breaking. In *Foundations of Computer Science (FOCS), 2012 IEEE 53rd Annual Symposium on*, pages 321–330, 2012.

[CPS14] Kai-Min Chung, Seth Pettie, and Hsin-Hao Su. Distributed algorithms for the lovász local lemma and graph coloring. In *ACM Symposium on Principles of Distributed Computing*, pages 134–143, 2014.

[CV86] R. Cole and U. Vishkin. Deterministic coin tossing and accelerating cascades: micro and macro techniques for designing parallel algorithms. In *18th annual ACM Symposium on Theory of Computing (STOC)*, 1986.

[GP87] Andrew V. Goldberg and Serge A. Plotkin. Parallel (Δ+1)-coloring of constant-degree graphs. *Inf. Process. Lett.*, 25(4):241–245, June 1987.

[GPS88] Andrew V. Goldberg, Serge A. Plotkin, and Gregory E. Shannon. Parallel Symmetry-Breaking in Sparse Graphs. *SIAM J. Discrete Math.*, 1(4):434–446, 1988.

[KMW05] Fabian Kuhn, Thomas Moscibroda, and Roger Wattenhofer. On the Locality of Bounded Growth. In *24th ACM Symposium on the Principles of Distributed Computing (PODC), Las Vegas, Nevada, USA*, July 2005.

[KP11] Kishore Kothapalli and Sriram V. Pemmaraju. Distributed graph coloring in a few rounds. In *30th ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing (PODC)*, 2011.

[KSOS06] Kishore Kothapalli, Christian Scheideler, Melih Onus, and Christian Schindelhauer. Distributed coloring in $O(\sqrt{\log n})$ Bit Rounds. In *20th international conference on Parallel and Distributed Processing (IPDPS)*, 2006.

[Kuh09] Fabian Kuhn. Weak graph colorings: distributed algorithms and applications. In *21st ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, 2009.

[Lin92] N. Linial. Locality in Distributed Graph Algorithms. *SIAM Journal on Computing*, 21(1)(1):193–201, February 1992.

[Pel00] David Peleg. *Distributed Computing: a Locality-Sensitive Approach*. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 2000.

[PS96] Alessandro Panconesi and Aravind Srinivasan. On the Complexity of Distributed Network Decomposition. *J. Algorithms*, 20(2):356–374, 1996.

[PS13] Seth Pettie and Hsin-Hao Su. Fast distributed coloring algorithms for triangle-free graphs. In *Automata, Languages, and Programming - 40th International Colloquium, ICALP*, pages 681–693, 2013.

[RS15] Joel Rybicki and Jukka Suomela. Exact bounds for distributed graph colouring. In *Structural Information and Communication Complexity - 22nd International Colloquium, SIROCCO 2015, Montserrat, Spain, July 14-16, 2015, Post-Proceedings*, pages 46–60, 2015.

[SW08] Johannes Schneider and Roger Wattenhofer. A Log-Star Distributed Maximal Independent Set Algorithm for Growth-Bounded Graphs. In *27th ACM Symposium on Principles of Distributed Computing (PODC), Toronto, Canada*, August 2008.

[SW10] Johannes Schneider and Roger Wattenhofer. A New Technique For Distributed Symmetry Breaking. In *29th Symposium on Principles of Distributed Computing (PODC), Zurich, Switzerland*, July 2010.

Chapter 2

Tree Algorithms

In this chapter we learn a few basic algorithms on trees, and how to construct trees in the first place so that we can run these (and other) algorithms. The good news is that these algorithms have many applications, the bad news is that this chapter is a bit on the simple side. But maybe that's not really bad news?!

2.1 Broadcast

Definition 2.1 (Broadcast)**.** *A* broadcast *operation is initiated by a single node, the source. The source wants to send a message to all other nodes in the system.*

Definition 2.2 (Distance, Radius, Diameter)**.** *The* distance *between two nodes u and v in an undirected graph G is the number of hops of a minimum path between u and v. The* radius *of a node u is the maximum distance between u and any other node in the graph. The radius of a graph is the minimum radius of any node in the graph. The* diameter *of a graph is the maximum distance between two arbitrary nodes.*

Remarks:

- Clearly there is a close relation between the radius R and the diameter D of a graph, such as $R \leq D \leq 2R$.

Definition 2.3 (Message Complexity)**.** *The* message complexity *of an algorithm is determined by the total number of messages exchanged.*

Theorem 2.4 (Broadcast Lower Bound)**.** *The message complexity of broadcast is at least $n-1$. The source's radius is a lower bound for the time complexity.*

Proof: Every node must receive the message.

Remarks:

- You can use a pre-computed spanning tree to do broadcast with tight message complexity. If the spanning tree is a breadth-first search spanning tree (for a given source), then the time complexity is tight as well.

Definition 2.5 (Clean)**.** *A graph (network) is* clean *if the nodes do not know the topology of the graph.*

Theorem 2.6 (Clean Broadcast Lower Bound)**.** *For a clean network, the number of edges m is a lower bound for the broadcast message complexity.*

Proof: If you do not try every edge, you might miss a whole part of the graph behind it.

Definition 2.7 (Asynchronous Distributed Algorithm)**.** *In the asynchronous model, algorithms are event driven ("upon receiving message ..., do ..."). Nodes cannot access a global clock. A message sent from one node to another will arrive in finite but unbounded time.*

Remarks:

- The asynchronous model and the synchronous model (Definition 1.8) are the cornerstone models in distributed computing. As they do not necessarily reflect reality there are several models in between synchronous and asynchronous. However, from a theoretical point of view the synchronous and the asynchronous model are the most interesting ones (because every other model is in between these extremes).

- Note that in the asynchronous model, messages that take a longer path may arrive earlier.

Definition 2.8 (Asynchronous Time Complexity)**.** *For asynchronous algorithms (as defined in 2.7) the* time complexity *is the number of time units from the start of the execution to its completion in the worst case (every legal input, every execution scenario), assuming that each message has a delay of at most one time unit.*

Remarks:

- You cannot use the maximum delay in the algorithm design. In other words, the algorithm has to be correct even if there is no such delay upper bound.

- The clean broadcast lower bound (Theorem 2.6) directly brings us to the well known *flooding* algorithm.

Algorithm 2.9 Flooding

1: The source (root) sends the message to all neighbors.
2: **Each other node** v upon receiving the message the first time forwards the message to all (other) neighbors.
3: Upon later receiving the message again (over other edges), a node can discard the message.

Remarks:

- If node v receives the message first from node u, then node v calls node u *parent*. This parent relation defines a spanning tree T. If the flooding algorithm is executed in a synchronous system, then T is a breadth-first search spanning tree (with respect to the root).
- More interestingly, also in asynchronous systems the flooding algorithm terminates after R time units, R being the radius of the source. However, the constructed spanning tree may not be a breadth-first search spanning tree.

2.2 Convergecast

Convergecast is the same as broadcast, just reversed: Instead of a root sending a message to all other nodes, all other nodes send information to a root (starting from the leaves, i.e., the tree T is known). The simplest convergecast algorithm is the echo algorithm:

Algorithm 2.10 Echo

1: A leave sends a message to its parent.
2: If an inner node has received a message from each child, it sends a message to the parent.

Remarks:

- Usually the echo algorithm is paired with the flooding algorithm, which is used to let the leaves know that they should start the echo process; this is known as flooding/echo.
- One can use convergecast for termination detection, for example. If a root wants to know whether all nodes in the system have finished some task, it initiates a flooding/echo; the message in the echo algorithm then means "This subtree has finished the task."
- Message complexity of the echo algorithm is $n-1$, but together with flooding it is $\mathcal{O}(m)$, where $m = |E|$ is the number of edges in the graph.
- The time complexity of the echo algorithm is determined by the depth of the spanning tree (i.e., the radius of the root within the tree) generated by the flooding algorithm.
- The flooding/echo algorithm can do much more than collecting acknowledgements from subtrees. One can for instance use it to compute the number of nodes in the system, or the maximum ID, or the sum of all values stored in the system, or a route-disjoint matching.
- Moreover, by combining results one can compute even fancier aggregations, e.g., with the number of nodes and the sum one can compute the average. With the average one can compute the standard deviation. And so on ...

2.3 BFS Tree Construction

In synchronous systems the flooding algorithm is a simple yet efficient method to construct a breadth-first search (BFS) spanning tree. However, in asynchronous systems the spanning tree constructed by the flooding algorithm may be far from BFS. In this section, we implement two classic BFS constructions—Dijkstra and Bellman-Ford—as asynchronous algorithms.

We start with the Dijkstra algorithm. The basic idea is to always add the "closest" node to the existing part of the BFS tree. We need to parallelize this idea by developing the BFS tree layer by layer. The algorithm proceeds in phases. In phase p the nodes with distance p to the root are detected. Let T_p be the tree in phase p.

Algorithm 2.11 Dijkstra BFS

1: We start with T_1 which is the root plus all direct neighbors of the root. We start with phase $p = 1$:
2: **repeat**
3: The root starts phase p by broadcasting "start p" within T_p.
4: When receiving "start p" a leaf node u of T_p (that is, a node that was newly discovered in the last phase) sends a "join $p+1$" message to all quiet neighbors. (A neighbor v is quiet if u has not yet "talked" to v.)
5: A node v receiving the first "join p+1" message replies with "ACK" and becomes a leaf of the tree T_{p+1}.
6: A node v receiving any further "join" message replies with "NACK".
7: The leaves of T_p collect all the answers of their neighbors; then the leaves start an echo algorithm back to the root.
8: When the echo process terminates at the root, the root increments the phase
9: **until** there was no new node detected

Theorem 2.12. *The time complexity of Algorithm 2.11 is $\mathcal{O}(D^2)$, the message complexity is $\mathcal{O}(m+nD)$, where D is the diameter of the graph, n the number of nodes, and m the number of edges.*

Proof: A broadcast/echo algorithm in T_p needs at most time $2D$. Finding new neighbors at the leaves costs 2 time units. Since the BFS tree height is bounded by the diameter, we have D phases, giving a total time complexity of $\mathcal{O}(D^2)$. Each node participating in broadcast/echo only receives (broadcasts) at most 1 message and sends (echoes) at most once. Since there are D phases, the cost is bounded by $\mathcal{O}(nD)$. On each edge there are at most 2 "join" messages. Replies to a "join" request are answered by 1 "ACK" or "NACK" , which means that we have at most 4 additional messages per edge. Therefore the message complexity is $\mathcal{O}(m+nD)$.

Remarks:

- The time complexity is not very exciting, so let's try Bellman-Ford!

The basic idea of Bellman-Ford is even simpler, and heavily used in the Internet, as it is a basic version of the omnipresent border gateway protocol (BGP). The idea is to simply keep the distance to the root accurate. If a

neighbor has found a better route to the root, a node might also need to update its distance.

Algorithm 2.13 Bellman-Ford BFS

1: Each node u stores an integer d_u which corresponds to the distance from u to the root. Initially $d_{\text{root}} = 0$, and $d_u = \infty$ for every other node u.
2: The root starts the algorithm by sending "1" to all neighbors.
3: **if** a node u receives a message "y" with $y < d_u$ from a neighbor v **then**
4: node u sets $d_u := y$
5: node u sends "$y + 1$" to all neighbors (except v)
6: **end if**

Theorem 2.14. *The time complexity of Algorithm 2.13 is $\mathcal{O}(D)$, the message complexity is $\mathcal{O}(nm)$, where D, n, m are defined as in Theorem 2.12.*

Proof: We can prove the time complexity by induction. We claim that a node at distance d from the root has received a message "d" by time d. The root knows by time 0 that it is the root. A node v at distance d has a neighbor u at distance $d-1$. Node u by induction sends a message "d" to v at time $d-1$ or before, which is then received by v at time d or before. Message complexity is easier: A node can reduce its distance at most $n-1$ times; each of these times it sends a message to all its neighbors. If all nodes do this, then we have $\mathcal{O}(nm)$ messages.

Remarks:

- Algorithm 2.11 has the better message complexity and Algorithm 2.13 has the better time complexity. The currently best algorithm (optimizing both) needs $\mathcal{O}(m + n\log^3 n)$ messages and $\mathcal{O}(D\log^3 n)$ time. This "trade-off" algorithm is beyond the scope of this chapter, but we will later learn the general technique.

2.4 MST Construction

There are several types of spanning trees, each serving a different purpose. A particularly interesting spanning tree is the minimum spanning tree (MST). The MST only makes sense on weighted graphs, hence in this section we assume that each edge e is assigned a weight ω_e.

Definition 2.15 (MST)**.** *Given a weighted graph $G = (V, E, \omega)$, the MST of G is a spanning tree T minimizing $\omega(T)$, where $\omega(G') = \sum_{e \in G'} \omega_e$ for any subgraph $G' \subseteq G$.*

Remarks:

- In the following we assume that no two edges of the graph have the same weight. This simplifies the problem as it makes the MST unique; however, this simplification is not essential as one can always break ties by adding the IDs of adjacent vertices to the weight.
- Obviously we are interested in computing the MST in a distributed way. For this we use a well-known lemma:

Definition 2.16 (Blue Edges)**.** *Let T be a spanning tree of the weighted graph G and $T' \subseteq T$ a subgraph of T (also called a* fragment*). Edge $e = (u, v)$ is an* outgoing edge *of T' if $u \in T'$ and $v \notin T'$ (or vice versa). The minimum weight outgoing edge $b(T')$ is the so-called* blue edge *of T'.*

Lemma 2.17. *For a given weighted graph G (such that no two weights are the same), let T denote the MST, and T' be a fragment of T. Then the blue edge of T' is also part of T, i.e., $T' \cup b(T') \subseteq T$.*

Proof: For the sake of contradiction, suppose that in the MST T there is edge $e \neq b(T')$ connecting T' with the remainder of T. Adding the blue edge $b(T')$ to the MST T we get a cycle including both e and $b(T')$. If we remove e from this cycle, then we still have a spanning tree, and since by the definition of the blue edge $\omega_e > \omega_{b(T')}$, the weight of that new spanning tree is less than than the weight of T. We have a contradiction.

Remarks:

- In other words, the blue edges seem to be the key to a distributed algorithm for the MST problem. Since every node itself is a fragment of the MST, every node directly has a blue edge! All we need to do is to grow these fragments! Essentially this is a distributed version of Kruskal's sequential algorithm.
- At any given time the nodes of the graph are partitioned into fragments (rooted subtrees of the MST). Each fragment has a root, the ID of the fragment is the ID of its root. Each node knows its parent and its children in the fragment. The algorithm operates in phases. At the beginning of a phase, nodes know the IDs of the fragments of their neighbor nodes.

Remarks:

- Algorithm 2.18 was stated in pseudo-code, with a few details not really explained. For instance, it may be that some fragments are much larger than others, and because of that some nodes may need to wait for others, e.g., if node u needs to find out whether neighbor v also wants to merge over the blue edge $b = (u, v)$. The good news is that all these details can be solved. We can for instance bound the asynchronicity by guaranteeing that nodes only start the new phase after the last phase is done, similarly to the phase-technique of Algorithm 2.11.

Theorem 2.19. *The time complexity of Algorithm 2.18 is $\mathcal{O}(n \log n)$, the message complexity is $\mathcal{O}(m \log n)$.*

Proof: Each phase mainly consists of two flooding/echo processes. In general, the cost of flooding/echo on a tree is $\mathcal{O}(D)$ time and $\mathcal{O}(n)$ messages. However, the diameter D of the fragments may turn out to be not related to the diameter of the graph because the MST may meander, hence it really is $\mathcal{O}(n)$ time. In addition, in the first step of each phase, nodes need to learn the fragment ID of their neighbors; this can be done in 2 steps but costs $\mathcal{O}(m)$ messages. There are a few more steps, but they are cheap. Altogether a phase costs $\mathcal{O}(n)$ time and

Algorithm 2.18 GHS (Gallager–Humblet–Spira)

1: Initially each node is the root of its own fragment. We proceed in phases:
2: **repeat**
3: All nodes learn the fragment IDs of their neighbors.
4: The root of each fragment uses flooding/echo in its fragment to determine the blue edge $b = (u, v)$ of the fragment.
5: The root sends a message to node u; while forwarding the message on the path from the root to node u all parent-child relations are inverted {such that u is the new temporary root of the fragment}
6: node u sends a merge request over the blue edge $b = (u, v)$.
7: **if** node v also sent a merge request over the same blue edge $b = (v, u)$ **then**
8: either u or v (whichever has the smaller ID) is the new fragment root
9: the blue edge b is directed accordingly
10: **else**
11: node v is the new parent of node u
12: **end if**
13: the newly elected root node informs all nodes in its fragment (again using flooding/echo) about its identity
14: **until** all nodes are in the same fragment (i.e., there is no outgoing edge)

$\mathcal{O}(m)$ messages. So we only have to figure out the number of phases: Initially all fragments are single nodes and hence have size 1. In a later phase, each fragment merges with at least one other fragment, that is, the size of the smallest fragment at least doubles. In other words, we have at most $\log n$ phases. The theorem follows directly.

Chapter Notes

Trees are one of the oldest graph structures, already appearing in the first book about graph theory [Koe36]. Broadcasting in distributed computing is younger, but not that much [DM78]. Overviews about broadcasting can be found for example in Chapter 3 of [Pel00] and Chapter 7 of [HKP+05]. For a introduction to centralized tree-construction, see e.g. [Eve79] or [CLRS09]. Overviews for the distributed case can be found in Chapter 5 of [Pel00] or Chapter 4 of [Lyn96]. The classic papers on routing are [For56, Bel58, Dij59]. In a later chapter, we will later learn a general technique to derive algorithms with an almost optimal time and message complexity.

Algorithm 2.18 is called "GHS" after Gallager, Humblet, and Spira, three pioneers in distributed computing [GHS83]. Their algorithm won the prestigious Edsger W. Dijkstra Prize in Distributed Computing in 2004, among other reasons because it was one of the first non-trivial asynchronous distributed algorithms. As such it can be seen as one of the seeds of this research area. We presented a simplified version of GHS. The original paper featured an improved message complexity of $\mathcal{O}(m + n \log n)$. Later, Awerbuch managed to further improve the GHS algorithm to get $\mathcal{O}(n)$ time and $\mathcal{O}(m + n \log n)$ message complexity, both asymptotically optimal [Awe87].

Bibliography

[Awe87] B. Awerbuch. Optimal distributed algorithms for minimum weight spanning tree, counting, leader election, and related problems. In *Proceedings of the nineteenth annual ACM symposium on Theory of computing*, STOC '87, pages 230–240, New York, NY, USA, 1987. ACM.

[Bel58] Richard Bellman. On a Routing Problem. *Quarterly of Applied Mathematics*, 16:87–90, 1958.

[CLRS09] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms (3. ed.)*. MIT Press, 2009.

[Dij59] E. W. Dijkstra. A Note on Two Problems in Connexion with Graphs. *Numerische Mathematik*, 1(1):269–271, 1959.

[DM78] Y.K. Dalal and R.M. Metcalfe. Reverse path forwarding of broadcast packets. *Communications of the ACM*, 12:1040–148, 1978.

[Eve79] S. Even. *Graph Algorithms*. Computer Science Press, Rockville, MD, 1979.

[For56] Lester R. Ford. Network Flow Theory. *The RAND Corporation Paper P-923*, 1956.

[GHS83] R. G. Gallager, P. A. Humblet, and P. M. Spira. Distributed Algorithm for Minimum-Weight Spanning Trees. *ACM Transactions on Programming Languages and Systems*, 5(1):66–77, January 1983.

[HKP+05] Juraj Hromkovic, Ralf Klasing, Andrzej Pelc, Peter Ruzicka, and Walter Unger. *Dissemination of Information in Communication Networks - Broadcasting, Gossiping, Leader Election, and Fault-Tolerance*. Texts in Theoretical Computer Science. An EATCS Series. Springer, 2005.

[Koe36] Denes Koenig. *Theorie der endlichen und unendlichen Graphen*. Teubner, Leipzig, 1936.

[Lyn96] Nancy A. Lynch. *Distributed Algorithms*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1996.

[Pel00] David Peleg. *Distributed Computing: a Locality-Sensitive Approach*. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 2000.

Chapter 3

Leader Election

Some algorithms (e.g. the slow tree coloring Algorithm 1.14) ask for a special node, a so-called "leader". Computing a leader is a very simple form of symmetry breaking. Algorithms based on leaders do generally not exhibit a high degree of parallelism, and therefore often suffer from poor time complexity. However, sometimes it is still useful to have a leader to make critical decisions in an easy (though non-distributed!) way.

3.1 Anonymous Leader Election

The process of choosing a leader is known as *leader election*. Although leader election is a simple form of symmetry breaking, there are some remarkable issues that allow us to introduce notable computational models.

In this chapter we concentrate on the ring topology. Many interesting challenges in distributed computing already reveal the root of the problem in the special case of the ring. Paying attention to the ring also makes sense from a practical point of view as some real world systems are based on a ring topology, e.g., the antiquated token ring standard.

Problem 3.1 (Leader Election)**.** *Each node eventually decides whether it is a leader or not, subject to the constraint that there is exactly one leader.*

Remarks:

- More formally, nodes are in one of three states: undecided, leader, not leader. Initially every node is in the undecided state. When leaving the undecided state, a node goes into a final state (leader or not leader).

Definition 3.2 (Anonymous)**.** *A system is anonymous if nodes do not have unique identifiers.*

Definition 3.3 (Uniform)**.** *An algorithm is called uniform if the number of nodes n is not known to the algorithm (to the nodes, if you wish). If n is known, the algorithm is called non-uniform.*

Whether a leader can be elected in an anonymous system depends on whether the network is symmetric (ring, complete graph, complete bipartite graph, etc.)

or asymmetric (star, single node with highest degree, etc.). We will now show that non-uniform anonymous leader election for synchronous rings is impossible. The idea is that in a ring, symmetry can always be maintained.

Lemma 3.4. *After round k of any deterministic algorithm on an anonymous ring, each node is in the same state s_k.*

Proof by induction: All nodes start in the same state. A round in a synchronous algorithm consists of the three steps sending, receiving, local computation (see Definition 1.8). All nodes send the same message(s), receive the same message(s), do the same local computation, and therefore end up in the same state.

Theorem 3.5 (Anonymous Leader Election)**.** *Deterministic leader election in an anonymous ring is impossible.*

Proof (with Lemma 3.4): If one node ever decides to become a leader (or a non-leader), then every other node does so as well, contradicting the problem specification 3.1 for $n > 1$. This holds for non-uniform algorithms, and therefore also for uniform algorithms. Furthermore, it holds for synchronous algorithms, and therefore also for asynchronous algorithms.

Remarks:

- Sense of direction is the ability of nodes to distinguish neighbor nodes in an anonymous setting. In a ring, for example, a node can distinguish the clockwise and the counterclockwise neighbor. Sense of direction does not help in anonymous leader election.
- Theorem 3.5 also holds for other symmetric network topologies (e.g., complete graphs, complete bipartite graphs, ...).
- Note that Theorem 3.5 does generally not hold for randomized algorithms; if nodes are allowed to toss a coin, some symmetries can be broken.
- However, more surprisingly, randomization does not always help. A randomized uniform anonymous algorithm can for instance not elect a leader in a ring. Randomization does not help to decide whether the ring has $n = 3$ or $n = 6$ nodes: Every third node may generate the same random bits, and as a result the nodes cannot distinguish the two cases. However, an approximation of n which is strictly better than a factor 2 will help.

3.2 Asynchronous Ring

We first concentrate on the asynchronous model from Definition 2.7. Throughout this section we assume non-anonymity; each node has a unique identifier. Having IDs seems to lead to a trivial leader election algorithm, as we can simply elect the node with, e.g., the highest ID.

Theorem 3.7. *Algorithm 3.6 is correct. The time complexity is $\mathcal{O}(n)$. The message complexity is $\mathcal{O}(n^2)$.*

Algorithm 3.6 Clockwise Leader Election

1: **Each node** v executes the following code:
2: v sends a message with its identifier (for simplicity also v) to its clockwise neighbor.
3: v sets $m := v$ {the largest identifier seen so far}
4: **if** v receives a message w with $w > m$ **then**
5: v forwards message w to its clockwise neighbor and sets $m := w$
6: v decides not to be the leader, if it has not done so already.
7: **else if** v receives its own identifier v **then**
8: v decides to be the leader
9: **end if**

Proof: Let node z be the node with the maximum identifier. Node z sends its identifier in clockwise direction, and since no other node can swallow it, eventually a message will arrive at z containing it. Then z declares itself to be the leader. Every other node will declare non-leader at the latest when forwarding message z. Since there are n identifiers in the system, each node will at most forward n messages, giving a message complexity of at most n^2. We start measuring the time when the first node that "wakes up" sends its identifier. For asynchronous time complexity (Definition 2.8) we assume that each message takes at most one time unit to arrive at its destination. After at most $n-1$ time units the message therefore arrives at node z, waking z up. Routing the message z around the ring takes at most n time units. Therefore node z decides no later than at time $2n-1$. Every other node decides before node z.

Remarks:

- Note that in Algorithm 3.6 nodes distinguish between clockwise and counterclockwise neighbors. This is not necessary: It is okay to simply send your own identifier to any neighbor, and forward a message to the neighbor you did not receive the message from. So nodes only need to be able to distinguish their two neighbors.
- Careful analysis shows, that while having worst-case message complexity of $\mathcal{O}(n^2)$, Algorithm 3.6 has an *average* message complexity of $\mathcal{O}(n \log n)$. Can we improve this algorithm?

Theorem 3.9. *Algorithm 3.8 is correct. The time complexity is $\mathcal{O}(n)$. The message complexity is $\mathcal{O}(n \log n)$.*

Proof: Correctness is as in Theorem 3.7. The time complexity is $\mathcal{O}(n)$ since the node with maximum identifier z sends messages with round-trip times $2, 4, 8, 16, \ldots, 2 \cdot 2^k$ with $k \leq \log(n+1)$. (Even if we include the additional wake-up overhead, the time complexity stays linear.) Proving the message complexity is slightly harder: if a node v manages to survive round r, no other node in distance 2^r (or less) survives round r. That is, node v is the only node in its 2^r-neighborhood that remains active in round $r+1$. Since this is the same for every node, less than $n/2^r$ nodes are active in round $r+1$. Being active in round r costs $2 \cdot 2 \cdot 2^r$ messages. Therefore, round r costs at most $2 \cdot 2 \cdot 2^r \cdot \frac{n}{2^{r-1}} = 8n$ messages. Since there are only logarithmic many possible rounds, the message complexity follows immediately.

Algorithm 3.8 Radius Growth

1: **Each node** v does the following:
2: Initially all nodes are *active*. {all nodes may still become leaders}
3: Whenever a node v sees a message w with $w > v$, then v decides to not be a leader and becomes *passive*.
4: Active nodes search in an exponentially growing neighborhood (clockwise and counterclockwise) for nodes with higher identifiers, by sending out *probe* messages. A probe message includes the ID of the original sender, a bit whether the sender can still become a leader, and a time-to-live number (TTL). The first probe message sent by node v includes a TTL of 1.
5: Nodes (active or passive) receiving a probe message decrement the TTL and forward the message to the next neighbor; if their ID is larger than the one in the message, they set the leader bit to zero, as the probing node does not have the maximum ID. If the TTL is zero, probe messages are returned to the sender using a *reply* message. The reply message contains the ID of the receiver (the original sender of the probe message) and the leader-bit. Reply messages are forwarded by all nodes until they reach the receiver.
6: Upon receiving the reply message: If there was no node with higher ID in the search area (indicated by the bit in the reply message), the TTL is doubled and two new probe messages are sent (again to the two neighbors). If there was a better candidate in the search area, then the node becomes passive.
7: If a node v receives its own probe message (not a reply) v decides to be the leader.

Remarks:

- This algorithm is asynchronous and uniform as well.
- The question may arise whether one can design an algorithm with an even lower message complexity. We answer this question in the next section.

3.3 Lower Bounds

Lower bounds in distributed computing are often easier than in the standard centralized (random access machine, RAM) model because one can argue about messages that need to be exchanged. In this section we present a first difficult lower bound. We show that Algorithm 3.8 is asymptotically optimal.

Definition 3.10 (Execution)**.** *An execution of a distributed algorithm is a list of events, sorted by time. An event is a record (time, node, type, message), where type is "send" or "receive".*

Remarks:

- We assume throughout this course that no two events happen at exactly the same time (or one can break ties arbitrarily).
- An execution of an asynchronous algorithm is generally not only determined by the algorithm but also by a "god-like" scheduler. If more than one message is in transit, the scheduler can choose which one arrives first.
- If two messages are transmitted over the same directed edge, then it is sometimes required that the message first transmitted will also be received first ("FIFO").

For our lower bound, we assume the following model:

- We are given an asynchronous ring, where nodes may wake up at arbitrary times (but at the latest when receiving the first message).
- We only accept uniform algorithms where the node with the maximum identifier can be the leader. Additionally, every node that is not the leader must know the identity of the leader. These two requirements can be dropped when using a more complicated proof; however, this is beyond the scope of this course.
- During the proof we will "play god" and specify which message in transmission arrives next in the execution. We respect the FIFO conditions for links.

Definition 3.11 (Open Schedule)**.** *A schedule is an execution chosen by the scheduler. An open (undirected) edge is an edge where no message traversing the edge has been received so far. A schedule for a ring is open if there is an open edge in the ring.*

The proof of the lower bound is by induction. First we show the base case:

Lemma 3.12. *Given a ring R with two nodes, we can construct an open schedule in which at least one message is received. The nodes cannot distinguish this schedule from one on a larger ring with all other nodes being where the open edge is.*

Proof: Let the two nodes be u and v with $u < v$. Node u must learn the identity of node v, thus receive at least one message. We stop the execution of the algorithm as soon as the first message is received. (If the first message is received by v, bad luck for the algorithm!) Then the other edge in the ring (on which the received message was not transmitted) is open. Since the algorithm needs to be uniform, maybe the open edge is not really an edge at all, nobody can tell. We could use this to glue two rings together, by breaking up this imaginary open edge and connect two rings by two edges. An example can be seen in Figure 3.13.

Lemma 3.14. *By gluing together two rings of size $n/2$ for which we have open schedules, we can construct an open schedule on a ring of size n. If $M(n/2)$ denotes the number of messages already received in each of these schedules, at least $2M(n/2) + n/4$ messages have to be exchanged in order to solve leader election.*

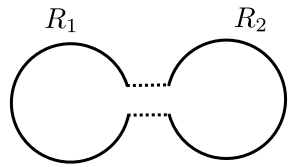


Figure 3.13: The rings R_1, R_2 are glued together at their open edge.

We divide the ring into two sub-rings R_1 and R_2 of size $n/2$. These subrings cannot be distinguished from rings with $n/2$ nodes if no messages are received from "outsiders". We can ensure this by not scheduling such messages until we want to. Note that executing both given open schedules on R_1 and R_2 "in parallel" is possible because we control not only the scheduling of the messages, but also when nodes wake up. By doing so, we make sure that $2M(n/2)$ messages are sent before the nodes in R_1 and R_2 learn anything of each other!

Without loss of generality, R_1 contains the maximum identifier. Hence, each node in R_2 must learn the identity of the maximum identifier, thus at least $n/2$ additional messages must be received. The only problem is that we cannot connect the two sub-rings with both edges since the new ring needs to remain open. Thus, only messages over one of the edges can be received. We look into the future: we check what happens when we close only one of these connecting edges.

Since we know that $n/2$ nodes have to be informed in R_2, there must be at least $n/2$ messages that must be received. Closing both edges must inform $n/2$ nodes, thus for one of the two edges there must be a node in distance $n/4$ which will be informed upon creating that edge. This results in $n/4$ additional messages. Thus, we pick this edge and leave the other one open which yields the claim.

Lemma 3.15. *Any uniform leader election algorithm for asynchronous rings has at least message complexity $M(n) \geq \frac{n}{4}(\log n + 1)$.*

Proof by induction: For the sake of simplicity we assume n being a power of 2. The base case $n = 2$ works because of Lemma 3.12 which implies that $M(2) \geq 1 = \frac{2}{4}(\log 2 + 1)$. For the induction step, using Lemma 3.14 and the induction hypothesis we have

$$\begin{aligned} M(n) &= 2 \cdot M\left(\frac{n}{2}\right) + \frac{n}{4} \\ &\geq 2 \cdot \left(\frac{n}{8}\left(\log \frac{n}{2} + 1\right)\right) + \frac{n}{4} \\ &= \frac{n}{4}\log n + \frac{n}{4} = \frac{n}{4}\left(\log n + 1\right). \end{aligned}$$

□

Remarks:

- To hide the ugly constants we use the "big Omega" notation, the lower bound equivalent of $\mathcal{O}()$. A function f is in $\Omega(g)$ if there are constants x_0 and $c > 0$ such that $|f(x)| \geq c|g(x)|$ for all $x \geq x_0$.

- In addition to the already presented parts of the "big O" notation, there are 3 additional ones. Remember that a function f is in $\mathcal{O}(g)$ if f grows at most as fast as g. A function f is in $o(g)$ if f grows slower than g.
- An analogous small letter notation exists for Ω. A function f is in $\omega(g)$ if f grows faster than g.
- Last but not least, we say that a function f is in $\Theta(g)$ if f grows as fast as g, i.e., $f \in \mathcal{O}(g)$ and $f \in \Omega(g)$.
- Again, we refer to standard text books for formal definitions.

Theorem 3.16 (Asynchronous Leader Election Lower Bound)**.** *Any uniform leader election algorithm for asynchronous rings has $\Omega(n \log n)$ message complexity.*

3.4 Synchronous Ring

The lower bound relied on delaying messages for a very long time. Since this is impossible in the synchronous model, we might get a better message complexity in this case. The basic idea is very simple: In the synchronous model, *not* receiving a message is information as well! First we make some additional assumptions:

- We assume that the algorithm is non-uniform (i.e., the ring size n is known).
- We assume that every node starts at the same time.
- The node with the minimum identifier becomes the leader; identifiers are integers.

Algorithm 3.17 Synchronous Leader Election

```
1: Each node v concurrently executes the following code:
2: The algorithm operates in synchronous phases. Each phase consists of n
   time steps. Node v counts phases, starting with 0.
3: if phase = v and v did not yet receive a message then
4:     v decides to be the leader
5:     v sends the message "v is leader" around the ring
6: end if
```

Remarks:

- Message complexity is indeed n.
- But the time complexity is huge! If m is the minimum identifier it is $m \cdot n$.
- The synchronous start and the non-uniformity assumptions can be dropped by using a wake-up technique (upon receiving a wake-up message, wake up your clockwise neighbors) and by letting messages travel slowly.

- There are several lower bounds for the synchronous model: comparison-based algorithms or algorithms where the time complexity cannot be a function of the identifiers have message complexity $\Omega(n \log n)$ as well.
- In general graphs, efficient leader election may be tricky. While time-optimal leader election can be done by parallel flooding-echo (see Chapter 2), bounding the message complexity is more difficult.

Chapter Notes

[Ang80] was the first to mention the now well-known impossibility result for anonymous rings and other networks, even when using randomization. The first algorithm for asynchronous rings was presented in [Lan77], which was improved to the presented clockwise algorithm in [CR79]. Later, [HS80] found the radius growth algorithm, which decreased the worst case message complexity. Algorithms for the unidirectional case with runtime $\mathcal{O}(n \log n)$ can be found in [DKR82, Pet82]. The $\Omega(n \log n)$ message complexity lower bound for comparison based algorithms was first published in [FL87]. In [Sch89] an algorithm with constant error probability for anonymous networks is presented. General results about limitations of computer power in synchronous rings are in [ASW88, AS88].

Bibliography

[Ang80] Dana Angluin. Local and global properties in networks of processors (Extended Abstract). In *12th ACM Symposium on Theory of Computing (STOC)*, 1980.

[AS88] Hagit Attiya and Marc Snir. Better Computing on the Anonymous Ring. In *Aegean Workshop on Computing (AWOC)*, 1988.

[ASW88] Hagit Attiya, Marc Snir, and Manfred K. Warmuth. Computing on an anonymous ring. volume 35, pages 845–875, 1988.

[CR79] Ernest Chang and Rosemary Roberts. An improved algorithm for decentralized extrema-finding in circular configurations of processes. *Commun. ACM*, 22(5):281–283, May 1979.

[DKR82] Danny Dolev, Maria M. Klawe, and Michael Rodeh. An $O(n \log n)$ Unidirectional Distributed Algorithm for Extrema Finding in a Circle. *J. Algorithms*, 3(3):245–260, 1982.

[FL87] Greg N. Frederickson and Nancy A. Lynch. Electing a leader in a synchronous ring. *J. ACM*, 34(1):98–115, 1987.

[HS80] D. S. Hirschberg and J. B. Sinclair. Decentralized extrema-finding in circular configurations of processors. *Commun. ACM*, 23(11):627–628, November 1980.

[Lan77] Gérard Le Lann. Distributed Systems - Towards a Formal Approach. In *International Federation for Information Processing (IFIP) Congress*, 1977.

[Pet82] Gary L. Peterson. An $O(n \log n)$ Unidirectional Algorithm for the Circular Extrema Problem. 4(4):758–762, 1982.

[Sch89] B. Schieber. Calling names on nameless networks. In *Proceedings of the eighth annual ACM Symposium on Principles of distributed computing*, PODC '89, pages 319–328, New York, NY, USA, 1989. ACM.

Chapter 4

Distributed Sorting

"Indeed, I believe that virtually *every* important aspect of programming arises somewhere in the context of sorting [and searching]!"

– Donald E. Knuth, The Art of Computer Programming

In this chapter we study a classic problem in computer science—sorting—from a distributed computing perspective. In contrast to an orthodox single-processor sorting algorithm, no node has access to all data, instead the to-be-sorted values are *distributed*. Distributed sorting then boils down to:

Definition 4.1 (Sorting)**.** *We choose a graph with n nodes $v_1, \ldots, v_n$. Initially each node stores a value. After applying a sorting algorithm, node v_k stores the k^{th} smallest value.*

Remarks:

- What if we route all values to the same central node v, let v sort the values locally, and then route them to the correct destinations?! According to the message passing model studied in the first few chapters this is perfectly legal. With a star topology sorting finishes in $\mathcal{O}(1)$ time!

Definition 4.2 (Node Contention)**.** *In each step of a synchronous algorithm, each node can only send and receive $\mathcal{O}(1)$ messages containing $\mathcal{O}(1)$ values, no matter how many neighbors the node has.*

Remarks:

- Using Definition 4.2 sorting on a star graph takes linear time.

4.1 Array & Mesh

To get a better intuitive understanding of distributed sorting, we start with two simple topologies, the array and the mesh. Let us begin with the array:

Algorithm 4.3 Odd/Even Sort

1: Given an array of n nodes $(v_1, \ldots, v_n)$, each storing a value (not sorted).
2: **repeat**
3: Compare and exchange the values at nodes i and $i+1$, i odd
4: Compare and exchange the values at nodes i and $i+1$, i even
5: **until** done

Remarks:

- The compare and exchange primitive in Algorithm 4.3 is defined as follows: Let the value stored at node i be v_i. After the compare and exchange node i stores value $\min(v_i, v_{i+1})$ and node $i+1$ stores value $\max(v_i, v_{i+1})$.
- How fast is the algorithm, and how can we prove correctness/efficiency?
- The most interesting proof uses the so-called 0-1 Sorting Lemma. It allows us to restrict our attention to an input of 0's and 1's only, and works for any "oblivious comparison-exchange" algorithm. (Oblivious means: Whether you exchange two values must only depend on the relative order of the two values, and not on anything else.)

Lemma 4.4 (0-1 Sorting Lemma)**.** *If an oblivious comparison-exchange algorithm sorts all inputs of 0's and 1's, then it sorts arbitrary inputs.*

Proof. We prove the opposite direction (does not sort arbitrary inputs $\Rightarrow$ does not sort 0's and 1's). Assume that there is an input $x = x_1, \ldots, x_n$ that is not sorted correctly. Then there is a smallest value k such that the value at node v_k after running the sorting algorithm is strictly larger than the k^{th} smallest value $x(k)$. Define an input $x_i^* = 0 \Leftrightarrow x_i \leq x(k)$, $x_i^* = 1$ else. Whenever the algorithm compares a pair of 1's or 0's, it is not important whether it exchanges the values or not, so we may simply assume that it does the same as on the input x. On the other hand, whenever the algorithm exchanges some values $x_i^* = 0$ and $x_j^* = 1$, this means that $x_i \leq x(k) < x_j$. Therefore, in this case the respective compare-exchange operation will do the same on both inputs. We conclude that the algorithm will order x^* the same way as x, i.e., the output with only 0's and 1's will also not be correct. □

Theorem 4.5. *Algorithm 4.3 sorts correctly in n steps.*

Proof. Thanks to Lemma 4.4 we only need to consider an array with 0's and 1's. Let j_1 be the node with the rightmost (highest index) 1. If j_1 is odd (even) it will move in the first (second) step. In any case it will move right in every following step until it reaches the rightmost node v_n. Let j_k be the node with the k^{th} rightmost 1. We show by induction that j_k is not "blocked" anymore (constantly moves until it reaches destination!) after step k. We have already anchored the induction at $k = 1$. Since j_{k-1} moves after step $k-1$, j_k gets a right 0-neighbor for each step after step k. (For matters of presentation we omitted a couple of simple details.) □

Remarks:

- Linear time is not very exciting, maybe we can do better by using a different topology? Let's try a mesh (a.k.a. grid) topology first.

Algorithm 4.6 Shearsort

1: We are given a mesh with m rows and m columns, m even, $n = m^2$.
2: The sorting algorithm operates in phases, and uses the odd/even sort algorithm on rows or columns.
3: **repeat**
4: In the odd phases $1, 3, \ldots$ we sort all the rows, in the even phases $2, 4, \ldots$ we sort all the columns, such that:
5: Columns are sorted such that the small values move up.
6: Odd rows $(1, 3, \ldots, m-1)$ are sorted such that small values move left.
7: Even rows $(2, 4, \ldots, m)$ are sorted such that small values move right.
8: **until** done

Theorem 4.7. *Algorithm 4.6 sorts n values in $\sqrt{n}(\log n + 1)$ time in snake-like order.*

Proof. Since the algorithm is oblivious, we can use Lemma 4.4. We show that after a row and a column phase, half of the previously unsorted rows will be sorted. More formally, let us call a row with only 0's (or only 1's) *clean*, a row with 0's *and* 1's is *dirty*. At any stage, the rows of the mesh can be divided into three regions. In the north we have a region of all-0 rows, in the south all-1 rows, in the middle a region of dirty rows. Initially all rows can be dirty. Since neither row nor column sort will touch already clean rows, we can concentrate on the dirty rows.

First we run an odd phase. Then, in the even phase, we run a peculiar column sorter: We group two consecutive dirty rows into pairs. Since odd and even rows are sorted in opposite directions, two consecutive dirty rows look as follows:

00000 ... 11111

11111 ... 00000

Such a pair can be in one of three states. Either we have more 0's than 1's, or more 1's than 0's, or an equal number of 0's and 1's. Column-sorting each pair will give us at least one clean row (and two clean rows if "$|0| = |1|$"). Then move the cleaned rows north/south and we will be left with half the dirty rows.

At first glance it appears that we need such a peculiar column sorter. However, any column sorter sorts the columns in exactly the same way (we are very grateful to have Lemma 4.4!).

All in all we need $2 \log m = \log n$ phases to remain only with 1 dirty row in the middle which will be sorted (not cleaned) with the last row-sort. □

Remarks:

- There are algorithms that sort in $3m + o(m)$ time on an m by m mesh (by diving the mesh into smaller blocks). This is asymptotically optimal, since a value might need to move $2m$ times.
- Such a $\sqrt{n}$-sorter is cute, but we are more ambitious. There are non-distributed sorting algorithms such as quicksort, heapsort, or mergesort that sort n values in (expected) $\mathcal{O}(n \log n)$ time. Using our n-fold parallelism effectively we might therefore hope for a distributed sorting algorithm that sorts in time $\mathcal{O}(\log n)$!

4.2 Sorting Networks

In this section we construct a graph topology which is carefully manufactured for sorting. This is a deviation from previous chapters where we always had to work with the topology that was given to us. In many application areas (e.g. peer-to-peer networks, communication switches, systolic hardware) it is indeed possible (in fact, crucial!) that an engineer can build the topology best suited for her application.

Definition 4.8 (Sorting Networks)**.** *A* comparator *is a device with two inputs x, y and two outputs x', y' such that $x' = min(x, y)$ and $y' = max(x, y)$. We construct so-called* comparison networks *that consist of wires that connect comparators (the output port of a comparator is sent to an input port of another comparator). Some wires are not connected to comparator outputs, and some are not connected to comparator inputs. The first are called input wires of the comparison network, the second output wires. Given n values on the input wires, a* sorting network *ensures that the values are sorted on the output wires. We will also use the term* width *to indicate the number of wires in the sorting network.*

Remarks:

- The odd/even sorter explained in Algorithm 4.3 can be described as a sorting network.
- Often we will draw all the wires on n horizontal lines (n being the "width" of the network). Comparators are then vertically connecting two of these lines.
- Note that a sorting network is an oblivious comparison-exchange network. Consequently we can apply Lemma 4.4 throughout this section. An example sorting network is depicted in Figure 4.9.

Definition 4.10 (Depth)**.** *The depth of an input wire is* 0*. The depth of a comparator is the maximum depth of its input wires plus one. The depth of an output wire of a comparator is the depth of the comparator. The depth of a comparison network is the maximum depth (of an output wire).*

Definition 4.11 (Bitonic Sequence)**.** *A* bitonic sequence *is a sequence of numbers that first monotonically increases, and then monotonically decreases, or vice versa.*

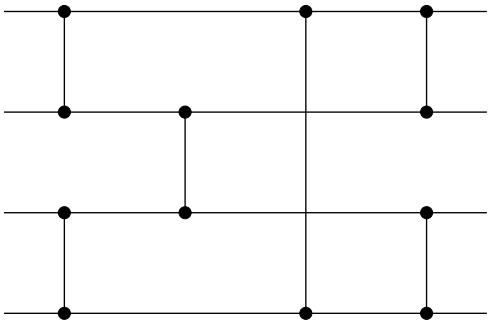


Figure 4.9: A sorting network.

Remarks:

- $< 1, 4, 6, 8, 3, 2 >$ or $< 5, 3, 2, 1, 4, 8 >$ are bitonic sequences.
- $< 9, 6, 2, 3, 5, 4 >$ or $< 7, 4, 2, 5, 9, 8 >$ are not bitonic.
- Since we restrict ourselves to 0's and 1's (Lemma 4.4), bitonic sequences have the form $0^i1^j0^k$ or $1^i0^j1^k$ for $i, j, k \geq 0$.

Algorithm 4.12 Half Cleaner

1: A half cleaner is a comparison network of depth 1, where we compare wire i with wire $i + n/2$ for $i = 1, \ldots, n/2$ (we assume n to be even).

Lemma 4.13. *Feeding a bitonic sequence into a half cleaner (Algorithm 4.12), the half cleaner cleans (makes all-*0 *or all-*1*) either the upper or the lower half of the n wires. The other half is bitonic.*

Proof. Assume that the input is of the form $0^i1^j0^k$ for $i, j, k \geq 0$. If the midpoint falls into the 0's, the input is already clean/bitonic and will stay so. If the midpoint falls into the 1's the half cleaner acts as Shearsort with two adjacent rows, exactly as in the proof of Theorem 4.7. The case $1^i0^j1^k$ is symmetric. □

Algorithm 4.14 Bitonic Sequence Sorter

1: A bitonic sequence sorter of width n (n being a power of 2) consists of a half cleaner of width n, and then two bitonic sequence sorters of width $n/2$ each.
2: A bitonic sequence sorter of width 1 is empty.

Lemma 4.15. *A bitonic sequence sorter (Algorithm 4.14) of width n sorts bitonic sequences. It has depth* $\log n$.

Proof. The proof follows directly from the Algorithm 4.14 and Lemma 4.13. □

Remarks:

- Clearly we want to sort arbitrary and not only bitonic sequences! To do this we need one more concept, merging networks.

Algorithm 4.16 Merging Network

1: A merging network of width n is a merger of width n followed by two bitonic sequence sorters of width $n/2$. A merger is a depth-one network where we compare wire i with wire $n - i + 1$, for $i = 1, \ldots, n/2$.

Remarks:

- Note that a merging network is a bitonic sequence sorter where we replace the (first) half-cleaner by a merger.

Lemma 4.17. *A merging network of width n (Algorithm 4.16) merges two sorted input sequences of length* $n/2$ *each into one sorted sequence of length n.*

Proof. We have two sorted input sequences. Essentially, a merger does to two sorted sequences what a half cleaner does to a bitonic sequence, since the lower part of the input is reversed. In other words, we can use the same argument as in Theorem 4.7 and Lemma 4.13: Again, after the merger step either the upper or the lower half is clean, the other is bitonic. The bitonic sequence sorters complete sorting. □

Remarks:

- How do you sort n values when you are able to merge two sorted sequences of size $n/2$? Piece of cake, just apply the merger recursively.

Algorithm 4.18 Batcher's "Bitonic" Sorting Network

1: A batcher sorting network of width n consists of two batcher sorting networks of width $n/2$ followed by a merging network of width n. (See Figure 4.19.)
2: A batcher sorting network of width 1 is empty.

Theorem 4.20. *A sorting network (Algorithm 4.18) sorts an arbitrary sequence of n values. It has depth* $\mathcal{O}(\log^2 n)$.

Proof. Correctness is immediate: at recursive stage k ($k = 1, 2, 3, \ldots, \log n$) we merge 2^k) sorted sequences into 2^{k-1} sorted sequences. The depth $d(n)$ of the sorting network of level n is the depth of a sorting network of level $n/2$ plus the depth $m(n)$ of a merging network with width n. The depth of a sorter of level 1 is 0 since the sorter is empty. Since a merging network of width n has the same depth as a bitonic sequence sorter of width n, we know by Lemma 4.15 that $m(n) = \log n$. This gives a recursive formula for $d(n)$ which solves to $d(n) = \frac{1}{2}\log^2 n + \frac{1}{2}\log n$. □

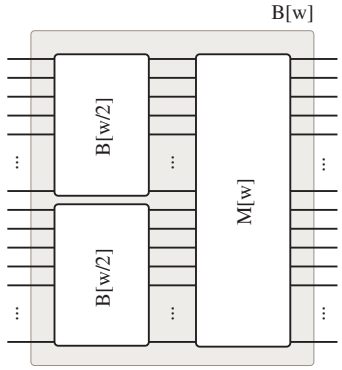


Figure 4.19: A batcher sorting network

Remarks:

- Simulating Batcher's sorting network on an ordinary sequential computer takes time $\mathcal{O}(n \log^2 n)$. As said, there are sequential sorting algorithms that sort in asymptotically optimal time $\mathcal{O}(n \log n)$. So a natural question is whether there is a sorting network with depth $\mathcal{O}(\log n)$. Such a network would have some remarkable advantages over sequential asymptotically optimal sorting algorithms such as heapsort. Apart from being highly parallel, it would be completely oblivious, and as such perfectly suited for a fast hardware solution. In 1983, Ajtai, Komlos, and Szemeredi presented a celebrated $\mathcal{O}(\log n)$ depth sorting network. (Unlike Batcher's sorting network the constant hidden in the big-O of the "AKS" sorting network is too large to be practical, however.)
- It can be shown that Batcher's sorting network and similarly others can be simulated by a Butterfly network and other hypercubic networks, see next chapter.
- What if a sorting network is asynchronous?!? Clearly, using a synchronizer we can still sort, but it is also possible to use it for something else. Check out the next section!

4.3 Counting Networks

In this section we address distributed counting, a distributed service which can for instance be used for load balancing.

Definition 4.21 (Distributed Counting)**.** *A* distributed counter *is a variable that is common to all processors in a system and that supports an atomic* test-and-increment *operation. The operation delivers the system's counter value to the requesting processor and increments it.*

Remarks:

- A naive distributed counter stores the system's counter value with a distinguished central node. When other nodes initiate the test-and-increment operation, they send a request message to the central node and in turn receive a reply message with the current counter value. However, with a large number of nodes operating on the distributed counter, the central processor will become a bottleneck. There will be a congestion of request messages at the central processor, in other words, the system will not scale.
- Is a scalable implementation (without any kind of bottleneck) of such a distributed counter possible, or is distributed counting a problem which is inherently centralized?!?
- Distributed counting could for instance be used to implement a load balancing infrastructure, i.e. by sending the job with counter value i (modulo n) to server i (out of n possible servers).

Definition 4.22 (Balancer)**.** *A balancer is an asynchronous flip-flop which forwards messages that arrive on the left side to the wires on the right, the first to the upper, the second to the lower, the third to the upper, and so on.*

Algorithm 4.23 Bitonic Counting Network.

1: Take Batcher's bitonic sorting network of width w and replace all the comparators with balancers.
2: When a node wants to count, it sends a message to an arbitrary input wire.
3: The message is then routed through the network, following the rules of the asynchronous balancers.
4: Each output wire is completed with a "mini-counter."
5: The mini-counter of wire k replies the value "$k + i \cdot w$" to the initiator of the i^{th} message it receives.

Definition 4.24 (Step Property)**.** *A sequence $y_0, y_1, \ldots, y_{w-1}$ is said to have the* step property*, if $0 \leq y_i - y_j \leq 1$, for any $i < j$.*

Remarks:

- If the output wires have the step property, then with r requests, exactly the values $1, \ldots, r$ will be assigned by the mini-counters. All we need to show is that the counting network has the step property. For that we need some additional facts...

Facts 4.25. *For a balancer, we denote the number of consumed messages on the i^{th} input wire with x_i, $i = 0, 1$. Similarly, we denote the number of sent messages on the i^{th} output wire with y_i, $i = 0, 1$. A balancer has these properties:*

(1) A balancer does not generate output-messages; that is, $x_0 + x_1 \geq y_0 + y_1$ in any state.

(2) Every incoming message is eventually forwarded. In other words, if we are in a quiescent state (no message in transit), then $x_0 + x_1 = y_0 + y_1$.

(3) The number of messages sent to the upper output wire is at most one higher than the number of messages sent to the lower output wire: in any state $y_0 = \lceil (y_0 + y_1)/2 \rceil$ (thus $y_1 = \lfloor (y_0 + y_1)/2 \rfloor$).

Facts 4.26. *If a sequence $y_0, y_1, \ldots, y_{w-1}$ has the step property,*

(1) then all its subsequences have the step property.

(2) then its even and odd subsequences satisfy

$$\sum_{i=0}^{w/2-1} y_{2i} = \left\lceil \frac{1}{2} \sum_{i=0}^{w-1} y_i \right\rceil \text{ and } \sum_{i=0}^{w/2-1} y_{2i+1} = \left\lfloor \frac{1}{2} \sum_{i=0}^{w-1} y_i \right\rfloor.$$

Facts 4.27. *If two sequences $x_0, x_1, \ldots, x_{w-1}$ and $y_0, y_1, \ldots, y_{w-1}$ have the step property,*

(1) and $\sum_{i=0}^{w-1} x_i = \sum_{i=0}^{w-1} y_i$, then $x_i = y_i$ for $i = 0, \ldots, w-1$.

(2) and $\sum_{i=0}^{w-1} x_i = \sum_{i=0}^{w-1} y_i + 1$, then there exists a unique j ($j = 0, 1, \ldots, w-1$) such that $x_j = y_j + 1$, and $x_i = y_i$ for $i = 0, \ldots, w-1$, $i \neq j$.

Remarks:

- An alternative representation of Batcher's network has been introduced in [AHS94]. It is isomorphic to Batcher's network, and relies on a Merger Network $M[w]$ which is defined inductively: $M[w]$ consists of two $M[w/2]$ networks (an upper and a lower one) whose output is fed to $w/2$ balancers. The upper balancer merges the even subsequence $x_0, x_2, \ldots, x_{w-2}$, while the lower balancer merges the odd subsequence $x_1, x_3, \ldots, x_{w-1}$. Call the outputs of these two $M[w/2]$, z and z' respectively. The final stage of the network combines z and z' by sending each pair of wires z_i and z'_i into a balancer whose outputs yield y_{2i} and y_{2i+1}.
- It is enough to prove that a merger network $M[w]$ preserves the step property.

Lemma 4.28. *Let $M[w]$ be a merger network of width w. In a quiescent state (no message in transit), if the inputs $x_0, x_1, \ldots, x_{w/2-1}$ resp. $x_{w/2}, x_{w/2+1}, \ldots, x_{w-1}$ have the step property, then the output $y_0, y_1, \ldots, y_{w-1}$ has the step property.*

Proof. By induction on the width w.

For $w = 2$: $M[2]$ is a balancer and a balancer's output has the step property (Fact 4.25.3).

For $w > 2$: Let z resp. z' be the output of the upper respectively lower $M[w/2]$ subnetwork. Since $x_0, x_1, \ldots, x_{w/2-1}$ and $x_{w/2}, x_{w/2+1}, \ldots, x_{w-1}$ both have the step property by assumption, their even and odd subsequences also have the step property (Fact 4.26.1). By induction hypothesis, the output of both $M[w/2]$ subnetworks have the step property. Let $Z := \sum_{i=0}^{w/2-1} z_i$ and $Z' := \sum_{i=0}^{w/2-1} z'_i$. From Fact 4.26.2 we conclude that $Z = \lceil \frac{1}{2} \sum_{i=0}^{w/2-1} x_i \rceil + \lfloor \frac{1}{2} \sum_{i=w/2}^{w-1} x_i \rfloor$ and $Z' = \lfloor \frac{1}{2} \sum_{i=0}^{w/2-1} x_i \rfloor + \lceil \frac{1}{2} \sum_{i=w/2}^{w-1} x_i \rceil$. Since $\lceil a \rceil + \lfloor b \rfloor$ and $\lfloor a \rfloor + \lceil b \rceil$ differ by at most 1 we know that Z and Z' differ by at most 1.

If $Z = Z'$, Fact 4.27.1 implies that $z_i = z'_i$ for $i = 0, \ldots, w/2-1$. Therefore, the output of $M[w]$ is $y_i = z_{\lfloor i/2 \rfloor}$ for $i = 0, \ldots, w-1$. Since $z_0, \ldots, z_{w/2-1}$ has the step property, so does the output of $M[w]$ and the lemma follows.

If Z and Z' differ by 1, Fact 4.27.2 implies that $z_i = z'_i$ for $i = 0, \ldots, w/2-1$, except a unique j such that z_j and z'_j differ by only 1, for $j = 0, \ldots, w/2-1$. Let $l := min(z_j, z'_j)$. Then, the output y_i (with $i < 2j$) is $l + 1$. The output y_i (with $i > 2j + 1$) is l. The output y_{2j} and y_{2j+1} are balanced by the final balancer resulting in $y_{2j} = l + 1$ and $y_{2j+1} = l$. Therefore $M[w]$ preserves the step property. □

A bitonic counting network is constructed to fulfill Lemma 4.28, i.e., the final output comes from a Merger whose upper and lower inputs are recursively merged. Therefore, the following theorem follows immediately.

Theorem 4.29 (Correctness)**.** *In a quiescent state, the w output wires of a bitonic counting network of width w have the step property.*

Remarks:

- Is every sorting network also a counting network? No. But surprisingly, the other direction is true!

Theorem 4.30 (Counting vs. Sorting)**.** *If a network is a counting network then it is also a sorting network, but not vice versa.*

Proof. There are sorting networks that are not counting networks (e.g. odd/even sort, or insertion sort). For the other direction, let C be a counting network and $I(C)$ be the isomorphic network, where every balancer is replaced by a comparator. Let $I(C)$ have an arbitrary input of 0's and 1's; that is, some of the input wires have a 0, all others have a 1. There is a message at C's i^{th} input wire if and only if $I(C)$'s i input wire is 0. Since C is a counting network, all messages are routed to the upper output wires. $I(C)$ is isomorphic to C, therefore a comparator in $I(C)$ will receive a 0 on its upper (lower) wire if and only if the corresponding balancer receives a message on its upper (lower) wire. Using an inductive argument, the 0's and 1's will be routed through $I(C)$ such that all 0's exit the network on the upper wires whereas all 1's exit the network on the lower wires. Applying Lemma 4.4 shows that $I(C)$ is a sorting network. □

Remarks:

- We claimed that the counting network is correct. However, it is only correct in a quiescent state.

Definition 4.31 (Linearizable)**.** *A system is* linearizable *if the order of the values assigned reflects the real-time order in which they were requested. More formally, if there is a pair of operations o_1, o_2, where operation o_1 terminates before operation o_2 starts, and the logical order is "o_2 before o_1", then a distributed system is not linearizable.*

Lemma 4.32 (Linearizability)**.** *The bitonic counting network is not linearizable.*

Proof. Consider the bitonic counting network with width 4 in Figure 4.33: Assume that two *inc* operations were initiated and the corresponding messages entered the network on wire 0 and 2 (both in light gray color). After having passed the second resp. the first balancer, these traversing messages "fall asleep"; In other words, both messages take unusually long time before they are received by the next balancer. Since we are in an asynchronous setting, this may be the case.

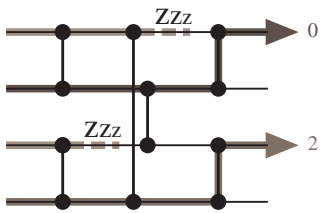


Figure 4.33: Linearizability Counter Example.

In the meantime, another *inc* operation (medium gray) is initiated and enters the network on the bottom wire. The message leaves the network on wire 2, and the *inc* operation is completed.

Strictly afterwards, another *inc* operation (dark gray) is initiated and enters the network on wire 1. After having passed all balancers, the message will leave the network wire 0. Finally (and not depicted in Figure 4.33), the two light gray messages reach the next balancer and will eventually leave the network on wires 1 resp. 3. Because the dark gray and the medium gray operation do conflict with Definition 4.31, the bitonic counting network is not linearizable. □

Remarks:

- Note that the example in Figure 4.33 behaves correctly in the quiescent state: Finally, exactly the values $0, 1, 2, 3$ are allotted.
- It was shown that linearizability comes at a high price (the depth grows linearly with the width).

Chapter Notes

The technique used for the famous lower bound of comparison-based sequential sorting first appeared in [FJ59]. Comprehensive introductions to the vast field of sorting can certainly be found in [Knu73]. Knuth also presents the 0/1 principle in the context of sorting networks, supposedly as a special case of a theorem for decision trees of W. G. Bouricius, and includes a historic overview of sorting network research.

Using a rather complicated proof not based on the 0/1 principle, [Hab72] first presented and analyzed Odd/Even sort on arrays. Shearsort for grids first appeared in [SSS86] as a sorting algorithm both easy to implement and to prove correct. Later it was generalized to meshes with higher dimension in [SS89]. A

bubble sort based algorithm is presented in [SI86]; it takes time $\mathcal{O}(\sqrt{n}\log n)$, but is fast in practice. Nevertheless, already [TK77] presented an asymptotically optimal algorithms for grid network which runs in $3n + O(n^{2/3}\log n)$ rounds for an $n \times n$ grid. A simpler algorithm was later found by [SS86] using $3n + O(n^{3/4})$ rounds.

Batcher presents his famous $\mathcal{O}(\log^2 n)$ depth sorting network in [Bat68]. It took until [AKS83] to find a sorting network with asymptotically optimal depth $\mathcal{O}(\log n)$. Unfortunately, the constants hidden in the big-O-notation render it rather impractical.

The notion of counting networks was introduced in [AHS91], and shortly afterward the notion of linearizability was studied by [HSW91]. Follow-up work in [AHS94] presents bitonic counting networks and studies contention in the counting network. An overview of research on counting networks can be found in [BH98].

Bibliography

[AHS91] James Aspnes, Maurice Herlihy, and Nir Shavit. Counting networks and multi-processor coordination. In *Proceedings of the twenty-third annual ACM symposium on Theory of computing*, STOC '91, pages 348–358, New York, NY, USA, 1991. ACM.

[AHS94] James Aspnes, Maurice Herlihy, and Nir Shavit. Counting networks. *J. ACM*, 41(5):1020–1048, September 1994.

[AKS83] Miklos Ajtai, Janos Komlós, and Endre Szemerédi. An 0(n log n) sorting network. In *Proceedings of the fifteenth annual ACM symposium on Theory of computing*, STOC '83, pages 1–9, New York, NY, USA, 1983. ACM.

[Bat68] Kenneth E. Batcher. Sorting networks and their applications. In *Proceedings of the April 30–May 2, 1968, spring joint computer conference*, AFIPS '68 (Spring), pages 307–314, New York, NY, USA, 1968. ACM.

[BH98] Costas Busch and Maurice Herlihy. A Survey on Counting Networks. In *WDAS*, pages 13–20, 1998.

[FJ59] Lester R. Ford and Selmer M. Johnson. A Tournament Problem. *The American Mathematical Monthly*, 66(5):pp. 387–389, 1959.

[Hab72] Nico Habermann. Parallel neighbor-sort (or the glory of the induction principle). Paper 2087, Carnegie Mellon University - Computer Science Departement, 1972.

[HSW91] M. Herlihy, N. Shavit, and O. Waarts. Low contention linearizable counting. In *Foundations of Computer Science, 1991. Proceedings., 32nd Annual Symposium on*, pages 526–535, oct 1991.

[Knu73] Donald E. Knuth. *The Art of Computer Programming, Volume III: Sorting and Searching*. Addison-Wesley, 1973.

[SI86] Kazuhiro Sado and Yoshihide Igarashi. Some parallel sorts on a mesh-connected processor array and their time efficiency. *Journal of Parallel and Distributed Computing*, 3(3):398–410, 1986.

[SS86] Claus Peter Schnorr and Adi Shamir. An optimal sorting algorithm for mesh connected computers. In *Proceedings of the eighteenth annual ACM symposium on Theory of computing*, STOC '86, pages 255–263, New York, NY, USA, 1986. ACM.

[SS89] Isaac D. Scherson and Sandeep Sen. Parallel sorting in two-dimensional VLSI models of computation. *Computers, IEEE Transactions on*, 38(2):238–249, feb 1989.

[SSS86] Isaac Scherson, Sandeep Sen, and Adi Shamir. Shear sort – A true two-dimensional sorting technique for VLSI networks. *1986 International Conference on Parallel Processing*, 1986.

[TK77] Clark David Thompson and Hsiang Tsung Kung. Sorting on a mesh-connected parallel computer. *Commun. ACM*, 20(4):263–271, April 1977.

Chapter 5

Shared Memory

In distributed computing, various different models exist. So far, the focus of the course was on loosely-coupled distributed systems such as the Internet, where nodes asynchronously communicate by exchanging messages. The "opposite" model is a tightly-coupled parallel computer where nodes access a common memory totally synchronously—in distributed computing such a system is called a Parallel Random Access Machine (PRAM).

5.1 Model

A third major model is somehow between these two extremes, the *shared memory* model. In a shared memory system, asynchronous processes (or processors) communicate via a common memory area of shared variables or registers:

Definition 5.1 (Shared Memory)**.** *A shared memory system is a system that consists of asynchronous processes that access a common (shared) memory. A process can atomically access a register in the shared memory through a set of predefined operations. An atomic modification appears to the rest of the system instantaneously. Apart from this shared memory, processes can also have some local (private) memory.*

Remarks:

- Various shared memory systems exist. A main difference is how they allow processes to access the shared memory. All systems can atomically read or write a shared register R. Most systems do allow for advanced *atomic* read-modify-write (RMW) operations, for example:
  - test-and-set(R): $t := R$; $R := 1$; return t
  - fetch-and-add(R, x): $t := R$; $R := R + x$; return t
  - compare-and-swap(R, x, y): if $R = x$ then $R := y$; return **true**; else return **false**; endif;
  - load-link(R)/store-conditional(R, x): Load-link returns the current value of the specified register R. A subsequent store-conditional to the same register will store a new value x (and return **true**) only if no updates have occurred to that register since the load-link. If any updates have occurred, the store-conditional is guaranteed to fail (and return **false**), even if the value read by the load-link has since been restored.
- The power of RMW operations can be measured with the so-called *consensus-number*: The consensus-number k of a RMW operation defines whether one can solve consensus for k processes. Test-and-set for instance has consensus-number 2 (one can solve consensus with 2 processes, but not 3), whereas the consensus-number of compare-and-swap is infinite. This insight had practical impact, as hardware designers stopped developing shared memory systems supporting weak RMW operations.
- Many of the results derived in the message passing model have an equivalent in the shared memory model. Consensus for instance is traditionally studied in the shared memory model.
- Whereas programming a message passing system is rather tricky (in particular if fault-tolerance has to be integrated), programming a shared memory system is generally considered easier, as programmers are given access to global variables directly and do not need to worry about exchanging messages correctly. Because of this, even distributed systems which physically communicate by exchanging messages can often be programmed through a shared memory middleware, making the programmer's life easier.
- We will most likely find the general spirit of shared memory systems in upcoming multi-core architectures. As for programming style, the multi-core community seems to favor an accelerated version of shared memory, *transactional memory*.
- From a message passing perspective, the shared memory model is like a bipartite graph: On one side you have the processes (the nodes) which pretty much behave like nodes in the message passing model (asynchronous, maybe failures). On the other side you have the shared registers, which just work perfectly (no failures, no delay).

5.2 Mutual Exclusion

A classic problem in shared memory systems is mutual exclusion. We are given a number of processes which occasionally need to access the same resource. The resource may be a shared variable, or a more general object such as a data structure or a shared printer. The catch is that only one process at the time is allowed to access the resource. More formally:

Definition 5.2 (Mutual Exclusion)**.** *We are given a number of processes, each executing the following code sections:*
<Entry> → <Critical Section> → <Exit> → <Remaining Code>
A mutual exclusion algorithm consists of code for entry and exit sections, such that the following holds

- *Mutual Exclusion: At all times* at most one *process is in the critical section.*
- *No deadlock: If some process manages to get to the entry section, later* some *(possibly different) process will get to the critical section.*

Sometimes we in addition ask for

- *No lockout: If some process manages to get to the entry section, later* the same *process will get to the critical section.*
- *Unobstructed exit: No process can get stuck in the exit section.*

Using RMW primitives one can build mutual exclusion algorithms quite easily. Algorithm 5.3 shows an example with the test-and-set primitive.

Algorithm 5.3 Mutual Exclusion: Test-and-Set

Input: Shared register $R := 0$
<**Entry**>
1: **repeat**
2: $r :=$ test-and-set(R)
3: **until** $r = 0$
<**Critical Section**>
4: ...
<**Exit**>
5: $R := 0$
<**Remainder Code**>
6: ...

Theorem 5.4. *Algorithm 5.3 solves the mutual exclusion problem as in Definition 5.2.*

Proof. Mutual exclusion follows directly from the test-and-set definition: Initially R is 0. Let p_i be the i^{th} process to successfully execute the test-and-set, where successfully means that the result of the test-and-set is 0. This happens at time t_i. At time t_i' process p_i resets the shared register R to 0. Between t_i and t_i' no other process can successfully test-and-set, hence no other process can enter the critical section concurrently.

Proving no deadlock works similar: One of the processes loitering in the entry section will successfully test-and-set as soon as the process in the critical section exited.

Since the exit section only consists of a single instruction (no potential infinite loops) we have unobstructed exit. □

Remarks:

- No lockout, on the other hand, is not given by this algorithm. Even with only two processes there are asynchronous executions where always the same process wins the test-and-set.
- Algorithm 5.3 can be adapted to guarantee fairness (no lockout), essentially by ordering the processes in the entry section in a queue.

- A natural question is whether one can achieve mutual exclusion with only reads and writes, that is without advanced RMW operations. The answer is yes!

Our read/write mutual exclusion algorithm is for two processes p_0 and p_1 only. In the remarks we discuss how it can be extended. The general idea is that process p_i has to mark its desire to enter the critical section in a "want" register W_i by setting $W_i := 1$. Only if the other process is not interested ($W_{1-i} = 0$) access is granted. This however is too simple since we may run into a deadlock. This deadlock (and at the same time also lockout) is resolved by adding a priority variable Π. See Algorithm 5.5.

Algorithm 5.5 Mutual Exclusion: Peterson's Algorithm

Initialization: Shared registers W_0, W_1, Π, all initially 0.
Code for process p_i , $i = \{0, 1\}$
<**Entry**>
1: $W_i := 1$
2: $\Pi := 1 - i$
3: **repeat until** $\Pi = i$ or $W_{1-i} = 0$
<**Critical Section**>
4: ...
<**Exit**>
5: $W_i := 0$
<**Remainder Code**>
6: ...

Remarks:

- Note that line 3 in Algorithm 5.5 represents a "spinlock" or "busy-wait", similarly to the lines 1-3 in Algorithm 5.3.

Theorem 5.6. *Algorithm 5.5 solves the mutual exclusion problem as in Definition 5.2.*

Proof. The shared variable Π elegantly grants priority to the process that passes line 2 first. If both processes are competing, only process p_Π can access the critical section because of Π. The other process $p_{1-\Pi}$ cannot access the critical section because $W_\Pi = 1$ (and $\Pi \neq 1 - \Pi$). The only other reason to access the critical section is because the other process is in the remainder code (that is, not interested). This proves mutual exclusion!

No deadlock comes directly with Π: Process p_Π gets direct access to the critical section, no matter what the other process does.

Since the exit section only consists of a single instruction (no potential infinite loops) we have unobstructed exit.

Thanks to the shared variable Π also no lockout (fairness) is achieved: If a process p_i loses against its competitor p_{1-i} in line 2, it will have to wait until the competitor resets $W_{1-i} := 0$ in the exit section. If process p_i is unlucky it will not check $W_{1-i} = 0$ early enough before process p_{1-i} sets $W_{1-i} := 1$ again in line 1. However, as soon as p_{1-i} hits line 2, process p_i gets the priority due to Π, and can enter the critical section. □

Remarks:

- Extending Peterson's Algorithm to more than 2 processes can be done by a tournament tree, like in tennis. With n processes every process needs to win $\log n$ matches before it can enter the critical section. More precisely, each process starts at the bottom level of a binary tree, and proceeds to the parent level if winning. Once winning the root of the tree it can enter the critical section. Thanks to the priority variables Π at each node of the binary tree, we inherit all the properties of Definition 5.2.

5.3 Store & Collect

5.3.1 Problem Definition

In this section, we will look at a second shared memory problem that has an elegant solution. Informally, the problem can be stated as follows. There are n processes $p_1, \ldots, p_n$. Every process p_i has a read/write register R_i in the shared memory where it can *store* some information that is destined for the other processes. Further, there is an operation by which a process can *collect* (i.e., read) the values of all the processes that stored some value in their register.

We say that an operation *op1* precedes an operation *op2* iff *op1* terminates before *op2* starts. An operation *op2* follows an operation *op1* iff *op1* precedes *op2*.

Definition 5.7 (Collect)**.** *There are two operations: A* STORE(val) *by process p_i sets val to be the latest value of its register R_i. A* COLLECT *operation returns a* view*, a partial function V from the set of processes to a set of values, where $V(p_i)$ is the latest value stored by p_i, for each process p_i. For a* COLLECT *operation* cop*, the following validity properties must hold for every process p_i:*

- *If $V(p_i) = \perp$, then no* STORE *operation by p_i precedes* cop.
- *If $V(p_i) = v \neq \perp$, then v is the value of a* STORE *operation* sop *of p_i that does not follow* cop*, and there is no* STORE *operation by p_i that follows* sop *and precedes* cop.

Hence, a COLLECT operation *cop* should not read from the future or miss a preceding STORE operation *sop*.

We assume that the read/write register R_i of every process p_i is initialized to $\perp$. We define the step complexity of an operation *op* to be the number of accesses to registers in the shared memory. There is a trivial solution to the *collect* problem as shown by Algorithm 5.8.

Remarks:

- Algorithm 5.8 clearly works. The step complexity of every STORE operation is 1, the step complexity of a COLLECT operation is n.
- At first sight, the step complexities of Algorithm 5.8 seem optimal. Because there are n processes, there clearly are cases in which a COLLECT operation needs to read all n registers. However, there are also scenarios in which the step complexity of the COLLECT operation seems very costly. Assume that there are only two processes p_i and p_j that have stored a value in their registers R_i and R_j. In this case, a COLLECT in principle only needs to read the registers R_i and R_j and can ignore all the other registers.
- Assume that up to a certain time t, $k \leq n$ processes have finished or started at least one operation. We call an operation *op* at time t *adaptive* to contention if the step complexity of *op* only depends on k and is independent of n.
- In the following, we will see how to implement adaptive versions of STORE and COLLECT.

Algorithm 5.8 Collect: Simple (Non-Adaptive) Solution

```
Operation STORE(val) (by process p_i) :
1: R_i := val
Operation COLLECT:
2: for i := 1 to n do
3:    V(p_i) := R_i                          // read register R_i
4: end for
```

5.3.2 Splitters

Algorithm 5.9 Splitter Code

```
Shared Registers: X : {⊥} ∪ {1,...,n}; Y : boolean
Initialization: X := ⊥; Y := false

Splitter access by process p_i:
1: X := i;
2: if Y then
3:    return right
4: else
5:    Y := true
6:    if X = i then
7:       return stop
8:    else
9:       return left
10:   end if
11: end if
```

To obtain adaptive collect algorithms, we need a synchronization primitive, called a *splitter*.

Definition 5.11 (Splitter)**.** *A splitter is a synchronization primitive with the following characteristic. A process entering a splitter exits with either* ***stop****,* ***left****, or* ***right****. If k processes enter a splitter, at most one process exits with* ***stop*** *and at most $k-1$ processes exit with* ***left*** *and* ***right****, respectively.*

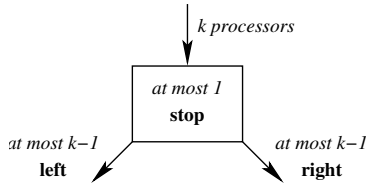


Figure 5.10: A Splitter

Hence, it is guaranteed that if a single process enters the splitter, then it obtains **stop**, and if two or more processes enter the splitter, then there is at most one process obtaining **stop** and there are two processes that obtain different values (i.e., either there is exactly one **stop** or there is at least one **left** and at least one **right**). For an illustration, see Figure 5.10. The code implementing a splitter is given by Algorithm 5.9.

Lemma 5.12. *Algorithm 5.9 correctly implements a splitter.*

Proof. Assume that k processes enter the splitter. Because the first process that checks whether $Y =$ **true** in line 2 will find that $Y =$ **false**, not all processes return **right**. Next, assume that i is the last process that sets $X := i$. If i does not return **right**, it will find $X = i$ in line 6 and therefore return **stop**. Hence, there is always a process that does not return **left**. It remains to show that at most 1 process returns **stop**. For the sake of contradiction, assume p_i and p_j are two processes that return **stop** and assume that p_i sets $X := i$ before p_j sets $X := j$. Both processes need to check whether Y is **true** before one of them sets $Y :=$ **true**. Hence, they both complete the assignment in line 1 before the first one of them checks the value of X in line 6. Hence, by the time p_i arrives at line 6, $X \neq i$ (p_j and maybe some other processes have overwritten X by then). Therefore, p_i does not return **stop** and we get a contradiction to the assumption that both p_i and p_j return **stop**. □

5.3.3 Binary Splitter Tree

Assume that we are given $2^n - 1$ splitters and that for every splitter S, there is an additional shared variable $Z_S : \{\perp\} \cup \{1, \ldots, n\}$ that is initialized to $\perp$ and an additional shared variable M_S : **boolean** that is initialized to **false**. We call a splitter S marked if $M_S =$ **true**. The $2^n - 1$ splitters are arranged in a complete binary tree of height $n - 1$. Let $S(v)$ be the splitter associated with a node v of the binary tree. The STORE and COLLECT operations are given by Algorithm 5.13.

Theorem 5.14. *Algorithm 5.13 correctly implements* STORE *and* COLLECT. *Let k be the number of participating processes. The step complexity of the first* STORE *of a process p_i is $\mathcal{O}(k)$, the step complexity of every additional* STORE *of p_i is $\mathcal{O}(1)$, and the step complexity of* COLLECT *is $\mathcal{O}(k)$.*

Proof. Because at most one process can stop at a splitter, it is sufficient to show that every process stops at some splitter at depth at most $k - 1 \leq n - 1$ when

Algorithm 5.13 Adaptive Collect: Binary Tree Algorithm

Operation STORE(val) (by process p_i) **:**
1: $R_i := val$
2: **if** first STORE operation by p_i **then**
3: $v :=$ root node of binary tree
4: $\alpha :=$ result of entering splitter $S(v)$;
5: $M_{S(v)} :=$ **true**
6: **while** $\alpha \neq$ **stop do**
7: **if** $\alpha =$ **left then**
8: $v :=$ left child of v
9: **else**
10: $v :=$ right child of v
11: **end if**
12: $\alpha :=$ result of entering splitter $S(v)$;
13: $M_{S(v)} :=$ **true**
14: **end while**
15: $Z_{S(v)} := i$
16: **end if**

Operation COLLECT**:**
Traverse marked part of binary tree:
17: **for all** marked splitters S **do**
18: **if** $Z_S \neq \perp$ **then**
19: $i := Z_S$; $V(p_i) := R_i$ // *read value of process p_i*
20: **end if**
21: **end for** // *$V(p_i) = \perp$ for all other processes*

invoking the first STORE operation to prove correctness. We prove that at most $k - i$ processes enter a subtree at depth i (i.e., a subtree where the root has distance i to the root of the whole tree). By definition of k, the number of processes entering the splitter at depth 0 (i.e., at the root of the binary tree) is k. For $i > 1$, the claim follows by induction because of the at most $k - i$ processes entering the splitter at the root of a depth i subtree, at most $k - i - 1$ obtain **left** and **right**, respectively. Hence, at the latest when reaching depth $k - 1$, a process is the only process entering a splitter and thus obtains **stop**. It thus also follows that the step complexity of the first invocation of STORE is $\mathcal{O}(k)$.

To show that the step complexity of COLLECT is $\mathcal{O}(k)$, we first observe that the marked nodes of the binary tree are connected, and therefore can be traversed by only reading the variables M_S associated to them and their neighbors. Hence, showing that at most $2k - 1$ nodes of the binary tree are marked is sufficient. Let x_k be the maximum number of marked nodes in a tree, where k processes access the root. We claim that $x_k \leq 2k - 1$, which is true for $k = 1$ because a single process entering a splitter will always compute **stop**. Now assume the inequality holds for $1, \ldots, k - 1$. Not all k processes may exit the splitter with **left** (or **right**), i.e., $k_l \leq k - 1$ processes will turn left and $k_r \leq \min\{k - k_l, k - 1\}$ turn right. The left and right children of the root are

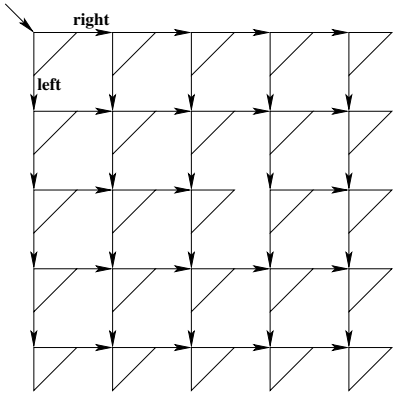


Figure 5.15: 5×5 Splitter Matrix

the roots of their subtrees, hence the induction hypothesis yields

$$x_k = x_{k_l} + x_{k_r} + 1 \leq (2k_l - 1) + (2k_r - 1) + 1 \leq 2k - 1,$$

concluding induction and proof. □

Remarks:

- The step complexities of Algorithm 5.13 are very good. Clearly, the step complexity of the COLLECT operation is asymptotically optimal. In order for the algorithm to work, we however need to allocate the memory for the complete binary tree of depth $n - 1$. The space complexity of Algorithm 5.13 therefore is exponential in n. We will next see how to obtain a polynomial space complexity at the cost of a worse COLLECT step complexity.

5.3.4 Splitter Matrix

Instead of arranging splitters in a binary tree, we arrange n^2 splitters in an $n \times n$ matrix as shown in Figure 5.15. The algorithm is analogous to Algorithm 5.13. The matrix is entered at the top left. If a process receives **left**, it next visits the splitter in the next row of the same column. If a process receives **right**, it next visits the splitter in the next column of the same row. Clearly, the space complexity of this algorithm is $\mathcal{O}(n^2)$. The following theorem gives bounds on the step complexities of STORE and COLLECT.

Theorem 5.16. *Let k be the number of participating processes. The step complexity of the first* STORE *of a process p_i is $\mathcal{O}(k)$, the step complexity of every additional* STORE *of p_i is $\mathcal{O}(1)$, and the step complexity of* COLLECT *is $\mathcal{O}(k^2)$.*

Proof. Let the top row be row 0 and the left-most column be column 0. Let x_i be the number of processes entering a splitter in row i. By induction on i, we show that $x_i \leq k - i$. Clearly, $x_0 \leq k$. Let us therefore consider the case $i > 0$. Let j be the largest column such that at least one process visits the splitter in row $i-1$ and column j. By the properties of splitters, not all processes entering the splitter in row $i - 1$ and column j obtain **left**. Therefore, not all processes entering a splitter in row $i - 1$ move on to row i. Because at least one process stays in every row, we get that $x_i \leq k - i$. Similarly, the number of processes entering column j is at most $k - j$. Hence, every process stops at the latest in row $k - 1$ and column $k - 1$ and the number of marked splitters is at most k^2. Thus, the step complexity of COLLECT is at most $\mathcal{O}(k^2)$. Because the longest path in the splitter matrix is $2k$, the step complexity of STORE is $\mathcal{O}(k)$. □

Remarks:

- With a slightly more complicated argument, it is possible to show that the number of processes entering the splitter in row i and column j is at most $k - i - j$. Hence, it suffices to only allocate the upper left half (including the diagonal) of the $n \times n$ matrix of splitters.

- The binary tree algorithm can be made space efficient by using a randomized version of a splitter. Whenever returning left or right, a randomized splitter returns left or right with probability $1/2$. With high probability, it then suffices to allocate a binary tree of depth $\mathcal{O}(\log n)$.

- Recently, it has been shown that with a considerably more complicated deterministic algorithm, it is possible to achieve $\mathcal{O}(k)$ step complexity and $\mathcal{O}(n^2)$ space complexity.

Chapter Notes

Already in 1965 Edsger Dijkstra gave a deadlock-free solution for mutual exclusion [Dij65]. Later, Maurice Herlihy suggested consensus-numbers [Her91], where he proved the "universality of consensus", i.e., the power of a shared memory system is determined by the consensus number. For this work, Maurice Herlihy was awarded the Dijkstra Prize in Distributed Computing in 2003. In 2016, Ellen et al. [EGSZ16] showed that some of the practical intuition about Herlihy's consensus number is misleading, as sets of instructions with a low consensus number can together achieve a high consensus number. In other words, in the world of instructions "the whole is greater than the sum of its parts". Petersons Algorithm is due to [PF77, Pet81], and adaptive collect was studied in the sequence of papers [MA95, AFG02, AL05, AKP+06].

Bibliography

[AFG02] Hagit Attiya, Arie Fouren, and Eli Gafni. An adaptive collect algorithm with applications. *Distributed Computing*, 15(2):87–96, 2002.

[AKP+06] Hagit Attiya, Fabian Kuhn, C. Greg Plaxton, Mirjam Wattenhofer, and Roger Wattenhofer. Efficient adaptive collect using randomization. *Distributed Computing*, 18(3):179–188, 2006.

[AL05] Yehuda Afek and Yaron De Levie. Space and Step Complexity Efficient Adaptive Collect. In *DISC*, pages 384–398, 2005.

[Dij65] Edsger W. Dijkstra. Solution of a problem in concurrent programming control. *Commun. ACM*, 8(9):569, 1965.

[EGSZ16] Faith Ellen, Rati Gelashvili, Nir Shavit, and Leqi Zhu. A complexity-based hierarchy for multiprocessor synchronization:[extended abstract]. In *Proceedings of the 2016 ACM Symposium on Principles of Distributed Computing*, pages 289–298. ACM, 2016.

[Her91] Maurice Herlihy. Wait-Free Synchronization. *ACM Trans. Program. Lang. Syst.*, 13(1):124–149, 1991.

[MA95] Mark Moir and James H. Anderson. Wait-Free Algorithms for Fast, Long-Lived Renaming. *Sci. Comput. Program.*, 25(1):1–39, 1995.

[Pet81] J.L. Peterson. Myths About the Mutual Exclusion Problem. *Information Processing Letters*, 12(3):115–116, 1981.

[PF77] G.L. Peterson and M.J. Fischer. Economical solutions for the critical section problem in a distributed system. In *Proceedings of the ninth annual ACM symposium on Theory of computing*, pages 91–97. ACM, 1977.

Chapter 6

Shared Objects

Assume that there is a common resource (e.g. a common variable or data structure), which different nodes in a network need to access from time to time. If the nodes are allowed to change the common object when accessing it, we need to guarantee that no two nodes have access to the object at the same time. In order to achieve this mutual exclusion, we need protocols that allow the nodes of a network to store and manage access to such a shared object.

6.1 Centralized Solutions

A simple and obvious solution is to store the shared object at a central location (see Algorithm 6.1).

Algorithm 6.1 Shared Object: Centralized Solution

Initialization: Shared object stored at root node r of a spanning tree of the network graph (i.e., each node knows its parent in the spanning tree).

Accessing Object: (by node v)

1: v sends request up the tree
2: request processed by root r (atomically)
3: result sent down the tree to node v

Remarks:

- Instead of a spanning tree, one can use routing.
- Algorithm 6.1 works, but it is not very efficient. Assume that the object is accessed by a single node v repeatedly. Then we get a high message/time complexity. Instead v could store the object, or at least cache it. But then, in case another node w accesses the object, we might run into consistency problems.
- Alternative idea: The accessing node should become the new master of the object. The shared object then becomes mobile. There exist several variants of this idea. The simplest version is a home-based solution like in Mobile IP (see Algorithm 6.2).

Algorithm 6.2 Shared Object: Home-Based Solution

Initialization: An object has a home base (a node) that is known to every node. All requests (accesses to the shared object) are routed through the home base.

Accessing Object: (by node v)

1: v acquires a lock at the home base, receives object.

Remarks:

- Home-based solutions suffer from the triangular routing problem. If two close-by nodes take turns to access the object, all the traffic is routed through the potentially far away home-base.

6.2 Arrow and Friends

We will now look at a protocol (called the Arrow algorithm) that always moves the shared object to the node currently accessing it without creating the triangular routing problem of home-based solutions. The protocol runs on a precomputed spanning tree. Assume that the spanning tree is rooted at the current position of the shared object. When a node u wants to access the shared object, it sends out a *find* request towards the current position of the object. While searching for the object, the edges of the spanning tree are redirected such that in the end, the spanning tree is rooted at u (i.e., the new holder of the object). The details of the algorithm are given by Algorithm 6.3. For simplicity, we assume that a node u only starts a find request if u is not currently the holder of the shared object and if u has finished all previous find requests (i.e., it is not currently waiting to receive the object).

Remarks:

- The parent pointers in Algorithm 6.3 are only needed for the find operation. Sending the variable to u in line 13 or to w.successor in line 23 is done using routing (on the spanning tree or on the underlying network).
- When we draw the parent pointers as arrows, in a quiescent moment (where no "find" is in motion), the arrows all point towards the node currently holding the variable (i.e., the tree is rooted at the node holding the variable)
- What is really great about the Arrow algorithm is that it works in a completely asynchronous and concurrent setting (i.e., there can be many find requests at the same time).

Theorem 6.4. *(Arrow, Analysis) In an asynchronous and concurrent setting, a "find" operation terminates with message and time complexity D, where D is the diameter of the spanning tree.*

Algorithm 6.3 Shared Object: Arrow Algorithm

Initialization: As for Algorithm 6.1, we are given a rooted spanning tree. Each node has a pointer to its parent, the root r is its own parent. The variable is initially stored at r. For all nodes v, v.successor := **null**, v.wait := **false**.

Start Find Request at Node u:

```
1: do atomically
2:    u sends "find by u" message to parent node
3:    u.parent := u
4:    u.wait := true
5: end do
```

Upon w Receiving "Find by u" Message from Node v:

```
6: do atomically
7:    if w.parent ≠ w then
8:       w sends "find by u" message to parent
9:       w.parent := v
10:   else
11:      w.parent := v
12:      if not w.wait then
13:         send variable to u      // w holds var. but does not need it any more
14:      else
15:         w.successor := u                // w will send variable to u a.s.a.p.
16:      end if
17:   end if
18: end do
```

Upon w Receiving Shared Object:

```
19: perform operation on shared object
20: do atomically
21:    w.wait := false
22:    if w.successor ≠ null then
23:       send variable to w.successor
24:       w.successor := null
25:    end if
26: end do
```


Before proving Theorem 6.4, we prove the following lemma.

Lemma 6.5. *An edge $\{u, v\}$ of the spanning tree is in one of four states:*

1.) Pointer from u to v (no message on the edge, no pointer from v to u)
2.) Message on the move from u to v (no pointer along the edge)
3.) Pointer from v to u (no message on the edge, no pointer from u to v)
4.) Message on the move from v to u (no pointer along the edge)

Proof. W.l.o.g., assume that initially the edge $\{u, v\}$ is in state 1. With a message arrival at u (or if u starts a "find by u" request, the edge goes to state 2. When the message is received at v, v directs its pointer to u and we are therefore in state 3. A new message at v (or a new request initiated by v) then brings the edge back to state 1. □

Proof of Theorem 6.4. Since the "find" message will only travel on a static tree, it suffices to show that it will not traverse an edge twice. Suppose for the sake of contradiction that there is a first "find" message f that traverses an edge $e = \{u, v\}$ for the second time and assume that e is the first edge that is traversed twice by f. The first time, f traverses e. Assume that e is first traversed from u to v. Since we are on a tree, the second time, e must be traversed from v to u. Because e is the first edge to be traversed twice, f must re-visit e before visiting any other edges. Right before f reaches v, the edge e is in state 2 (f is on the move) and in state 3 (it will immediately return with the pointer from v to u). This is a contradiction to Lemma 6.5. □

Remarks:

- Finding a good tree is an interesting problem. We would like to have a tree with low stretch, low diameter, low degree, etc.
- It seems that the Arrow algorithm works especially well when lots of "find" operations are initiated concurrently. Most of them will find a "close-by" node, thus having low message/time complexity. For the sake of simplicity we analyze a synchronous system.

Theorem 6.6. *(Arrow, Concurrent Analysis) Let the system be synchronous. Initially, the system is in a quiescent state. At time 0, a set S of nodes initiates a "find" operation. The message complexity of all "find" operations is $\mathcal{O}(\log |S| \cdot m^*)$ where m^* is the message complexity of an optimal (with global knowledge) algorithm on the tree.*

Proof Sketch. Let d be the minimum distance of any node in S to the root. There will be a node u_1 at distance d from the root that reaches the root in d time steps, turning all the arrows on the path to the root towards u_1. A node u_2 that finds (is queued behind) u_1 cannot distinguish the system from a system where there was no request u_1, and instead the root was initially located at u_1. The message cost of u_2 is consequentially the distance between u_1 and u_2 on the spanning tree. By induction the total message complexity is exactly as if a collector starts at the root and then "greedily" collects tokens located at the nodes in S (greedily in the sense that the collector always goes towards the closest token). Greedy collecting the tokens is not a good strategy in general because it will traverse the same edge more than twice in the worst

case. An asymptotically optimal algorithm can also be translated into a depth-first-search collecting paradigm, traversing each edge at most twice. In another area of computer science, we would call the Arrow algorithm a nearest-neighbor TSP heuristic (without returning to the start/root though), and the optimal algorithm TSP-optimal. It was shown that nearest-neighbor has a logarithmic overhead, which concludes the proof. □

Remarks:

- An average request set S on a not-too-bad tree gives usually a much better bound. However, there is an almost tight $\log|S|/\log\log|S|$ worst-case example.
- It was recently shown that Arrow can do as good in a dynamic setting (where nodes are allowed to initiate requests at any time). In particular the message complexity of the dynamic analysis can be shown to have a $\log D$ overhead only, where D is the diameter of the spanning tree (note that for logarithmic trees, the overhead becomes $\log\log n$).
- What if the spanning tree is a star? Then with Theorem 6.4, each find will terminate in 2 steps! Since also an optimal algorithm has message cost 1, the algorithm is 2-competitive...? Yes, but because of its high degree the star center experiences contention... It can be shown that the contention overhead is at most proportional to the largest degree Δ of the spanning tree.
- Thought experiment: Assume a balanced binary spanning tree—by Theorem 6.4, the message complexity per operation is $\log n$. Because a binary tree has maximum degree 3, the time per operation therefore is at most $3\log n$.
- There are better and worse choices for the spanning tree. The stretch of an edge $\{u,v\}$ is defined as distance between u and v in a spanning tree. The maximum stretch of a spanning tree is the maximum stretch over all edges. A few years ago, it was shown how to construct spanning trees that are $\mathcal{O}(\log n)$-stretch-competitive.

What if most nodes just want to read the shared object? Then it does not make sense to acquire a lock every time. Instead we can use caching (see Algorithm 6.7).

Theorem 6.8. *Algorithm 6.7 is correct. More surprisingly, the message complexity is 3-competitive (at most a factor 3 worse than the optimum).*

Proof. Since the accesses do not overlap by definition, it suffices to show that between two writes, we are 3-competitive. The sequence of accessing nodes is w_0, r_1, r_2, ..., r_k, w_1. After w_0, the object is stored at w_0 and not cached anywhere else. All reads cost twice the smallest subtree T spanning the write w_0 and all the reads since each read only goes to the first copy. The write w_1 costs T plus the path P from w_1 to T. Since any data management scheme must use an edge in T and P at least once, and our algorithm uses edges in T at most 3 times (and in P at most once), the theorem follows. □

Algorithm 6.7 Shared Object: Read/Write Caching

- Nodes can either read or write the shared object. For simplicity we first assume that reads or writes do not overlap in time (access to the object is sequential).
- Nodes store three items: a parent pointer pointing to one of the neighbors (as with Arrow), and a cache bit for each edge, plus (potentially) a copy of the object.
- Initially the object is stored at a single node u; all the parent pointers point towards u, all the cache bits are false.
- When initiating a read, a message follows the arrows (this time: without inverting them!) until it reaches a cached version of the object. Then a copy of the object is cached along the path back to the initiating node, and the cache bits on the visited edges are set to true.
- A write at u writes the new value locally (at node u), then searches (follow the parent pointers and reverse them towards u) a first node with a copy. Delete the copy and follow (in parallel, by flooding) all edge that have the cache flag set. Point the parent pointer towards u, and remove the cache flags.

Remarks:

- Concurrent reads are not a problem, also multiple concurrent reads and one write work just fine.
- What about concurrent writes? To achieve consistency writes need to invalidate the caches before writing their value. It is claimed that the strategy then becomes 4-competitive.
- Is the algorithm also time competitive? Well, not really: The optimal algorithm that we compare to is usually offline. This means it knows the whole access sequence in advance. It can then cache the object before the request even appears!
- Algorithms on trees are often simpler, but have the disadvantage that they introduce the extra stretch factor. In a ring, for example, any tree has stretch $n-1$; so there is always a bad request pattern.

Algorithm 6.9 Shared Object: Pointer Forwarding

Initialization: Object is stored at root r of a precomputed spanning tree T (as in the Arrow algorithm, each node has a parent pointer pointing towards the object).

Accessing Object: (by node u)

1: follow parent pointers to current root r of T
2: send object from r to u
3: r.parent := u; u.parent := u; *// u is the new root*

Algorithm 6.10 Shared Object: Ivy

Initialization: Object is stored at root r of a precomputed spanning tree T (as before, each node has a parent pointer pointing towards the object). For simplicity, we assume that accesses to the object are sequential.

Start Find Request at Node u:

1: u sends "find by u" message to parent node
2: u.parent := u

Upon v receiving "Find by u" Message:

3: **if** v.parent = v **then**
4: send object to u
5: **else**
6: send "find by u" message to v.parent
7: **end if**
8: v.parent := u *// u will become the new root*

6.3 Ivy and Friends

In the following we study algorithms that do not restrict communication to a tree. Of particular interest is the special case of a complete graph (clique). A simple solution for this case is given by Algorithm 6.9.

Remarks:

- If the graph is not complete, routing can be used to find the root.
- Assume that the nodes line up in a linked list. If we always choose the first node of the linked list to acquire the object, we have message/time complexity n. The new topology is again a linear linked list. Pointer forwarding is therefore bad in a worst-case.
- If edges are not FIFO, it can even happen that the number of steps is unbounded for a node having bad luck. An algorithm with such a property is named "not fair," or "not wait-free." (Example: Initially we have the list $4 \to 3 \to 2 \to 1$; 4 starts a find; when the message of 4 passes 3, 3 itself starts a find. The message of 3 may arrive at 2 and then 1 earlier, thus the new end of the list is $2 \to 1 \to 3$; once the message of 4 passes 2, the game re-starts.)

There seems to be a natural improvement of the pointer forwarding idea. Instead of simply redirecting the parent pointer from the old root to the new root, we can redirect all the parent pointers of the nodes on the path visited

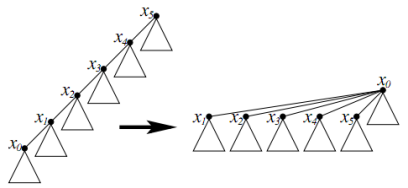


Figure 6.11: Reversal of the path $x_0, x_1, x_2, x_3, x_4, x_5$.

during a find message to the new root. The details are given by Algorithm 6.10. Figure 6.11 shows how the pointer redirecting affects a given tree (the right tree results from a find request started at node x_0 on the left tree).

Remarks:

- Also with Algorithm 6.10, we might have a bad linked list situation. However, if the start of the list acquires the object, the linked list turns into a star. As the following theorem shows, the search paths are not long on average. Since paths sometimes can be bad, we will need amortized analysis.

Theorem 6.12. *If the initial tree is a star, a find request of Algorithm 6.10 needs at most* $\log n$ *steps on average, where* n *is the number of processors.*

Proof. All logarithms in the following proof are to base 2. We assume that accesses to the shared object are sequential. We use a potential function argument. Let $s(u)$ be the size of the subtree rooted at node u (the number of nodes in the subtree including u itself). We define the potential Φ of the whole tree T as (V is the set of all nodes)

$$\Phi(T) = \sum_{u \in V} \frac{\log s(u)}{2}.$$

Assume that the path traversed by the i^{th} operation has length k_i, i.e., the i^{th} operation redirects k_i pointers to the new root. Clearly, the number of steps of the i^{th} operation is proportional to k_i. We are interested in the cost of m consecutive operations, $\sum_{i=1}^{m} k_i$.

Let T_0 be the initial tree and let T_i be the tree after the i^{th} operation. Further, let $a_i = k_i - \Phi(T_{i-1}) + \Phi(T_i)$ be the *amortized cost* of the i^{th} operation. We have

$$\sum_{i=1}^{m} a_i = \sum_{i=1}^{m} \big(k_i - \Phi(T_{i-1}) + \Phi(T_i)\big) = \sum_{i=1}^{m} k_i - \Phi(T_0) + \Phi(T_m).$$

For any tree T, we have $\Phi(T) \geq \log(n)/2$. Because we assume that T_0 is a star, we also have $\Phi(T_0) = \log(n)/2$. We therefore get that

$$\sum_{i=1}^{m} a_i \geq \sum_{i=1}^{m} k_i.$$

Hence, it suffices to upper bound the amortized cost of every operation. We thus analyze the amortized cost a_i of the i^{th} operation. Let $x_0, x_1, x_2, \ldots, x_{k_i}$ be the path that is reversed by the operation. Further for $0 \leq j \leq k_i$, let s_j be the size of the subtree rooted at x_j before the reversal. The size of the subtree rooted at x_0 after the reversal is s_{k_i} and the size of the one rooted at x_j after the reversal, for $1 \leq j \leq k_i$, is $s_j - s_{j-1}$ (see Figure 6.11). For all other nodes, the sizes of their subtrees are the same, therefore the corresponding terms cancel out in the ammortized cost a_i. We can thus write a_i as

$$\begin{aligned} a_i &= k_i - \left(\sum_{j=0}^{k_i} \frac{1}{2}\log s_j\right) + \left(\frac{1}{2}\log s_{k_i} + \sum_{j=1}^{k_i} \frac{1}{2}\log(s_j - s_{j-1})\right) \\ &= k_i + \frac{1}{2}\cdot\sum_{j=0}^{k_i-1}\big(\log(s_{j+1} - s_j) - \log s_j\big) \\ &= k_i + \frac{1}{2}\cdot\sum_{j=0}^{k_i-1}\log\left(\frac{s_{j+1}-s_j}{s_j}\right). \end{aligned}$$

For $0 \leq j \leq k_i - 1$, let $\alpha_j = s_{j+1}/s_j$. Note that $s_{j+1} > s_j$ and thus that $\alpha_j > 1$. Further note, that $(s_{j+1} - s_j)/s_j = \alpha_j - 1$. We therefore have that

$$\begin{aligned} a_i &= k_i + \frac{1}{2}\cdot\sum_{j=0}^{k_i-1}\log(\alpha_j - 1) \\ &= \sum_{j=0}^{k_i-1}\left(1 + \frac{1}{2}\log(\alpha_j - 1)\right). \end{aligned}$$

For $\alpha > 1$, it can be shown that $1 + \log(\alpha - 1)/2 \leq \log\alpha$ (see Lemma 6.13). From this inequality, we obtain

$$\begin{aligned} a_i &\leq \sum_{j=0}^{k_i-1}\log\alpha_j = \sum_{j=0}^{k_i-1}\log\frac{s_{j+1}}{s_j} = \sum_{j=0}^{k_i-1}(\log s_{j+1} - \log s_j) \\ &= \log s_{k_i} - \log s_0 \leq \log n, \end{aligned}$$

because $s_{k_i} = n$ and $s_0 \geq 1$. This concludes the proof. □

Lemma 6.13. *For $\alpha > 1$, $1 + \log(\alpha - 1)/2 \leq \log\alpha$.*

Proof. The claim can be verified by the following chain of reasoning:

$$\begin{aligned} 0 &\leq (\alpha - 2)^2 \\ 0 &\leq \alpha^2 - 4\alpha + 4 \\ 4(\alpha - 1) &\leq \alpha^2 \\ \log_2\big(4(\alpha - 1)\big) &\leq \log_2\big(\alpha^2\big) \\ 2 + \log_2(\alpha - 1) &\leq 2\log_2\alpha \\ 1 + \frac{1}{2}\log_2(\alpha - 1) &\leq \log_2\alpha. \end{aligned}$$

□

Remarks:

- Systems guys (the algorithm is called Ivy because it was used in a system with the same name) have some fancy heuristics to improve performance even more: For example, the root every now and then broadcasts its name such that paths will be shortened.

- What about concurrent requests? It works with the same argument as in Arrow. Also for Ivy an argument including congestion is missing (and more pressing, since the dynamic topology of a tree cannot be chosen to have low degree and thus low congestion as in Arrow).

- Sometimes the type of accesses allows that several accesses can be combined into one to reduce congestion higher up the tree. Let the tree in Algorithm 6.1 be a balanced binary tree. If the access to a shared variable for example is "add value x to the shared variable", two or more accesses that accidentally meet at a node can be combined into one. Clearly accidental meeting is rare in an asynchronous model. We might be able to use synchronizers (or maybe some other timing tricks) to help meeting a little bit.

Chapter Notes

The Arrow protocol was designed by Raymond [Ray89]. There are real life implementations of the Arrow protocol, such as the Aleph Toolkit [Her99]. The performance of the protocol under high loads was tested in [HW99] and other implementations and variations of the protocol were given in, e.g., [PR99, HTW00].

It has been shown that the find operations of the protocol do not backtrack, i.e., the time and message complexities are $\mathcal{O}(D)$ [DH98], and that the Arrow protocol is fault tolerant [HT01]. Given a set of concurrent request, Herlihy et al. [HTW01] showed that the time and message complexities are within factor $\log R$ from the optimal, where R is the number of requests. Later, this analysis was extended to long-lived and asynchronous systems. In particular, Herlihy et al. [HKTW06] showed that the competitive ratio in this asynchronous concurrent setting is $\mathcal{O}(\log D)$. Thanks to the lower bound of the greedy TSP heuristic, this is almost tight.

The Ivy system was introduced in [Li88, LH89]. On the theory side, it was shown by Ginat et al. [GST89] that the amortized cost of a single request of the Ivy protocol is $\Theta(\log n)$. Closely related work to the Ivy protocol on the practical side is research on virtual memory and parallel computing on loosely coupled multiprocessors. For example [BB81, LSHL82, FR86] contain studies on variations of the network models, limitations on data sharing between processes and different approaches.

Later, the research focus shifted towards systems where most data operations were read operations, i.e., efficient caching became one of the main objects of study, e.g., [MMVW97].

Bibliography

[BB81] Thomas J. Buckholtz and Helen T. Buckholtz. Apollo Domain Architecture. Technical report, Apollo Computer, Inc., 1981.

[DH98] Michael J. Demmer and Maurice Herlihy. The Arrow Distributed Directory Protocol. In *Proceedings of the 12th International Symposium on Distributed Computing (DISC)*, 1998.

[FR86] Robert Fitzgerald and Richard F. Rashid. The Integration of Virtual Memory Management and Interprocess Communication in Accent. *ACM Transactions on Computer Systems*, 4(2):147–177, 1986.

[GST89] David Ginat, Daniel Sleator, and Robert Tarjan. A Tight Amortized Bound for Path Reversal. *Information Processing Letters*, 31(1):3–5, 1989.

[Her99] Maurice Herlihy. The Aleph Toolkit: Support for Scalable Distributed Shared Objects. In *Proceedings of the Third International Workshop on Network-Based Parallel Computing: Communication, Architecture, and Applications (CANPC)*, pages 137–149, 1999.

[HKTW06] Maurice Herlihy, Fabian Kuhn, Srikanta Tirthapura, and Roger Wattenhofer. Dynamic Analysis of the Arrow Distributed Protocol. In *Theory of Computing Systems, Volume 39, Number 6*, November 2006.

[HT01] Maurice Herlihy and Srikanta Tirthapura. Self Stabilizing Distributed Queuing. In *Proceedings of the 15th International Conference on Distributed Computing (DISC)*, pages 209–223, 2001.

[HTW00] Maurice Herlihy, Srikanta Tirthapura, and Roger Wattenhofer. Ordered Multicast and Distributed Swap. In *Operating Systems Review, Volume 35/1, 2001. Also in PODC Middleware Symposium, Portland, Oregon*, July 2000.

[HTW01] Maurice Herlihy, Srikanta Tirthapura, and Roger Wattenhofer. Competitive Concurrent Distributed Queuing. In *Twentieth ACM Symposium on Principles of Distributed Computing (PODC)*, August 2001.

[HW99] Maurice Herlihy and Michael Warres. A Tale of Two Directories: Implementing Distributed Shared Objects in Java. In *Proceedings of the ACM 1999 conference on Java Grande (JAVA)*, pages 99–108, 1999.

[LH89] Kai Li and Paul Hudak. Memory Coherence in Shared Virtual Memory Systems. *ACM Transactions on Computer Systems*, 7(4):312–359, November 1989.

[Li88] Kai Li. IVY: Shared Virtual Memory System for Parallel Computing. In *International Conference on Parallel Processing*, 1988.

[LSHL82] Paul J. Leach, Bernard L. Stumpf, James A. Hamilton, and Paul H. Levine. UIDs as Internal Names in a Distributed File System. In *Proceedings of the First ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing (PODC)*, pages 34–41, 1982.

[MMVW97] B. Maggs, F. Meyer auf der Heide, B. Voecking, and M. Westermann. Exploiting Locality for Data Management in Systems of Limited Bandwidth. In *IEEE Symposium on Foundations of Computer Science (FOCS)*, 1997.

[PR99] David Peleg and Eilon Reshef. A Variant of the Arrow Distributed Directory Protocol with Low Average Complexity. In *Proceedings of the 26th International Colloquium on Automata, Languages and Programming (ICALP)*, pages 615–624, 1999.

[Ray89] Kerry Raymond. A Tree-based Algorithm for Distributed Mutual Exclusion. *ACM Transactions on Computer Systems*, 7:61–77, 1989.

Chapter 7

Maximal Independent Set

In this chapter we present a highlight of this course, a fast maximal independent set (MIS) algorithm. The algorithm is the first randomized algorithm that we study in this class. In distributed computing, randomization is a powerful and therefore omnipresent concept, as it allows for relatively simple yet efficient algorithms. As such the studied algorithm is archetypal.

A MIS is a basic building block in distributed computing, some other problems pretty much follow directly from the MIS problem. At the end of this chapter, we will give two examples: matching and vertex coloring (see Chapter 1).

7.1 MIS

Definition 7.1 (Independent Set)**.** *Given an undirected Graph $G = (V, E)$ an* independent set *is a subset of nodes $U \subseteq V$, such that no two nodes in U are adjacent. An independent set is* maximal *if no node can be added without violating independence. An independent set of maximum cardinality is called* maximum.

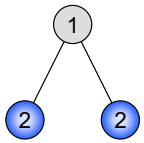


Figure 7.2: Example graph with 1) a maximal independent set (MIS) and 2) a maximum independent set (MaxIS).

Remarks:

- Computing a maximum independent set (MaxIS) is a notoriously difficult problem. It is equivalent to maximum clique on the complementary graph. Both problems are NP-hard, in fact not approximable within $n^{\frac{1}{2}-\epsilon}$ within polynomial time.
- In this course we concentrate on the maximal independent set (MIS) problem. Please note that MIS and MaxIS can be quite different, indeed e.g. on a star graph there exists an MIS that is $\Theta(n)$ smaller than the MaxIS (cf. Figure 7.2).
- Computing a MIS sequentially is trivial: Scan the nodes in arbitrary order. If a node u does not violate independence, add u to the MIS. If u violates independence, discard u. So the only question is how to compute a MIS in a distributed way.

Algorithm 7.3 Slow MIS

Require: Node IDs

Every node v executes the following code:

1: **if** all neighbors of v with larger identifiers have decided not to join the MIS **then**
2: v decides to join the MIS
3: **end if**

Remarks:

- Not surprisingly the slow algorithm is not better than the sequential algorithm in the worst case, because there might be one single point of activity at any time. Formally:

Theorem 7.4 (Analysis of Algorithm 7.3)**.** *Algorithm 7.3 features a time complexity of $\mathcal{O}(n)$ and a message complexity of $\mathcal{O}(m)$.*

Remarks:

- This is not very exciting.
- There is a relation between independent sets and node coloring (Chapter 1), since each color class is an independent set, however, not necessarily a MIS. Still, starting with a coloring, one can easily derive a MIS algorithm: In the first round all nodes of the first color join the MIS and notify their neighbors. Then, all nodes of the second color which do not have a neighbor that is already in the MIS join the MIS and inform their neighbors. This process is repeated for all colors. Thus the following corollary holds:

Corollary 7.5. *Given a coloring algorithm that runs in time T and needs C colors, we can construct a MIS in time $T + C$.*

Remarks:

- Using Theorem 1.23 and Corollary 7.5 we get a distributed deterministic MIS algorithm for trees (and for bounded degree graphs) with time complexity $\mathcal{O}(\log^* n)$.
- With a lower bound argument one can show that this deterministic MIS algorithm is asymptotically optimal for rings.
- There have been attempts to extend Algorithm 1.17 to more general graphs, however, so far without much success. Below we present a radically different approach that uses randomization.

7.2 Original Fast MIS

Algorithm 7.6 Fast MIS

The algorithm operates in synchronous rounds, grouped into phases.
A single phase is as follows:
1) Each node v marks itself with probability $\frac{1}{2d(v)}$, where $d(v)$ is the current degree of v.
2) If no higher degree neighbor of v is also marked, node v joins the MIS. If a higher degree neighbor of v is marked, node v unmarks itself again. (If the neighbors have the same degree, ties are broken arbitrarily, e.g., by identifier).
3) Delete all nodes that joined the MIS and their neighbors, as they cannot join the MIS anymore.

Remarks:

- Correctness in the sense that the algorithm produces an independent set is relatively simple: Steps 1 and 2 make sure that if a node v joins the MIS, then v's neighbors do not join the MIS at the same time. Step 3 makes sure that v's neighbors will never join the MIS.
- Likewise the algorithm eventually produces a MIS, because the node with the highest degree will mark itself at some point in Step 1.
- So the only remaining question is how fast the algorithm terminates. To understand this, we need to dig a bit deeper.

Lemma 7.7 (Joining MIS)**.** *A node v joins the MIS in Step 2 with probability $p \geq \frac{1}{4d(v)}$.*

Proof: Let M be the set of marked nodes in Step 1 and MIS be the set of nodes that join the MIS in Step 2. Let $H(v)$ be the set of neighbors of v with higher degree, or same degree and higher identifier. Using independence of the random

choices of v and nodes in $H(v)$ in Step 1 we get

$$\begin{aligned} P[v \notin \text{MIS} | v \in M] &= P[\text{there is a node } w \in H(v), w \in M | v \in M] \\ &= P[\text{there is a node } w \in H(v), w \in M] \\ &\leq \sum_{w \in H(v)} P[w \in M] = \sum_{w \in H(v)} \frac{1}{2d(w)} \\ &\leq \sum_{w \in H(v)} \frac{1}{2d(v)} \leq \frac{d(v)}{2d(v)} = \frac{1}{2}. \end{aligned}$$

Then

$$P[v \in \text{MIS}] = P[v \in \text{MIS} | v \in M] \cdot P[v \in M] \geq \frac{1}{2} \cdot \frac{1}{2d(v)}.$$

□

Lemma 7.8 (Good Nodes)**.** *A node v is called* good *if*

$$\sum_{w \in N(v)} \frac{1}{2d(w)} \geq \frac{1}{6},$$

where $N(v)$ is the set of neighbors of v. Otherwise we call v a bad *node. A good node will be removed in Step 3 with probability $p \geq \frac{1}{36}$.*

Proof: Let node v be good. Intuitively, good nodes have lots of low-degree neighbors, thus chances are high that one of them goes into the independent set, in which case v will be removed in Step 3 of the algorithm.

If there is a neighbor $w \in N(v)$ with degree at most 2 we are done: With Lemma 7.7 the probability that node w joins the MIS is at least $\frac{1}{8}$, and our good node will be removed in Step 3.

So all we need to worry about is that all neighbors have at least degree 3: For any neighbor w of v we have $\frac{1}{2d(w)} \leq \frac{1}{6}$. Since $\sum_{w \in N(v)} \frac{1}{2d(w)} \geq \frac{1}{6}$ there is a subset of neighbors $S \subseteq N(v)$ such that $\frac{1}{6} \leq \sum_{w \in S} \frac{1}{2d(w)} \leq \frac{1}{3}$

We can now bound the probability that node v will be removed. Let therefore R be the event of v being removed. Again, if a neighbor of v joins the MIS in Step 2, node v will be removed in Step 3. We have

$$\begin{aligned} P[R] &\geq P[\text{there is a node } u \in S, u \in \text{MIS}] \\ &\geq \sum_{u \in S} P[u \in \text{MIS}] - \sum_{u,w \in S; u \neq w} P[u \in \text{MIS } \textbf{and } w \in \text{MIS}]. \end{aligned}$$

For the last inequality we used the inclusion-exclusion principle truncated after the second order terms. Let M again be the set of marked nodes after

Step 1. Using $P[u \in M] \geq P[u \in \text{MIS}]$ we get

$$\begin{aligned} P[R] &\geq \sum_{u \in S} P[u \in \text{MIS}] - \sum_{u,w \in S; u \neq w} P[u \in M \textbf{ and } w \in M] \\ &\geq \sum_{u \in S} P[u \in \text{MIS}] - \sum_{u \in S} \sum_{w \in S} P[u \in M] \cdot P[w \in M] \\ &\geq \sum_{u \in S} \frac{1}{4d(u)} - \sum_{u \in S} \sum_{w \in S} \frac{1}{2d(u)} \frac{1}{2d(w)} \\ &\geq \sum_{u \in S} \frac{1}{2d(u)} \left(\frac{1}{2} - \sum_{w \in S} \frac{1}{2d(w)} \right) \geq \frac{1}{6} \left(\frac{1}{2} - \frac{1}{3} \right) = \frac{1}{36}. \end{aligned}$$

□

Remarks:

- We would be almost finished if we could prove that many nodes are good in each phase. Unfortunately this is not the case: In a star-graph, for instance, only a single node is good! We need to find a work-around.

Lemma 7.9 (Good Edges)**.** *An edge $e = (u, v)$ is called* bad *if both u and v are bad; else the edge is called* good*. The following holds: At any time at least half of the edges are good.*

Proof: For the proof we construct a directed auxiliary graph: Direct each edge towards the higher degree node (if both nodes have the same degree direct it towards the higher identifier). Now we need a little helper lemma before we can continue with the proof.

Lemma 7.10. *A bad node has outdegree (number of edges pointing away from bad node) at least twice its indegree (number of edges pointing towards bad node).*

Proof: For the sake of contradiction, assume that a bad node v does not have outdegree at least twice its indegree. In other words, at least one third of the neighbor nodes (let's call them S) have degree at most $d(v)$. But then

$$\sum_{w \in N(v)} \frac{1}{2d(w)} \geq \sum_{w \in S} \frac{1}{2d(w)} \geq \sum_{w \in S} \frac{1}{2d(v)} \geq \frac{d(v)}{3} \frac{1}{2d(v)} = \frac{1}{6}$$

which means v is good, a contradiction. □

Continuing the proof of Lemma 7.9: According to Lemma 7.10 the number of edges directed into bad nodes is at most half the number of edges directed out of bad nodes. Thus, the number of edges directed into bad nodes is at most half the number of edges. Thus, at least half of the edges are directed into good nodes. Since these edges are not bad, they must be good.

Theorem 7.11 (Analysis of Algorithm 7.6)**.** *Algorithm 7.6 terminates in expected time $\mathcal{O}(\log n)$.*

Proof: With Lemma 7.8 a good node (and therefore a good edge!) will be deleted with constant probability. Since at least half of the edges are good (Lemma 7.9) a constant fraction of edges will be deleted in each phase.

More formally: With Lemmas 7.8 and 7.9 we know that at least half of the edges will be removed with probability at least 1/36. Let R be the number of edges to be removed in a certain phase. Using linearity of expectation (cf. Theorem 7.13) we know that $\mathbb{E}[R] \geq m/72$, m being the total number of edges at the start of the phase. Now let $p := P[R \leq \mathbb{E}[R]/2]$. Bounding the expectation yields

$$\begin{aligned} \mathbb{E}[R] = \sum_r P[R = r] \cdot r &\leq P[R \leq \mathbb{E}[R]/2] \cdot \mathbb{E}[R]/2 + P[R > \mathbb{E}[R]/2] \cdot m \\ &= p \cdot \mathbb{E}[R]/2 + (1-p) \cdot m. \end{aligned}$$

Solving for p we get

$$p \leq \frac{m - \mathbb{E}[R]}{m - \mathbb{E}[R]/2} < \frac{m - \mathbb{E}[R]/2}{m} \leq 1 - 1/144.$$

In other words, with probability at least 1/144 at least $m/144$ edges are removed in a phase. After expected $\mathcal{O}(\log m)$ phases all edges are deleted. Since $m \leq n^2$ and thus $\mathcal{O}(\log m) = O(\log n)$ the Theorem follows. □

Remarks:

- With a bit of more math one can even show that Algorithm 7.6 terminates in time $\mathcal{O}(\log n)$ "with high probability".

7.3 Fast MIS v2

Algorithm 7.12 Fast MIS 2

The algorithm operates in synchronous rounds, grouped into phases.
A single phase is as follows:
1) Each node v chooses a random value $r(v) \in [0, 1]$ and sends it to its neighbors.
2) If $r(v) < r(w)$ for all neighbors $w \in N(v)$, node v enters the MIS and informs its neighbors.
3) If v or a neighbor of v entered the MIS, v terminates (v and all edges adjacent to v are removed from the graph), otherwise v enters the next phase.

Remarks:

- Correctness in the sense that the algorithm produces an independent set is simple: Steps 1 and 2 make sure that if a node v joins the MIS, then v's neighbors do not join the MIS at the same time. Step 3 makes sure that v's neighbors will never join the MIS.
- Likewise the algorithm eventually produces a MIS, because the node with the globally smallest value will always join the MIS, hence there is progress.
- So the only remaining question is how fast the algorithm terminates. To understand this, we need to dig a bit deeper.

- Our proof will rest on a simple, yet powerful observation about expected values of random variables that *may not be independent*:

Theorem 7.13 (Linearity of Expectation)**.** *Let X_i, $i = 1, \ldots, k$ denote random variables, then*

$$\mathbb{E}\left[\sum_i X_i\right] = \sum_i \mathbb{E}[X_i].$$

Proof. It is sufficient to prove $\mathbb{E}[X+Y] = \mathbb{E}[X]+\mathbb{E}[Y]$ for two random variables X and Y, because then the statement follows by induction. Since

$$\begin{aligned} P[(X,Y)=(x,y)] &= P[X=x]\cdot P[Y=y|X=x] \\ &= P[Y=y]\cdot P[X=x|Y=y] \end{aligned}$$

we get that

$$\begin{aligned} \mathbb{E}[X+Y] &= \sum_{(X,Y)=(x,y)} P[(X,Y)=(x,y)]\cdot(x+y) \\ &= \sum_{X=x}\sum_{Y=y} P[X=x]\cdot P[Y=y|X=x]\cdot x \\ &+ \sum_{Y=y}\sum_{X=x} P[Y=y]\cdot P[X=x|Y=y]\cdot y \\ &= \sum_{X=x} P[X=x]\cdot x + \sum_{Y=y} P[Y=y]\cdot y \\ &= \mathbb{E}[X]+\mathbb{E}[Y]. \end{aligned}$$

□

Remarks:

- How can we prove that the algorithm only needs $\mathcal{O}(\log n)$ phases in expectation? It would be great if this algorithm managed to remove a constant fraction of nodes in each phase. Unfortunately, it does not.
- Instead we will prove that the number of *edges* decreases quickly. Again, it would be great if any single edge was removed with constant probability in Step 3. But again, unfortunately, this is not the case.
- Maybe we can argue about the expected number of edges to be removed in one single phase? Let's see: A node v enters the MIS with probability $1/(d(v)+1)$, where $d(v)$ is the degree of node v. By doing so, not only are v's edges removed, but indeed all the edges of v's neighbors as well – generally these are much more than $d(v)$ edges. So there is hope, but we need to be careful: If we do this the most naive way, we will count the same edge many times.
- How can we fix this? The nice observation is that it is enough to count just some of the removed edges. Given a new MIS node v and a neighbor $w \in N(v)$, we count the edges only if $r(v) < r(x)$ for all $x \in N(w)$. This looks promising. In a star graph, for instance, only the smallest random value can be accounted for removing all the edges of the star.

Lemma 7.14 (Edge Removal)**.** *In a single phase, we remove at least half of the edges in expectation.*

Proof. To simplify the notation, at the start of our phase, the graph is simply $G = (V, E)$. In addition, to ease presentation, we replace each undirected edge $\{v, w\}$ by the two directed edges (v, w) and (w, v).

Suppose that a node v joins the MIS in this phase, i.e., $r(v) < r(w)$ for all neighbors $w \in N(v)$. If in addition we have $r(v) < r(x)$ for all neighbors x of a neighbor w of v, we call this event $(v \to w)$. The probability of event $(v \to w)$ is at least $1/(d(v) + d(w))$, since $d(v) + d(w)$ is the maximum number of nodes adjacent to v or w (or both). As v joins the MIS, all (directed) edges (w, x) with $x \in N(w)$ will be removed; there are $d(w)$ of these edges.

We now count the removed edges. Whether we remove the edges adjacent to w because of event $(v \to w)$ is a random variable $X_{(v\to w)}$. If event $(v \to w)$ occurs, $X_{(v\to w)}$ has the value $d(w)$, if not it has the value 0. For each undirected edge $\{v, w\}$ we have two such variables, $X_{(v\to w)}$ and $X_{(w\to v)}$. Due to Theorem 7.13, the expected value of the sum X of all these random variables is at least

$$\begin{aligned} \mathbb{E}[X] &= \sum_{\{v,w\}\in E} \mathbb{E}[X_{(v\to w)}] + \mathbb{E}[X_{(w\to v)}] \\ &= \sum_{\{v,w\}\in E} P[\text{Event } (v\to w)]\cdot d(w) + P[\text{Event } (w\to v)]\cdot d(v) \\ &\geq \sum_{\{v,w\}\in E} \frac{d(w)}{d(v)+d(w)} + \frac{d(v)}{d(w)+d(v)} \\ &= \sum_{\{v,w\}\in E} 1 = |E|. \end{aligned}$$

In other words, in expectation $|E|$ directed edges are removed in a single phase! Note that we did not double count any edge removals, as a directed edge (w, x) can only be removed by an event $(v \to w)$. The event $(v \to w)$ inhibits a concurrent event $(v' \to w)$ since $r(v) < r(v')$ for all $v' \in N(w)$. We may have counted an undirected edge at most twice (once in each direction). So, in expectation at least half of the undirected edges are removed. □

Remarks:

- This enables us to follow a bound on the expected running time of Algorithm 7.12 quite easily.

Theorem 7.15 (Expected running time of Algorithm 7.12)**.** *Algorithm 7.12 terminates after at most $3\log_{4/3} m + 1 \in O(\log n)$ phases in expectation.*

Proof: The probability that in a single phase at least a quarter of all edges are removed is at least 1/3. For the sake of contradiction, assume not. Then with probability less than 1/3 we may be lucky and many (potentially all) edges are removed. With probability more than 2/3 less than 1/4 of the edges are removed. Hence the expected fraction of removed edges is strictly less than $1/3 \cdot 1 + 2/3 \cdot 1/4 = 1/2$. This contradicts Lemma 7.14.

Hence, in expectation at least every third phase is "good" and removes at least a quarter of the edges. To get rid of all but two edges we need $\log_{4/3} m$

good phases in expectation. The last two edges will certainly be removed in the next phase. Hence a total of $3\log_{4/3} m + 1$ phases are enough in expectation.

Remarks:

- Sometimes one expects a bit more of an algorithm: Not only should the expected time to terminate be good, but the algorithm should *always* terminate quickly. As this is impossible in randomized algorithms (after all, the random choices may be "unlucky" all the time!), researchers often settle for a compromise, and just demand that the probability that the algorithm does not terminate in the specified time can be made absurdly small. For our algorithm, this can be deduced from Lemma 7.14 and another standard tool, namely Chernoff's Bound.

Definition 7.16 (W.h.p.)**.** *We say that an algorithm terminates* w.h.p. *(with high probability) within $\mathcal{O}(t)$ time if it does so with probability at least $1 - 1/n^c$ for any choice of $c \geq 1$. Here c may affect the constants in the Big-O notation because it is considered a "tunable constant" and usually kept small.*

Definition 7.17 (Chernoff's Bound)**.** *Let $X = \sum_{i=1}^{k} X_i$ be the sum of k independent $0-1$ random variables. Then* Chernoff's bound *states that w.h.p.*

$$|X - \mathbb{E}[X]| \in O\left(\log n + \sqrt{\mathbb{E}[X] \log n}\right).$$

Corollary 7.18 (Running Time of Algorithm 7.12)**.** *Algorithm 7.12 terminates w.h.p. in $\mathcal{O}(\log n)$ time.*

Proof: In Theorem 7.15 we used that *independently* of everything that happened before, in each phase we have a constant probability p that a quarter of the edges are removed. Call such a phase *good*. For some constants C_1 and C_2, let us check after $C_1 \log n + C_2 \in O(\log n)$ phases, in how many phases at least a quarter of the edges have been removed. In expectation, these are at least $p(C_1 \log n + C_2)$ many. Now we look at the random variable $X = \sum_{i=1}^{C_1 \log n + C_2} X_i$, where the X_i are independent $0-1$ variables being one with exactly probability p. Certainly, if X is at least x with some probability, then the probability that we have x good phases can only be larger (if no edges are left, certainly "all" of the remaining edges are removed). To X we can apply Chernoff's bound. If C_1 and C_2 are chosen large enough, they will overcome the constants in the Big-O from Chernoff's bound, i.e., w.h.p. it holds that $|X - \mathbb{E}[X]| \leq \mathbb{E}[X]/2$, implying $X \geq \mathbb{E}[X]/2$. Choosing C_1 large enough, we will have w.h.p. sufficiently many good phases, i.e., the algorithm terminates w.h.p. in $\mathcal{O}(\log n)$ phases.

Remarks:

- The algorithm can be improved. Drawing random real numbers in each phase for instance is not necessary. One can achieve the same by sending only a total of $\mathcal{O}(\log n)$ random (and as many non-random) bits over each edge.
- One of the main open problems in distributed computing is whether one can beat this logarithmic time, or at least achieve it with a deterministic algorithm.
- Let's turn our attention to applications of MIS next.

7.4 Applications

Definition 7.19 (Matching)**.** *Given a graph $G = (V, E)$ a* matching *is a subset of edges $M \subseteq E$, such that no two edges in M are adjacent (i.e., where no node is adjacent to two edges in the matching). A matching is* maximal *if no edge can be added without violating the above constraint. A matching of maximum cardinality is called* maximum*. A matching is called* perfect *if each node is adjacent to an edge in the matching.*

Remarks:

- In contrast to MaxIS, a maximum matching can be found in polynomial time, and is also easy to approximate, since any maximal matching is a 2-approximation.
- An independent set algorithm is also a matching algorithm: Let $G = (V, E)$ be the graph for which we want to construct the matching. The so-called line graph G' is defined as follows: for every edge in G there is a node in G'; two nodes in G' are connected by an edge if their respective edges in G are adjacent. A (maximal) independent set in the line graph G' is a (maximal) matching in the original graph G, and vice versa. Using Algorithm 7.12 directly produces a $\mathcal{O}(\log n)$ bound for maximal matching.
- More importantly, our MIS algorithm can also be used for vertex coloring (Problem 1.1):

Algorithm 7.20 General Graph Coloring

1: Given a graph $G = (V, E)$ we virtually build a graph $G' = (V', E')$ as follows:
2: Every node $v \in V$ clones itself $d(v)+1$ times $(v_0, \ldots, v_{d(v)} \in V')$, $d(v)$ being the degree of v in G.
3: The edge set E' of G' is as follows:
4: First all clones are in a clique: $(v_i, v_j) \in E'$, for all $v \in V$ and all $0 \leq i < j \leq d(v)$
5: Second all i^{th} clones of neighbors in the original graph G are connected: $(u_i, v_i) \in E'$, for all $(u, v) \in E$ and all $0 \leq i \leq \min(d(u), d(v))$.
6: Now we simply run (simulate) the fast MIS Algorithm 7.12 on G'.
7: If node v_i is in the MIS in G', then node v gets color i.

Theorem 7.21 (Analysis of Algorithm 7.20)**.** *Algorithm 7.20 $(\Delta + 1)$-colors an arbitrary graph in $\mathcal{O}(\log n)$ time, with high probability, Δ being the largest degree in the graph.*

Proof: Thanks to the clique among the clones at most one clone is in the MIS. And because of the $d(v)+1$ clones of node v every node will get a free color! The running time remains logarithmic since G' has $\mathcal{O}\left(n^2\right)$ nodes and the exponent becomes a constant factor when applying the logarithm.

Remarks:

- This solves our open problem from Chapter 1.1!
- Together with Corollary 7.5 we get quite close ties between $(\Delta+1)$-coloring and the MIS problem.
- Computing a MIS also solves another graph problem on graphs of bounded independence.

Definition 7.22 (Bounded Independence)**.** *$G=(V,E)$ is of* bounded independence*, if for every node $v \in V$ the largest independent set in the neighborhood $N(v)$ is bounded by a constant.*

Definition 7.23 ((Minimum) Dominating Sets)**.** *A* dominating set *is a subset of the nodes such that each node is in the set or adjacent to a node in the set. A* minimum dominating set *is a dominating set containing the least possible number of nodes.*

Remarks:

- In general, finding a dominating set less than factor $\log n$ larger than an minimum dominating set is NP-hard.
- Any MIS is a dominating set: if a node was not covered, it could join the independent set.
- In general a MIS and a minimum dominating sets have not much in common (think of a star). For graphs of bounded independence, this is different.

Corollary 7.24. *On graphs of bounded independence, a constant-factor approximation to a minimum dominating set can be found in time $\mathcal{O}(\log n)$ w.h.p.*

Proof: Denote by M a minimum dominating set and by I a MIS. Since M is a dominating set, each node from I is in M or adjacent to a node in M. Since the graph is of bounded independence, no node in M is adjacent to more than constantly many nodes from I. Thus, $|I| \in O(|M|)$. Therefore, we can compute a MIS with Algorithm 7.12 and output it as the dominating set, which takes $\mathcal{O}(\log n)$ rounds w.h.p.

Chapter Notes

As we have seen, a MIS can be used in versatile ways. Indeed, it was once argued that the cells of a fly compute a MIS to decide where to grow hair [AAB+11]. The fast MIS algorithm is a simplified version of an algorithm by Luby [Lub86]. Around the same time there have been a number of other papers dealing with the same or related problems, for instance by Alon, Babai, and Itai [ABI86], or by Israeli and Itai [II86]. The analysis presented in Section 7.2 takes elements of all these papers, and from other papers on distributed weighted matching [WW04]. The analysis in the book [Pel00] by David Peleg is different, and only achieves $\mathcal{O}(\log^2 n)$ time. The new MIS variant (with the simpler analysis) of Section 7.3 is by Métivier, Robson, Saheb-Djahromi and Zemmari [MRSDZ11]. With

some adaptations, the algorithms [Lub86, MRSDZ11] only need to exchange a total of $\mathcal{O}(\log n)$ bits per node, which is asymptotically optimum, even on unoriented trees [KSOS06]. However, the distributed time complexity for MIS is still somewhat open, as the strongest lower bounds are $\Omega(\sqrt{\log n})$ or $\Omega(\log \Delta)$ [KMW04]. Recent research regarding the MIS problem focused on improving the $\mathcal{O}(\log n)$ time complexity for special graph classes, for instances growth-bounded graphs [SW08] or trees [LW11]. There are also results that depend on the degree of the graph [BE09, Kuh09]. Deterministic MIS algorithms are still far from the lower bounds, as the best deterministic MIS algorithm takes $2^{O(\sqrt{\log n})}$ time [PS96]. The maximum matching algorithm mentioned in the remarks is the blossom algorithm by Jack Edmonds.

Bibliography

[AAB+11] Yehuda Afek, Noga Alon, Omer Barad, Eran Hornstein, Naama Barkai, and Ziv Bar-Joseph. A Biological Solution to a Fundamental Distributed Computing Problem. volume 331, pages 183–185. American Association for the Advancement of Science, January 2011.

[ABI86] Noga Alon, László Babai, and Alon Itai. A Fast and Simple Randomized Parallel Algorithm for the Maximal Independent Set Problem. *J. Algorithms*, 7(4):567–583, 1986.

[BE09] Leonid Barenboim and Michael Elkin. Distributed (delta+1)-coloring in linear (in delta) time. In *41st ACM Symposium On Theory of Computing (STOC)*, 2009.

[II86] Amos Israeli and Alon Itai. A Fast and Simple Randomized Parallel Algorithm for Maximal Matching. *Inf. Process. Lett.*, 22(2):77–80, 1986.

[KMW04] F. Kuhn, T. Moscibroda, and R. Wattenhofer. What Cannot Be Computed Locally! In *Proceedings of the 23rd ACM Symposium on Principles of Distributed Computing (PODC)*, July 2004.

[KSOS06] Kishore Kothapalli, Christian Scheideler, Melih Onus, and Christian Schindelhauer. Distributed coloring in $O(\sqrt{\log n})$ Bit Rounds. In *20th international conference on Parallel and Distributed Processing (IPDPS)*, 2006.

[Kuh09] Fabian Kuhn. Weak graph colorings: distributed algorithms and applications. In *21st ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, 2009.

[Lub86] Michael Luby. A Simple Parallel Algorithm for the Maximal Independent Set Problem. *SIAM J. Comput.*, 15(4):1036–1053, 1986.

[LW11] Christoph Lenzen and Roger Wattenhofer. MIS on trees. In *PODC*, pages 41–48, 2011.

[MRSDZ11] Yves Métivier, John Michael Robson, Nasser Saheb-Djahromi, and Akka Zemmari. An optimal bit complexity randomized distributed MIS algorithm. *Distributed Computing*, 23(5-6):331–340, 2011.

[Pel00] David Peleg. *Distributed Computing: a Locality-Sensitive Approach*. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 2000.

[PS96] Alessandro Panconesi and Aravind Srinivasan. On the Complexity of Distributed Network Decomposition. *J. Algorithms*, 20(2):356–374, 1996.

[SW08] Johannes Schneider and Roger Wattenhofer. A Log-Star Distributed Maximal Independent Set Algorithm for Growth-Bounded Graphs. In *27th ACM Symposium on Principles of Distributed Computing (PODC), Toronto, Canada*, August 2008.

[WW04] Mirjam Wattenhofer and Roger Wattenhofer. Distributed Weighted Matching. In *18th Annual Conference on Distributed Computing (DISC), Amsterdam, Netherlands*, October 2004.

Chapter 8

Locality Lower Bounds

In Chapter 1, we looked at distributed algorithms for coloring. In particular, we saw that rings and rooted trees can be colored with 3 colors in $\log^* n + O(1)$ rounds.

8.1 Model

In this chapter, we will reconsider the distributed coloring problem. We will look at a classic lower bound that shows that the result of Chapter 1 is tight: Coloring rings (and rooted trees) indeed requires $\Omega(\log^* n)$ rounds. In particular, we will prove a lower bound for coloring in the following setting:

- We consider deterministic, synchronous algorithms.
- Message size and local computations are unbounded.
- We assume that the network is a directed ring with n nodes.
- Nodes have unique labels (identifiers) from 1 to n.

Remarks:

- A generalization of the lower bound to randomized algorithms is possible.
- Except for restricting to deterministic algorithms, all the conditions above make a lower bound stronger: Any lower bound for synchronous algorithms certainly also holds for asynchronous ones. A lower bound that is true if message size and local computations are not restricted is clearly also valid if we require a bound on the maximal message size or the amount of local computations. Similarly, assuming that the ring is directed and that node labels are from 1 to n (instead of choosing IDs from a more general domain) also strengthens the lower bound.
- Instead of directly proving that 3-coloring a ring needs $\Omega(\log^* n)$ rounds, we will prove a slightly more general statement. We will consider deterministic algorithms with time complexity r (for arbitrary r) and derive a lower bound on the number of colors that are needed if we want to properly color an n-node ring with an r-round algorithm. A 3-coloring lower bound can then be derived by taking the smallest r for which an r-round algorithm needs 3 or fewer colors.

Algorithm 8.1 Synchronous Algorithm: Canonical Form

```
1: In r rounds: send complete initial state to nodes at distance at most r
2:                                        // do all the communication first
3: Compute output based on complete information about r-neighborhood
4:                                        // do all the computation in the end
```

8.2 Locality

Let us for a moment look at distributed algorithms more generally (i.e., not only at coloring and not only at rings). Assume that initially, all nodes only know their own label (identifier) and potentially some additional input. As information needs at least r rounds to travel r hops, after r rounds, a node v can only learn about other nodes at distance at most r. If message size and local computations are not restricted, it is in fact not hard to see, that in r rounds, a node v can exactly learn all the node labels and inputs up to distance r. As shown by the following lemma, this allows to transform every deterministic r-round synchronous algorithm into a simple canonical form.

Lemma 8.2. *If message size and local computations are not bounded, every deterministic, synchronous r-round algorithm can be transformed into an algorithm of the form given by Algorithm 8.1 (i.e., it is possible to first communicate for r rounds and then do all the computations in the end).*

Proof. Consider some r-round algorithm $\mathcal{A}$. We want to show that $\mathcal{A}$ can be brought to the canonical form given by Algorithm 8.1. First, we let the nodes communicate for r rounds. Assume that in every round, every node sends its complete state to all of its neighbors (remember that there is no restriction on the maximal message size). By induction, after i rounds, every node knows the initial state of all other nodes at distance at most i. Hence, after r rounds, a node v has the combined initial knowledge of all the nodes in its r-neighborhood. We want to show that this suffices to locally (at node v) simulate enough of Algorithm $\mathcal{A}$ to compute all the messages that v receives in the r communication rounds of a regular execution of Algorithm $\mathcal{A}$.

Concretely, we prove the following statement by induction on i. For all nodes at distance at most $r - i + 1$ from v, node v can compute all messages of the first i rounds of a regular execution of $\mathcal{A}$. Note that this implies that v can compute all the messages it receives from its neighbors during all r rounds. Because v knows the initial state of all nodes in the r-neighborhood, v can clearly compute all messages of the first round (i.e., the statement is true for $i = 1$). Let us now consider the induction step from i to $i+1$. By the induction hypothesis, v can compute the messages of the first i rounds of all nodes in its $(r - i + 1)$-neighborhood. It can therefore compute all messages that are received by nodes in the $(r - i)$-neighborhood in the first i rounds. This is of

course exactly what is needed to compute the messages of round $i+1$ of nodes in the $(r-i)$-neighborhood. □

Remarks:

- It is straightforward to generalize the canonical form to randomized algorithms: Every node first computes all the random bits it needs throughout the algorithm. The random bits are then part of the initial state of a node.

Definition 8.3 (r-hop view)**.** *We call the collection of the initial states of all nodes in the r-neighborhood of a node v, the r-hop view of v.*

Remarks:

- Assume that initially, every node knows its degree, its label (identifier) and potentially some additional input. The r-hop view of a node v then includes the complete topology of the r-neighborhood (excluding edges between nodes at distance r) and the labels and additional inputs of all nodes in the r-neighborhood.

Based on the definition of an r-hop view, we can state the following corollary of Lemma 8.2.

Corollary 8.4. *A deterministic r-round algorithm $\mathcal{A}$ is a function that maps every possible r-hop view to the set of possible outputs.*

Proof. By Lemma 8.2, we know that we can transform Algorithm $\mathcal{A}$ to the canonical form given by Algorithm 8.1. After r communication rounds, every node v knows exactly its r-hop view. This information suffices to compute the output of node v. □

Remarks:

- Note that the above corollary implies that two nodes with equal r-hop views have to compute the same output in every r-round algorithm.
- For coloring algorithms, the only input of a node v is its label. The r-hop view of a node therefore is its labeled r-neighborhood.
- If we only consider rings, r-hop neighborhoods are particularly simple. The labeled r-neighborhood of a node v (and hence its r-hop view) in an oriented ring is simply a $(2r+1)$-tuple $(\ell_{-r}, \ell_{-r+1}, \ldots, \ell_0, \ldots, \ell_r)$ of distinct node labels where ℓ_0 is the label of v. Assume that for $i > 0$, ℓ_i is the label of the i^{th} clockwise neighbor of v and ℓ_{-i} is the label of the i^{th} counterclockwise neighbor of v. A deterministic coloring algorithm for oriented rings therefore is a function that maps $(2r+1)$-tuples of node labels to colors.
- Consider two r-hop views $\mathcal{V}_r = (\ell_{-r}, \ldots, \ell_r)$ and $\mathcal{V}'_r = (\ell'_{-r}, \ldots, \ell'_r)$. If $\ell'_i = \ell_{i+1}$ for $-r \leq i \leq r-1$ and if $\ell'_r \neq \ell_i$ for $-r \leq i \leq r$, the r-hop view $\mathcal{V}'_r$ can be the r-hop view of a clockwise neighbor of a node with r-hop view $\mathcal{V}_r$. Therefore, every algorithm $\mathcal{A}$ that computes a valid coloring needs to assign different colors to $\mathcal{V}_r$ and $\mathcal{V}'_r$. Otherwise, there is a ring labeling for which $\mathcal{A}$ assigns the same color to two adjacent nodes.

8.3 The Neighborhood Graph

We will now make the above observations concerning colorings of rings a bit more formal. Instead of thinking of an r-round coloring algorithm as a function from all possible r-hop views to colors, we will use a slightly different perspective. Interestingly, the problem of understanding distributed coloring algorithms can itself be seen as a classical graph coloring problem.

Definition 8.5 (Neighborhood Graph)**.** *For a given family of network graphs $\mathcal{G}$, the r-neighborhood graph $\mathcal{N}_r(\mathcal{G})$ is defined as follows. The node set of $\mathcal{N}_r(\mathcal{G})$ is the set of all possible labeled r-neighborhoods (i.e., all possible r-hop views). There is an edge between two labeled r-neighborhoods $\mathcal{V}_r$ and $\mathcal{V}'_r$ if $\mathcal{V}_r$ and $\mathcal{V}'_r$ can be the r-hop views of two adjacent nodes.*

Lemma 8.6. *For a given family of network graphs $\mathcal{G}$, there is an r-round algorithm that colors graphs of $\mathcal{G}$ with c colors iff the chromatic number of the neighborhood graph is $\chi(\mathcal{N}_r(\mathcal{G})) \leq c$.*

Proof. We have seen that a coloring algorithm is a function that maps every possible r-hop view to a color. Hence, a coloring algorithm assigns a color to every node of the neighborhood graph $\mathcal{N}_r(\mathcal{G})$. If two r-hop views $\mathcal{V}_r$ and $\mathcal{V}'_r$ can be the r-hop views of two adjacent nodes u and v (for some labeled graph in $\mathcal{G}$), every correct coloring algorithm must assign different colors to $\mathcal{V}_r$ and $\mathcal{V}'_r$. Thus, specifying an r-round coloring algorithm for a family of network graphs $\mathcal{G}$ is equivalent to coloring the respective neighborhood graph $\mathcal{N}_r(\mathcal{G})$. □

Instead of directly defining the neighborhood graph for directed rings, we define directed graphs $\mathcal{B}_k$ that are closely related to the neighborhood graph. The node set of $\mathcal{B}_k$ contains all k-tuples of increasing node labels ($[n] = \{1, \ldots, n\}$):

$$V[\mathcal{B}_k] \;=\; \big\{(\alpha_1, \ldots, \alpha_k) : \alpha_i \in [n], i < j \to \alpha_i < \alpha_j\big\} \tag{8.1}$$

For $\underline{\alpha} = (\alpha_1, \ldots, \alpha_k)$ and $\underline{\beta} = (\beta_1, \ldots, \beta_k)$ there is a directed edge from $\underline{\alpha}$ to $\underline{\beta}$ iff

$$\forall i \in \{1, \ldots, k-1\} : \beta_i = \alpha_{i+1}. \tag{8.2}$$

Lemma 8.7. *Viewed as an undirected graph, the graph $\mathcal{B}_{2r+1}$ is a subgraph of the r-neighborhood graph of directed n-node rings with node labels from $[n]$.*

Proof. The claim follows directly from the observations regarding r-hop views of nodes in a directed ring from Section 8.2. The set of k-tuples of increasing node labels is a subset of the set of k-tuples of distinct node labels. Two nodes of $\mathcal{B}_{2r+1}$ are connected by a directed edge iff the two corresponding r-hop views are connected by a directed edge in the neighborhood graph. Note that if there is an edge between $\underline{\alpha}$ and $\underline{\beta}$ in $\mathcal{B}_k$, $\alpha_1 \neq \beta_k$ because the node labels in $\underline{\alpha}$ and $\underline{\beta}$ are increasing. □

To determine a lower bound on the number of colors an r-round algorithm needs for directed n-node rings, it therefore suffices to determine a lower bound on the chromatic number of $\mathcal{B}_{2r+1}$. To obtain such a lower bound, we need the following definition.

Definition 8.8 (Diline Graph)**.** *The directed line graph (diline graph) $\mathcal{DL}(G)$ of a directed graph $G = (V, E)$ is defined as follows. The node set of $\mathcal{DL}(G)$ is $V[\mathcal{DL}(G)] = E$. There is a directed edge $\big((w, x), (y, z)\big)$ between $(w, x) \in E$ and $(y, z) \in E$ iff $x = y$, i.e., if the first edge ends where the second one starts.*

Lemma 8.9. *If $n > k$, the graph $\mathcal{B}_{k+1}$ can be defined recursively as follows:*

$$\mathcal{B}_{k+1} = \mathcal{DL}(\mathcal{B}_k).$$

Proof. The edges of $\mathcal{B}_k$ are pairs of k-tuples $\underline{\alpha} = (\alpha_1, \ldots, \alpha_k)$ and $\underline{\beta} = (\beta_1, \ldots, \beta_k)$ that satisfy Conditions (8.1) and (8.2). Because the last $k-1$ labels in $\underline{\alpha}$ are equal to the first $k-1$ labels in $\underline{\beta}$, the pair $(\underline{\alpha}, \underline{\beta})$ can be represented by a $(k+1)$-tuple $\underline{\gamma} = (\gamma_1, \ldots, \gamma_{k+1})$ with $\gamma_1 = \alpha_1$, $\gamma_i = \beta_{i-1} = \alpha_i$ for $2 \leq i \leq k$, and $\gamma_{k+1} = \beta_k$. Because the labels in $\underline{\alpha}$ and the labels in $\underline{\beta}$ are increasing, the labels in $\underline{\gamma}$ are increasing as well. The two graphs $\mathcal{B}_{k+1}$ and $\mathcal{DL}(\mathcal{B}_k)$ therefore have the same node sets. There is an edge between two nodes $(\underline{\alpha}_1, \underline{\beta}_1)$ and $(\underline{\alpha}_2, \underline{\beta}_2)$ of $\mathcal{DL}(\mathcal{B}_k)$ if $\underline{\beta}_1 = \underline{\alpha}_2$. This is equivalent to requiring that the two corresponding $(k+1)$-tuples $\underline{\gamma}_1$ and $\underline{\gamma}_2$ are neighbors in $\mathcal{B}_{k+1}$, i.e., that the last k labels of $\underline{\gamma}_1$ are equal to the first k labels of $\underline{\gamma}_2$. □

The following lemma establishes a useful connection between the chromatic numbers of a directed graph G and its diline graph $\mathcal{DL}(G)$.

Lemma 8.10. *For the chromatic numbers $\chi(G)$ and $\chi(\mathcal{DL}(G))$ of a directed graph G and its diline graph, it holds that*

$$\chi\big(\mathcal{DL}(G)\big) \geq \log_2\big(\chi(G)\big).$$

Proof. Given a c-coloring of $\mathcal{DL}(G)$, we show how to construct a 2^c coloring of G. The claim of the lemma then follows because this implies that $\chi(G) \leq 2^{\chi(\mathcal{DL}(G))}$.

Assume that we are given a c-coloring of $\mathcal{DL}(G)$. A c-coloring of the diline graph $\mathcal{DL}(G)$ can be seen as a coloring of the edges of G such that no two adjacent edges have the same color. For a node v of G, let S_v be the set of colors of its outgoing edges. Let u and v be two nodes such that G contains a directed edge (u, v) from u to v and let x be the color of (u, v). Clearly, $x \in S_u$ because (u, v) is an outgoing edge of u. Because adjacent edges have different colors, no outgoing edge (v, w) of v can have color x. Therefore $x \notin S_v$. This implies that $S_u \neq S_v$. We can therefore use these color sets to obtain a vertex coloring of G, i.e., the color of u is S_u and the color of v is S_v. Because the number of possible subsets of $[c]$ is 2^c, this yields a 2^c-coloring of G. □

Let $\log^{(i)} x$ be the i-fold application of the base-2 logarithm to x:

$$\log^{(1)} x = \log_2 x, \quad \log^{(i+1)} x = \log_2(\log^{(i)} x).$$

Remember from Chapter 1 that

$$\log^* x = 1 \text{ if } x \leq 2, \quad \log^* x = 1 + \min\{i : \log^{(i)} x \leq 2\}.$$

For the chromatic number of $\mathcal{B}_k$, we obtain

Lemma 8.11. *For all $n \geq 1$, $\chi(\mathcal{B}_1) = n$. Further, for $n \geq k \geq 2$, $\chi(\mathcal{B}_k) \geq \log^{(k-1)} n$.*

Proof. For $k = 1$, $\mathcal{B}_k$ is the complete graph on n nodes with a directed edge from node i to node j iff $i < j$. Therefore, $\chi(\mathcal{B}_1) = n$. For $k > 2$, the claim follows by induction and Lemmas 8.9 and 8.10. □

This finally allows us to state a lower bound on the number of rounds needed to color a directed ring with 3 colors.

Theorem 8.12. *Every deterministic, distributed algorithm to color a directed ring with 3 or less colors needs at least $(\log^* n)/2 - 1$ rounds.*

Proof. Using the connection between $\mathcal{B}_k$ and the neighborhood graph for directed rings, it suffices to show that $\chi(\mathcal{B}_{2r+1}) > 3$ for all $r < (\log^* n)/2 - 1$. From Lemma 8.11, we know that $\chi(\mathcal{B}_{2r+1}) \geq \log^{(2r)} n$. To obtain $\log^{(2r)} n \leq 2$, we need $r \geq (\log^* n)/2 - 1$. Because $\log_2 3 < 2$, we therefore have $\log^{(2r)} n > 3$ if $r < \log^* n/2 - 1$. □

Corollary 8.13. *Every deterministic, distributed algorithm to compute an MIS of a directed ring needs at least $\log^* n/2 - O(1)$ rounds.*

Remarks:

- It is straightforward to see that also for a constant $c > 3$, the number of rounds needed to color a ring with c or less colors is $\log^* n/2 - O(1)$.
- There basically (up to additive constants) is a gap of a factor of 2 between the $\log^* n + O(1)$ upper bound of Chapter 1 and the $\log^* n/2 - O(1)$ lower bound of this chapter. It is possible to show that the lower bound is tight, even for undirected rings (for directed rings, this will be part of the exercises).
- Alternatively, the lower bound can also be presented as an application of Ramsey's theory. Ramsey's theory is best introduced with an example: Assume you host a party, and you want to invite people such that there are no three people who mutually know each other, and no three people which are mutual strangers. How many people can you invite? This is an example of Ramsey's theorem, which says that for any given integer c, and any given integers $n_1, \ldots, n_c$, there is a Ramsey number $R(n_1, \ldots, n_c)$, such that if the edges of a complete graph with $R(n_1, \ldots, n_c)$ nodes are colored with c different colors, then for some color i the graph contains some complete subgraph of color i of size n_i. The special case in the party example is looking for $R(3, 3)$.
- Ramsey theory is more general, as it deals with hyperedges. A normal edge is essentially a subset of two nodes; a hyperedge is a subset of k nodes. The party example can be explained in this context: We have (hyper)edges of the form $\{i, j\}$, with $1 \leq i, j \leq n$. Choosing n sufficiently large, coloring the edges with two colors must exhibit a set S of 3 edges $\{i, j\} \subset \{v_1, v_2, v_3\}$, such that all edges in S have the same color. To prove our coloring lower bound using Ramsey theory, we form all hyperedges of size $k = 2r+1$, and color them with 3 colors. Choosing n sufficiently large, there must be a set $S = \{v_1, \ldots, v_{k+1}\}$ of $k+1$ identifiers, such that all $k+1$ hyperedges consisting of k

nodes from S have the same color. Note that both $\{v_1, \ldots, v_k\}$ and $\{v_2, \ldots, v_{k+1}\}$ are in the set S, hence there will be two neighboring views with the same color. Ramsey theory shows that in this case n will grow as a power tower (tetration) in k. Thus, if n is so large that k is smaller than some function growing like $\log^* n$, the coloring algorithm cannot be correct.

- The neighborhood graph concept can be used more generally to study distributed graph coloring. It can for instance be used to show that with a single round (every node sends its identifier to all neighbors) it is possible to color a graph with $(1+o(1))\Delta^2 \ln n$ colors, and that every one-round algorithm needs at least $\Omega(\Delta^2/\log^2 \Delta + \log\log n)$ colors.

- One may also extend the proof to other problems, for instance one may show that a constant approximation of the minimum dominating set problem on unit disk graphs costs at least log-star time.

- Using r-hop views and the fact that nodes with equal r-hop views have to make the same decisions is the basic principle behind almost all locality lower bounds (in fact, we are not aware of a locality lower bound that does not use this principle). Using this basic technique (but a completely different proof otherwise), it is for instance possible to show that computing an MIS (and many other problems) in a general graph requires at least $\Omega(\sqrt{\log n/\log\log n})$ and $\Omega(\log\Delta/\log\log\Delta)$ rounds.

Chapter Notes

The lower bound proof in this chapter is by Linial [Lin92], proving asymptotic optimality of the technique of Chapter 1. This proof can also be found in Chapter 7.5 of [Pel00]. An alternative proof that omits the neighborhood graph construction is presented in [LS14]. The lower bound is also true for randomized algorithms [Nao91]. Recently, this lower bound technique was adapted to other problems [CHW08, LW08]. In some sense, Linial's seminal work raised the question of what can be computed in $\mathcal{O}(1)$ time [NS93], essentially starting distributed complexity theory.

More recently, using a different argument, Kuhn et al. [KMW04, KMW16] managed to show more substantial lower bounds for a number of combinatorial problems including minimum vertex cover (MVC), minimum dominating set (MDS), maximal matching, or maximal independent set (MIS). More concretely, Kuhn et al. showed that all these problems need polylogarithmic time (for a polylogarithmic approximation, in case of approximation problems such as MVC and MDS). Some of these bounds are tight, e.g. the MVC $\Omega(\log\Delta/\log\log\Delta)$ lower bound is surprisingly tight [BYCHS16]. For recent surveys regarding locality lower bounds we refer to e.g. [Suo12, KMW16].

Ramsey theory was started by Frank P. Ramsey with his 1930 article called "On a problem of formal logic" [Ram30]. For an introduction to Ramsey theory we refer to e.g. [NR90, LR03].

Bibliography

[BYCHS16] Reuven Bar-Yehuda, Keren Censor-Hillel, and Gregory Schwartzman. A distributed $(2+\epsilon)$-approximation for vertex cover in $o(\log\delta/\epsilon \log\log \delta)$ rounds. pages 3–8, 2016.

[CHW08] A. Czygrinow, M. Hańćkowiak, and W. Wawrzyniak. Fast Distributed Approximations in Planar Graphs. In *Proceedings of the 22nd International Symposium on Distributed Computing (DISC)*, 2008.

[KMW04] F. Kuhn, T. Moscibroda, and R. Wattenhofer. What Cannot Be Computed Locally! In *Proceedings of the 23rd ACM Symposium on Principles of Distributed Computing (PODC)*, July 2004.

[KMW16] Fabian Kuhn, Thomas Moscibroda, and Roger Wattenhofer. Local Computation: Lower and Upper Bounds. In *Journal of the ACM (JACM)*, 2016.

[Lin92] N. Linial. Locality in Distributed Graph Algorithms. *SIAM Journal on Computing*, 21(1)(1):193–201, February 1992.

[LR03] Bruce M. Landman and Aaron Robertson. *Ramsey Theory on the Integers*. American Mathematical Society, 2003.

[LS14] Juhana Laurinharju and Jukka Suomela. Brief Announcement: Linial's Lower Bound Made Easy. In *Proceedings of the 2014 ACM Symposium on Principles of Distributed Computing*, PODC '14, pages 377–378, New York, NY, USA, 2014. ACM.

[LW08] Christoph Lenzen and Roger Wattenhofer. Leveraging Linial's Locality Limit. In *22nd International Symposium on Distributed Computing (DISC), Arcachon, France*, September 2008.

[Nao91] Moni Naor. A Lower Bound on Probabilistic Algorithms for Distributive Ring Coloring. *SIAM J. Discrete Math.*, 4(3):409–412, 1991.

[NR90] Jaroslav Nesetril and Vojtech Rodl, editors. *Mathematics of Ramsey Theory*. Springer Berlin Heidelberg, 1990.

[NS93] Moni Naor and Larry Stockmeyer. What can be Computed Locally? In *Proceedings of the twenty-fifth annual ACM symposium on Theory of computing*, STOC '93, pages 184–193, New York, NY, USA, 1993. ACM.

[Pel00] David Peleg. *Distributed Computing: a Locality-Sensitive Approach*. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 2000.

[Ram30] F. P. Ramsey. On a Problem of Formal Logic. *Proc. London Math. Soc. (3)*, 30:264–286, 1930.

[Suo12] Jukka Suomela. Survey of Local Algorithms. http://www.cs.helsinki.fi/local-survey/, 2012.

Chapter 9

Social Networks

Distributed computing is applicable in various contexts. This lecture exemplarily studies one of these contexts, social networks, an area of study whose origins date back a century. To give you a first impression, consider Figure 9.1.

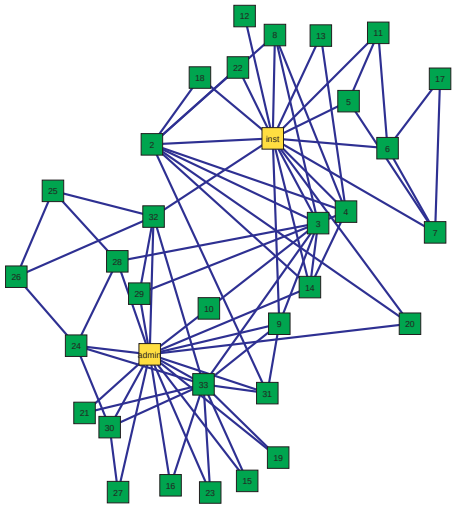


Figure 9.1: This graph shows the social relations between the members of a karate club, studied by anthropologist Wayne Zachary in the 1970s. Two people (nodes) stand out, the instructor and the administrator of the club, both happen to have many friends among club members. At some point, a dispute caused the club to split into two. Can you predict how the club partitioned? (If not, just search the Internet for Zachary and Karate.)

9.1 Small World Networks

Back in 1929, Frigyes Karinthy published a volume of short stories that postulated that the world was "shrinking" because human beings were connected more and more. Some claim that he was inspired by radio network pioneer Guglielmo Marconi's 1909 Nobel Prize speech. Despite physical distance, the growing density of human "networks" renders the actual social distance smaller and smaller. As a result, it is believed that any two individuals can be connected through at most five (or so) acquaintances, i.e., within six hops.

The topic was hot in the 1960s. For instance, in 1964, Marshall McLuhan coined the metaphor "Global Village". He wrote: "As electrically contracted, the globe is no more than a village". He argues that due to the almost instantaneous reaction times of new ("electric") technologies, each individual inevitably feels the consequences of his actions and thus automatically deeply participates in the global society. McLuhan understood what we now can directly observe – real and virtual world are moving together. He realized that the transmission medium, rather than the transmitted information is at the core of change, as expressed by his famous phrase "the medium is the message".

This idea has been followed ardently in the 1960s by several sociologists, first by Michael Gurevich, later by Stanley Milgram. Milgram wanted to know the average path length between two "random" humans, by using various experiments, generally using randomly chosen individuals from the US Midwest as starting points, and a stockbroker living in a suburb of Boston as target. The starting points were given name, address, occupation, plus some personal information about the target. They were asked to send a letter to the target. However, they were not allowed to *directly* send the letter, rather, they had to pass it to somebody they knew on first-name basis and that they thought to have a higher probability to know the target person. This process was repeated, until somebody knew the target person, and could deliver the letter. Shortly after starting the experiment, letters have been received. Most letters were lost during the process, but if they arrived, the average path length was about 5.5. The observation that the entire population is connected by short acquaintance chains got later popularized by the terms "six degrees of separation" and "small world".

Statisticians tried to explain Milgram's experiments, by essentially giving network models that allowed for short diameters, i.e., each node is connected to each other node by only a few hops. Until today there is a thriving research community in statistical physics that tries to understand network properties that allow for "small world" effects.

The world is often fascinated by graphs with a small radius. For example, movie fanatics study the who-acted-with-whom-in-the-same-movie graph. For this graph it has long been believed that the actor Kevin Bacon has a particularly small radius. The number of hops from Bacon even got a name, the Bacon Number. In the meantime, however, it has been shown that there are "better" centers in the Hollywood universe, such as Sean Connery, Christopher Lee, Rod Steiger, Gene Hackman, or Michael Caine. The center of other social networks has also been explored, Paul Erdös for instance is well known in the math community.

One of the keywords in this area are power-law graphs, networks where node degrees are distributed according to a power-law distribution, i.e., the number

of nodes with degree δ is proportional to $\delta^{-\alpha}$, for some $\alpha > 1$. Such power-law graphs have been witnessed in many application areas, apart from social networks also in the web, or in biology or physics.

Obviously, two power-law graphs might look and behave completely differently, even if α and the number of edges is exactly the same.

One well-known model towards this end is the Watts-Strogatz model. Watts and Strogatz argued that social networks should be modeled by a combination of two networks: As the basis we take a network that has a large cluster coefficient ...

Definition 9.2. *The cluster coefficient of a network is defined by the probability that two friends of a node are likely to be friends as well, averaged over all the nodes.*

..., then we augment such a graph with random links, every node for instance points to a constant number of other nodes, chosen uniformly at random. This augmentation represents acquaintances that connect nodes to parts of the network that would otherwise be far away.

Remarks:

- Without further information, knowing the cluster coefficient is of questionable value: Assume we arrange the nodes in a grid. Technically, if we connect each node to its four closest neighbors, the graph has cluster coefficient 0, since there are no triangles; if we instead connect each node with its eight closest neighbors, the cluster coefficient is 3/7. The cluster coefficient is quite different, even though both networks have similar characteristics.

This is interesting, but not enough to really understand what is going on. For Milgram's experiments to work, it is not sufficient to connect the nodes in a certain way. In addition, the nodes *themselves* need to know how to forward a message to one of their neighbors, even though they cannot know whether that neighbor is really closer to the target. In other words, nodes are not just following physical laws, but they make decisions themselves.

Let us consider an artificial network with nodes on a grid topology, plus some additional random links per node. In a quantitative study it was shown that the random links need a specific distance distribution to allow for efficient greedy routing. This distribution marks the sweet spot for any navigable network.

Definition 9.4 (Augmented Grid)**.** *We take $n = m^2$ nodes $(i,j) \in V = \{1,\dots,m\}^2$ that are identified with the lattice points on an $m \times m$ grid. We define the distance between two nodes (i,j) and (k,ℓ) as $d\big((i,j),(k,\ell)\big) = |k-i| + |\ell - j|$ as the distance between them on the $m \times m$ lattice. The network is modeled using a parameter $\alpha \geq 0$. Each node u has a directed edge to every lattice neighbor. These are the* local contacts *of a node. In addition, each node also has an additional random link (the* long-range contact*). For all u and v, the long-range contact of u points to node v with probability proportional to $d(u,v)^{-\alpha}$, i.e., with probability $d(u,v)^{-\alpha} / \sum_{w \in V \setminus \{u\}} d(u,w)^{-\alpha}$. Figure 9.3 illustrates the model.*

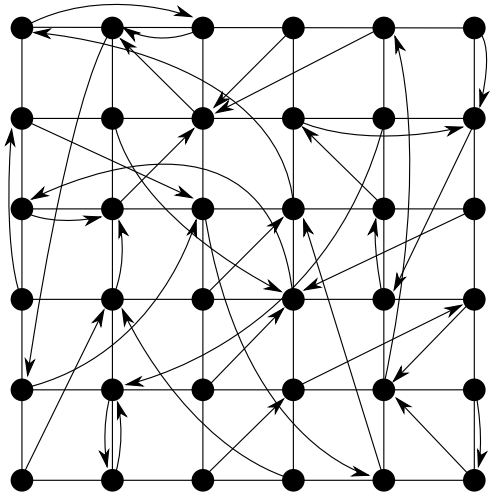


Figure 9.3: Augmented grid with $m = 6$

Remarks:

- The network model has the following geographic interpretation: nodes (individuals) live on a grid and know their neighbors on the grid. Further, each node has some additional acquaintances throughout the network.

- The parameter α controls how the additional neighbors are distributed across the grid. If $\alpha = 0$, long-range contacts are chosen uniformly at random (as in the Watts-Strogatz model). As α increases, long-range contacts become shorter on average. In the extreme case, if $\alpha \to \infty$, all long-range contacts are to immediate neighbors on the grid.

- It can be shown that as long as $\alpha \leq 2$, the diameter of the resulting graph is polylogarithmic in n (polynomial in $\log n$) with high probability. In particular, if the long-range contacts are chosen uniformly at random ($\alpha = 0$), the diameter is $\mathcal{O}(\log n)$.

Since the augmented grid contains random links, we do not know anything for sure about how the random links are distributed. In theory, all links could point to the same node! However, this is almost certainly not the case. Formally this is captured by the term *with high probability*.

Definition 9.5 (With High Probability)**.** *Some probabilistic event is said to occur* with high probability (w.h.p.)*, if it happens with a probability $p \geq 1 -$*

$1/n^c$, where c is a constant. The constant c may be chosen arbitrarily, but it is considered constant with respect to Big-O notation.

Remarks:

- For instance, a running time bound of $c \log n$ or $e^{c!} \log n + 5000c$ with probability at least $1 - 1/n^c$ would be $\mathcal{O}(\log n)$ w.h.p., but a running time of n^c would not be $\mathcal{O}(n)$ w.h.p. since c might also be 50.
- This definition is very powerful, as any polynomial (in n) number of statements that hold w.h.p. also holds w.h.p. at the same time, regardless of any dependencies between random variables!

Theorem 9.6. *The diameter of the augmented grid with $\alpha = 0$ is $\mathcal{O}(\log n)$ with high probability.*

Proof Sketch. For simplicity, we will only show that we can reach a target node t starting from some source node s. However, it can be shown that (essentially) each of the intermediate claims holds with high probability, which then by means of the union bound yields that *all* of the claims hold simultaneously with high probability for *all* pairs of nodes (see exercises).

Let N_s be the $\lceil \log n \rceil$-hop neighborhood of source s on the grid, containing $\Omega(\log^2 n)$ nodes. Each of the nodes in N_s has a random link, probably leading to distant parts of the graph. As long as we have reached only $o(n)$ nodes, any new random link will with probability $1 - o(1)$ lead to a node for which none of its grid neighbors has been visited yet. Thus, in expectation we find almost $|N_s|$ new nodes whose neighbors are "fresh". Using their grid links, we will reach $(4 - o(1))|N_s|$ more nodes within one more hop. If bad luck strikes, it could still happen that many of these links lead to a few nodes, already visited nodes, or nodes that are very close to each other. But that is very unlikely, as we have lots of random choices! Indeed, it can be shown that not only in expectation, but with high probability $(5 - o(1))|N_s|$ many nodes are reached this way (see exercises).

Because all the new nodes have (so far unused) random links, we can repeat this reasoning inductively, implying that the number of nodes grows by (at least) a constant factor for every two hops. Thus, after $\mathcal{O}(\log n)$ hops, we will have reached $n/\log n$ nodes (which is still small compared to n). Finally, consider the expected number of links from these nodes that enter the $(\log n)$-neighborhood of some target node t with respect to the grid. Since this neighborhood consists of $\Omega(\log^2 n)$ nodes, in expectation $\Omega(\log n)$ links come close enough to target t. This is large enough to almost guarantee that this happens (see exercises). Summing everything up, we still used merely $\mathcal{O}(\log n)$ hops in total to get from s to t.

□

This shows that for $\alpha = 0$ (and in fact for all $\alpha \leq 2$), the resulting network has a small diameter. Recall however that we also wanted the network to be navigable. For this, we consider a simple greedy routing strategy (Algorithm 9.7).

Lemma 9.8. *In the augmented grid, Algorithm 9.7 finds a routing path of length at most $2(m-1) \in O(\sqrt{n})$.*

Algorithm 9.7 Greedy Routing

```
1: while not at destination do
2:     go to a neighbor which is closest to destination (considering grid distance only)
3: end while
```

Proof. Because of the grid, there is always a neighbor which is closer to the destination. Since with each hop we reduce the distance to the target at least by one in one of the two grid dimensions, we will reach the destination within $2(m-1)$ steps. □

This is not really what Milgram's experiment promises. We want to know how much the additional random links speed up the process. To this end, we first need to understand how likely it is that the random link of node u points to node v, in terms of their grid distance $d(u,v)$, the number of nodes n, and the constant parameter α.

Lemma 9.9. *Node u's random link points to a node v with probability*

- $\Theta(1/(d(u,v)^\alpha m^{2-\alpha}))$ *if* $\alpha < 2$.
- $\Theta(1/(d(u,v)^2 \log n))$ *if* $\alpha = 2$,
- $\Theta(1/d(u,v)^\alpha)$ *if* $\alpha > 2$.

Moreover, if $\alpha > 2$, the probability to see a link of length at least d is in $\Theta(1/d^{\alpha-2})$.

Proof. For a constant $\alpha \neq 2$, we have that

$$\sum_{w \in V \setminus \{u\}} \frac{1}{d(u,w)^\alpha} \in \sum_{r=1}^{m} \frac{\Theta(r)}{r^\alpha} = \Theta\left(\int_{r=1}^{m} \frac{1}{r^{\alpha-1}}\, dr\right) = \Theta\left(\left[\frac{r^{2-\alpha}}{2-\alpha}\right]_1^m\right).$$

If $\alpha < 2$, this gives $\Theta(m^{2-\alpha})$, if $\alpha > 2$, it is in $\Theta(1)$. If $\alpha = 2$, we get

$$\sum_{w \in V \setminus \{u\}} \frac{1}{d(u,w)^\alpha} \in \sum_{r=1}^{m} \frac{\Theta(r)}{r^2} = \Theta(1) \cdot \sum_{r=1}^{m} \frac{1}{r} = \Theta(\log m) = \Theta(\log n).$$

Multiplying with $d(u,v)^\alpha$ yields the first three bounds. For the last statement, compute

$$\sum_{\substack{v \in V \\ d(u,v) \geq d}} \Theta(1/d(u,v)^\alpha) = \Theta\left(\int_{r=d}^{m} \frac{r}{r^\alpha}\, dr\right) = \Theta\left(\left[\frac{r^{2-\alpha}}{2-\alpha}\right]_d^m\right) = \Theta(1/d^{\alpha-2}).$$

□

Remarks:

- If $\alpha > 2$, according to the lemma, the probability to see a random link of length at least $d = m^{1/(\alpha-1)}$ is $\Theta(1/d^{\alpha-2}) = \Theta(1/m^{(\alpha-2)/(\alpha-1)})$. In expectation we have to take $\Theta(m^{(\alpha-2)/(\alpha-1)})$ hops until we see a random link of length at least d. When just following links of length less than d, it takes more than $m/d = m/m^{1/(\alpha-1)} = m^{(\alpha-2)/(\alpha-1)}$ hops. In other words, in expectation, either way we need at least $m^{(\alpha-2)/(\alpha-1)} = m^{\Omega(1)}$ hops to the destination.
- If $\alpha < 2$, there is a (slightly more complicated) argument. First we draw a border around the nodes in distance $m^{(2-\alpha)/3}$ to the target. Within this border there are about $m^{2(2-\alpha)/3}$ many nodes in the target area. Assume that the source is outside the target area. Starting at the source, the probability to find a random link that leads directly inside the target area is according to the lemma at most $m^{2(2-\alpha)/3} \cdot \Theta(1/m^{2-\alpha})) = \Theta(1/m^{(2-\alpha)/3})$. In other words, until we find a random link that leads into the target area, in expectation, we have to do $\Theta(m^{(2-\alpha)/3})$ hops. This is too slow, and our greedy strategy is probably faster, as thanks to having $\alpha < 2$ there are many long-range links. However, it means that we will probably enter the border of the target area on a regular grid link. Once inside the target area, again the probability of short-cutting our trip by a random long-range link is $\Theta(1/m^{(2-\alpha)/3})$, so we probably just follow grid links, $m^{(2-\alpha)/3} = m^{\Omega(1)}$ many of them.
- In summary, if $\alpha \neq 2$, our greedy routing algorithm takes $m^{\Omega(1)} = n^{\Omega(1)}$ expected hops to reach the destination. This is polynomial in the number of nodes n, and the social network can hardly be called a "small world".
- Maybe we can get a polylogarithmic bound on n if we set $\alpha = 2$?

Definition 9.10 (Phase)**.** *Consider routing from source s to target t and assume that we are at some intermediate node w. We say that we are in phase j at node w if the lattice distance $d(w,t)$ to the target node t is between $2^j < d(w,t) \leq 2^{j+1}$.*

Remarks:

- Enumerating the phases in decreasing order is useful, as notation becomes less cumbersome.
- There are $\lceil \log m \rceil \in O(\log n)$ phases.

Lemma 9.11. *Assume that we are in phase j at node w when routing from s to t. The probability for getting (at least) to phase $j-1$ in one step is at least $\Omega(1/\log n)$.*

Proof. Let B_j be the set of nodes x with $d(x,t) \leq 2^j$. We get from phase j to (at least) phase $j-1$ if the long-range contact of node w points to some node in B_j. Note that we always make progress while following the greedy routing path. Therefore, we have not seen node w before and the long-range contact of

w points to a random node that is independent of anything seen on the path from s to w.

For all nodes $x \in B_j$, we have $d(w,x) \leq d(w,t) + d(x,t) \leq 2^{j+1} + 2^j < 2^{j+2}$. Hence, for each node $x \in B_j$, the probability that the long-range contact of w points to x is $\Omega(1/2^{2j+4} \log n)$. Further, the number of nodes in B_j is at least $(2^j)^2/2 = 2^{2j-1}$. Hence, the probability that some node in B_j is the long range contact of w is at least

$$\Omega\left(|B_j| \cdot \frac{1}{2^{2j+4}\log n}\right) = \Omega\left(\frac{2^{2j-1}}{2^{2j+4}\log n}\right) = \Omega\left(\frac{1}{\log n}\right). \qquad \square$$

Theorem 9.12. *Consider the greedy routing path from a node s to a node t on an augmented grid with parameter $\alpha = 2$. The expected length of the path is $\mathcal{O}(\log^2 n)$.*

Proof. We already observed that the total number of phases is $\mathcal{O}(\log n)$ (the distance to the target is halved when we go from phase j to phase $j-1$). At each point during the routing process, the probability of proceeding to the next phase is at least $\Omega(1/\log n)$. Let X_j be the number of steps in phase j. Because the probability for ending the phase is $\Omega(1/\log n)$ in each step, in expectation we need $\mathcal{O}(\log n)$ steps to proceed to the next phase, i.e., $\mathbb{E}[X_j] \in O(\log n)$. Let $X = \sum_j X_j$ be the total number of steps of the routing process. By linearity of expectation, we have

$$\mathbb{E}[X] = \sum_j \mathbb{E}[X_j] \in O(\log^2 n). \qquad \square$$

Remarks:

- One can show that the $\mathcal{O}(\log^2 n)$ result also holds w.h.p.
- In real world social networks, the parameter α was evaluated experimentally. The assumption is that you are connected to the geographically closest nodes, and then have some random long-range contacts. For Facebook grandpa LiveJournal it was shown that α is not really 2, but rather around 1.25.

9.2 Propagation Studies

In networks, nodes may influence each other's behavior and decisions. There are many applications where nodes influence their neighbors, e.g., they may impact their opinions, or they may bias what products they buy, or they may pass on a disease.

On a beach (modeled as a line segment), it is best to place an ice cream stand right in the middle of the segment, because you will be able to "control" the beach most easily. What about the second stand, where should it settle? The answer generally depends on the model, but assuming that people will buy ice cream from the stand that is closer, it should go right next to the first stand.

Rumors can spread surprisingly fast through social networks. Traditionally this happens by word of mouth, but with the emergence of the Internet and its possibilities new ways of rumor propagation are available. People write

email, use instant messengers or publish their thoughts in a blog. Many factors influence the dissemination of rumors. It is especially important where in a network a rumor is initiated and how convincing it is. Furthermore the underlying network structure decides how fast the information can spread and how many people are reached. More generally, we can speak of diffusion of information in networks. The analysis of these diffusion processes can be useful for viral marketing, e.g., to target a few influential people to initiate marketing campaigns. A company may wish to distribute the rumor of a new product via the most influential individuals in popular social networks such as Facebook. A second company might want to introduce a competing product and has hence to select where to seed the information to be disseminated. Rumor spreading is quite similar to our ice cream stand problem.

More formally, we may study propagation problems in graphs. Given a graph, and two players. Let the first player choose a seed node u_1; afterwards let the second player choose a seed node u_2, with $u_2 \neq u_1$. The goal of the game is to maximize the number of nodes that are closer to one's own seed node.

In many graphs it is an advantage to choose first. In a star graph for instance the first player can choose the center node of the star, controlling all but one node. In some other graphs, the second player can at least score even. But is there a graph where the second player has an advantage?

Theorem 9.13. *In a two player rumor game where both players select one node to initiate their rumor in the graph, the first player does not always win.*

Proof. See Figure 9.14 for an example where the second player will always win, regardless of the decision the first player. If the first player chooses the node x_0 in the center, the second player can select x_1. Choice x_1 will be outwitted by x_2, and x_2 itself can be answered by z_1. All other strategies are either symmetric, or even less promising for the first player. □

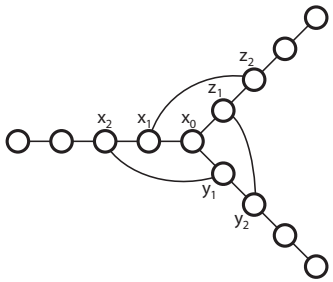


Figure 9.14: Counter example.

Chapter Notes

A simple form of a social network is the famous stable marriage problem [DS62] in which a stable matching bipartite graph has to be found. There exists a great many of variations which are based on this initial problem, e.g., [KC82, KMV94, EO06, FKPS10, Hoe11]. Social networks like Facebook, Twitter and others have grown very fast in the last years and hence spurred interest to research them. How users influence other users has been studied both from a theoretical point of view [KKT03] and in practice [CHBG10]. The structure of these networks can be measured and studied [MMG+07]. More than half of the users in social networks share more information than they expect to [LGKM11].

The small world phenomenon that we presented in this chapter is analyzed by Kleinberg [Kle00]. A general overview is in [DJ10].

This chapter has been written in collaboration with Michael Kuhn.

Bibliography

[CHBG10] Meeyoung Cha, Hamed Haddadi, Fabrício Benevenuto, and P. Krishna Gummadi. Measuring User Influence in Twitter: The Million Follower Fallacy. In *ICWSM*, 2010.

[DJ10] Easley David and Kleinberg Jon. *Networks, Crowds, and Markets: Reasoning About a Highly Connected World*. Cambridge University Press, New York, NY, USA, 2010.

[DS62] D. Gale and L.S. Shapley. College Admission and the Stability of Marriage. *American Mathematical Monthly*, 69(1):9–15, 1962.

[EO06] Federico Echenique and Jorge Oviedo. A theory of stability in many-to-many matching markets. *Theoretical Economics*, 1(2):233–273, 2006.

[FKPS10] Patrik Floréen, Petteri Kaski, Valentin Polishchuk, and Jukka Suomela. Almost Stable Matchings by Truncating the Gale-Shapley Algorithm. *Algorithmica*, 58(1):102–118, 2010.

[Hoe11] Martin Hoefer. Local Matching Dynamics in Social Networks. *Automata Languages and Programming*, pages 113–124, 2011.

[Kar29] Frigyes Karinthy. Chain-Links, 1929.

[KC82] Alexander S. Kelso and Vincent P. Crawford. Job Matching, Coalition Formation, and Gross Substitutes. *Econometrica*, 50(6):1483–1504, 1982.

[KKT03] David Kempe, Jon M. Kleinberg, and Éva Tardos. Maximizing the spread of influence through a social network. In *KDD*, 2003.

[Kle00] Jon M. Kleinberg. The small-world phenomenon: an algorithm perspective. In *STOC*, 2000.

[KMV94] Samir Khuller, Stephen G. Mitchell, and Vijay V. Vazirani. On-line algorithms for weighted bipartite matching and stable marriages. *Theoretical Computer Science*, 127:255–267, May 1994.

[LGKM11] Yabing Liu, Krishna P. Gummadi, Balanchander Krishnamurthy, and Alan Mislove. Analyzing Facebook privacy settings: User expectations vs. reality. In *Proceedings of the 11th ACM/USENIX Internet Measurement Conference (IMC'11)*, Berlin, Germany, November 2011.

[McL64] Marshall McLuhan. *Understanding media: The extensions of man*. McGraw-Hill, New York, 1964.

[Mil67] Stanley Milgram. The Small World Problem. *Psychology Today*, 2:60–67, 1967.

[MMG+07] Alan Mislove, Massimiliano Marcon, P. Krishna Gummadi, Peter Druschel, and Bobby Bhattacharjee. Measurement and analysis of online social networks. In *Internet Measurement Comference*, 2007.

[WS98] Duncan J. Watts and Steven H. Strogatz. Collective dynamics of "small-world" networks. *Nature*, 393(6684):440–442, Jun 1998.

[Zac77] W W Zachary. An information flow model for conflict and fission in small groups. *Journal of Anthropological Research*, 33(4):452–473, 1977.

Chapter 10

Synchronization

So far, we have mainly studied synchronous algorithms. Generally, asynchronous algorithms are more difficult to obtain. Also it is substantially harder to reason about asynchronous algorithms than about synchronous ones. For instance, computing a BFS tree (Chapter 2) efficiently requires much more work in an asynchronous system. However, many real systems are not synchronous, and we therefore have to design asynchronous algorithms. In this chapter, we will look at general simulation techniques, called *synchronizers*, that allow running synchronous algorithms in asynchronous environments.

10.1 Basics

A synchronizer generates sequences of *clock pulses* at each node of the network satisfying the condition given by the following definition.

Definition 10.1 (valid clock pulse)**.** *We call a clock pulse generated at a node v* valid *if it is generated after v received all the messages of the synchronous algorithm sent to v by its neighbors in the previous pulses.*

Given a mechanism that generates the clock pulses, a synchronous algorithm is turned into an asynchronous algorithm in an obvious way: As soon as the i^{th} clock pulse is generated at node v, v performs all the actions (local computations and sending of messages) of round i of the synchronous algorithm.

Theorem 10.2. *If all generated clock pulses are valid according to Definition 10.1, the above method provides an asynchronous algorithm that behaves exactly the same way as the given synchronous algorithm.*

Proof. When the i^{th} pulse is generated at a node v, v has sent and received exactly the same messages and performed the same local computations as in the first $i-1$ rounds of the synchronous algorithm. □

The main problem when generating the clock pulses at a node v is that v cannot know what messages its neighbors are sending to it in a given synchronous round. Because there are no bounds on link delays, v cannot simply wait "long enough" before generating the next pulse. In order satisfy Definition 10.1, nodes have to send additional messages for the purpose of synchronization. The total

complexity of the resulting asynchronous algorithm depends on the overhead introduced by the synchronizer. For a synchronizer $\mathcal{S}$, let $T(\mathcal{S})$ and $M(\mathcal{S})$ be the time and message complexities of $\mathcal{S}$ for each generated clock pulse. As we will see, some of the synchronizers need an initialization phase. We denote the time and message complexities of the initialization by $T_{\text{init}}(\mathcal{S})$ and $M_{\text{init}}(\mathcal{S})$, respectively. If $T(\mathcal{A})$ and $M(\mathcal{A})$ are the time and message complexities of the given synchronous algorithm $\mathcal{A}$, the total time and message complexities T_{tot} and M_{tot} of the resulting asynchronous algorithm then become

$$T_{tot} = T_{\text{init}}(\mathcal{S}) + T(\mathcal{A}) \cdot (1 + T(\mathcal{S})) \text{ and } M_{tot} = M_{\text{init}}(\mathcal{S}) + M(\mathcal{A}) + T(\mathcal{A}) \cdot M(\mathcal{S}),$$

respectively.

Remarks:

- Because the initialization only needs to be done once for each network, we will mostly be interested in the overheads $T(\mathcal{S})$ and $M(\mathcal{S})$ per round of the synchronous algorithm.

Definition 10.3 (Safe Node)**.** *A node v is* safe *with respect to a certain clock pulse if all messages of the synchronous algorithm sent by v in that pulse have already arrived at their destinations.*

Lemma 10.4. *If all neighbors of a node v are safe with respect to the current clock pulse of v, the next pulse can be generated for v.*

Proof. If all neighbors of v are safe with respect to a certain pulse, v has received all messages of the given pulse. Node v therefore satisfies the condition of Definition 10.1 for generating a valid next pulse. □

Remarks:

- In order to detect safety, we require that all algorithms send acknowledgements for all received messages. As soon as a node v has received an acknowledgement for each message that it has sent in a certain pulse, it knows that it is safe with respect to that pulse. Note that sending acknowledgements does not increase the asymptotic time and message complexities.

10.2 The Local Synchronizer α

Algorithm 10.5 Synchronizer α (at node v)

```
1: wait until v is safe
2: send SAFE to all neighbors
3: wait until v receives SAFE messages from all neighbors
4: start new pulse
```

Synchronizer α is very simple. It does not need an initialization. Using acknowledgements, each node eventually detects that it is safe. It then reports this fact directly to all its neighbors. Whenever a node learns that all its neighbors are safe, a new pulse is generated. Algorithm 10.5 formally describes the synchronizer α.

Theorem 10.6. *The time and message complexities of synchronizer α per synchronous round are*

$$T(\alpha) = O(1) \quad \textit{and} \quad M(\alpha) = O(m).$$

Proof. Communication is only between neighbors. As soon as all neighbors of a node v become safe, v knows of this fact after one additional time unit. For every clock pulse, synchronizer α sends at most four additional messages over every edge: Each of the nodes may have to acknowledge a message and reports safety. □

Remarks:

- Synchronizer α was presented in a framework, mostly set up to have a common standard to discuss different synchronizers. Without the framework, synchronizer α can be explained more easily:
  1. Send message to all neighbors, include round information i and actual data of round i (if any).
  2. Wait for message of round i from all neighbors, and go to next round.
- Although synchronizer α allows for simple and fast synchronization, it produces awfully many messages. Can we do better? Yes.

10.3 The Global Synchronizer β

Algorithm 10.7 Synchronizer β (at node v)

```
1: wait until v is safe
2: wait until v receives SAFE messages from all its children in T
3: if v ≠ ℓ then
4:    send SAFE message to parent in T
5:    wait until PULSE message received from parent in T
6: end if
7: send PULSE message to children in T
8: start new pulse
```

Synchronizer β needs an initialization that computes a leader node ℓ and a spanning tree T rooted at ℓ. As soon as all nodes are safe, this information is propagated to ℓ by a convergecast. The leader then broadcasts this information to all nodes. The details of synchronizer β are given in Algorithm 10.7.

Theorem 10.8. *The time and message complexities of synchronizer β per synchronous round are*

$$T(\beta) = O(\text{diameter}(T)) \leq O(n) \quad \textit{and} \quad M(\beta) = O(n).$$

The time and message complexities for the initialization are

$$T_{\text{init}}(\beta) = O(n) \quad \textit{and} \quad M_{\text{init}}(\beta) = O(m + n \log n).$$

Proof. Because the diameter of T is at most $n-1$, the convergecast and the broadcast together take at most $2n-2$ time units. Per clock pulse, the synchronizer sends at most $2n-2$ synchronization messages (one in each direction over each edge of T).

With the improved variant of the GHS algorithm (Algorithm 2.18) mentioned in Chapter 2, it is possible to construct an MST in time $\mathcal{O}(n)$ with $\mathcal{O}(m + n \log n)$ messages in an asynchronous environment. Once the tree is computed, the tree can be made rooted in time $\mathcal{O}(n)$ with $\mathcal{O}(n)$ messages. □

Remarks:

- We now got a time-efficient synchronizer (α) and a message-efficient synchronizer (β), it is only natural to ask whether we can have the best of both worlds. And, indeed, we can. How is that synchronizer called? Quite obviously: γ.

10.4 The Hybrid Synchronizer γ

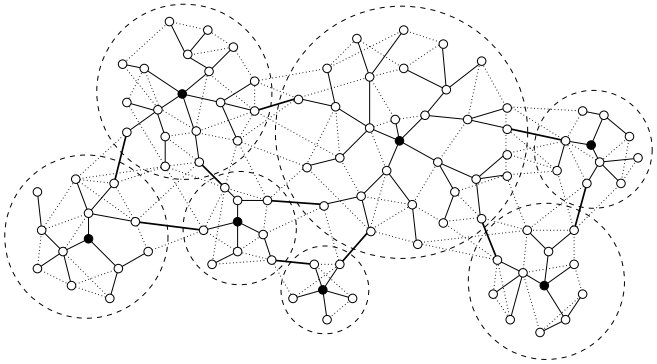


Figure 10.9: A cluster partition of a network: The dashed cycles specify the clusters, cluster leaders are black, the solid edges are the edges of the intracluster trees, and the bold solid edges are the intercluster edges

Synchronizer γ can be seen as a combination of synchronizers α and β. In the initialization phase, the network is partitioned into clusters of small diameter. In each cluster, a leader node is chosen and a BFS tree rooted at this leader node is computed. These trees are called the *intracluster trees*. Two clusters C_1 and C_2 are called neighboring if there are nodes $u \in C_1$ and $v \in C_2$ for which $(u, v) \in E$. For every two neighboring clusters, an *intercluster edge* is chosen, which will serve for communication between these clusters. Figure 10.9 illustrates this partitioning into clusters. We will discuss the details of how to construct such a partition in the next section. We say that a cluster is safe if all its nodes are safe.

Synchronizer γ works in two phases. In a first phase, synchronizer β is applied separately in each cluster by using the intracluster trees. Whenever the leader of a cluster learns that its cluster is safe, it reports this fact to all the nodes in the clusters as well as to the leaders of the neighboring clusters. Now, the nodes of the cluster enter the second phase where they wait until all the neighboring clusters are known to be safe and then generate the next pulse. Hence, we essentially apply synchronizer α between clusters. A detailed description is given by Algorithm 10.10.

Algorithm 10.10 Synchronizer γ (at node v)

```
1: wait until v is safe
2: wait until v receives SAFE messages from all children in intracluster tree
3: if v is not cluster leader then
4:    send SAFE message to parent in intracluster tree
5:    wait until CLUSTERSAFE message received from parent
6: end if
7: send CLUSTERSAFE message to all children in intracluster tree
8: send NEIGHBORSAFE message over all intercluster edges of v
9: wait until v receives NEIGHBORSAFE messages from all adjacent inter-
   cluster edges and all children in intracluster tree
10: if v is not cluster leader then
11:    send NEIGHBORSAFE message to parent in intracluster tree
12:    wait until PULSE message received from parent
13: end if
14: send PULSE message to children in intracluster tree
15: start new pulse
```

Theorem 10.11. *Let m_C be the number of intercluster edges and let k be the maximum cluster radius (i.e., the maximum distance of a leaf to its cluster leader). The time and message complexities of synchronizer γ are*

$$T(\gamma) = O(k) \quad \text{and} \quad M(\gamma) = O(n + m_C).$$

Proof. We ignore acknowledgements, as they do not affect the asymptotic complexities. Let us first look at the number of messages. Over every intracluster tree edge, exactly one SAFE message, one CLUSTERSAFE message, one NEIGHBORSAFE message, and one PULSE message is sent. Further, one NEIGHBORSAFE message is sent over every intercluster edge. Because there are less than n intracluster tree edges, the total message complexity therefore is at most $4n + 2m_C = O(n + m_C)$.

For the time complexity, note that the depth of each intracluster tree is at most k. On each intracluster tree, two convergecasts (the SAFE and NEIGHBORSAFE messages) and two broadcasts (the CLUSTERSAFE and PULSE messages) are performed. The time complexity for this is at most $4k$. There is one more time unit needed to send the NEIGHBORSAFE messages over the intercluster edges. The total time complexity therefore is at most $4k + 1 = O(k)$. □

10.5 Network Partition

We will now look at the initialization phase of synchronizer γ. Algorithm 10.12 describes how to construct a partition into clusters that can be used for synchronizer γ. In Algorithm 10.12, $B(v, r)$ denotes the ball of radius r around v, i.e., $B(v, r) = \{u \in V : d(u, v) \leq r\}$ where $d(u, v)$ is the hop distance between u and v. The algorithm has a parameter $\rho > 1$. The clusters are constructed sequentially. Each cluster is started at an arbitrary node that has not been included in a cluster. Then the cluster radius is grown as long as the cluster grows by a factor more than ρ.

Algorithm 10.12 Cluster construction

```
1: while unprocessed nodes do
2:    select an arbitrary unprocessed node v;
3:    r := 0;
4:    while |B(v, r + 1)| > ρ|B(v, r)| do
5:       r := r + 1
6:    end while
7:    makeCluster(B(v, r))          // all nodes in B(v, r) are now processed
8: end while
```

Remarks:

- The algorithm allows a trade-off between the cluster diameter k (and thus the time complexity) and the number of intercluster edges m_C (and thus the message complexity). We will quantify the possibilities in the next section.
- Two very simple partitions would be to make a cluster out of every single node or to make one big cluster that contains the whole graph. We then get synchronizers α and β as special cases of synchronizer γ.

Theorem 10.13. *Algorithm 10.12 computes a partition of the network graph into clusters of radius at most $\log_\rho n$. The number of intercluster edges is at most $(\rho - 1) \cdot n$.*

Proof. The radius of a cluster is initially 0 and does only grow as long as it grows by a factor larger than ρ. Since there are only n nodes in the graph, this can happen at most $\log_\rho n$ times.

To count the number of intercluster edges, observe that an edge can only become an intercluster edge if it connects a node at the boundary of a cluster with a node outside a cluster. Consider a cluster C of size $|C|$. We know that $C = B(v, r)$ for some $v \in V$ and $r \geq 0$. Further, we know that $|B(v, r + 1)| \leq \rho \cdot |B(v, r)|$. The number of nodes adjacent to cluster C is therefore at most $|B(v, r + 1) \setminus B(v, r)| \leq \rho \cdot |C| - |C|$. Because there is only one intercluster edge connecting two clusters by definition, the number of intercluster edges adjacent to C is at most $(\rho - 1) \cdot |C|$. Summing over all clusters, we get that the total number of intercluster edges is at most $(\rho - 1) \cdot n$. □

Corollary 10.14. *Using $\rho = 2$, Algorithm 10.12 computes a clustering with cluster radius at most $\log_2 n$ and with at most n intercluster edges.*

Corollary 10.15. *Using $\rho = n^{1/k}$, Algorithm 10.12 computes a clustering with cluster radius at most k and at most $\mathcal{O}(n^{1+1/k})$ intercluster edges.*

Remarks:

- Algorithm 10.12 describes a centralized construction of the partitioning of the graph. For $\rho \geq 2$, the clustering can be computed by an asynchronous distributed algorithm in time $\mathcal{O}(n)$ with $\mathcal{O}(m + n \log n)$ (reasonably sized) messages (showing this will be part of the exercises).
- It can be shown that the trade-off between cluster radius and number of intercluster edges of Algorithm 10.12 is asymptotically optimal. There are graphs for which every clustering into clusters of radius at most k requires $n^{1+c/k}$ intercluster edges for some constant c.

The above remarks lead to a complete characterization of the complexity of synchronizer γ.

Corollary 10.16. *The time and message complexities of synchronizer γ per synchronous round are*

$$T(\gamma) \;=\; O(k) \quad \textit{and} \quad M(\gamma) \;=\; O(n^{1+1/k}).$$

The time and message complexities for the initialization are

$$T_{\text{init}}(\gamma) \;=\; O(n) \quad \textit{and} \quad M_{\text{init}}(\gamma) \;=\; O(m + n \log n).$$

Remarks:

- In Chapter 2, you have seen that by using flooding, there is a very simple synchronous algorithm to compute a BFS tree in time $\mathcal{O}(D)$ with message complexity $\mathcal{O}(m)$. If we use synchronizer γ to make this algorithm asynchronous, we get an algorithm with time complexity $\mathcal{O}(n+D \log n)$ and message complexity $\mathcal{O}(m+n \log n+D \cdot n)$ (including initialization).
- The synchronizers α, β, and γ achieve global synchronization, i.e. every node generates every clock pulse. The disadvantage of this is that nodes that do not participate in a computation also have to participate in the synchronization. In many computations (e.g. in a BFS construction), many nodes only participate for a few synchronous rounds. In such scenarios, it is possible to achieve time and message complexity $\mathcal{O}(\log^3 n)$ per synchronous round (without initialization).
- It can be shown that if all nodes in the network need to generate all pulses, the trade-off of synchronizer γ is asymptotically optimal.
- Partitions of networks into clusters of small diameter and coverings of networks with clusters of small diameters come in many variations and have various applications in distributed computations. In particular, apart from synchronizers, algorithms for routing, the construction of sparse spanning subgraphs, distributed data structures, and even computations of local structures such as a MIS or a dominating set are based on some kind of network partitions or covers.

10.6 Clock Synchronization

"A man with one clock knows what time it is – a man with two is never sure."

Synchronizers can directly be used to give nodes in an asynchronous network a common notion of time. In wireless networks, for instance, many basic protocols need an accurate time. Sometimes a common time in the whole network is needed, often it is enough to synchronize neighbors. The purpose of the time division multiple access (TDMA) protocol is to use the common wireless channel as efficiently as possible, i.e., interfering nodes should never transmit at the same time (on the same frequency). If we use synchronizer β to give the nodes a common notion of time, every single clock cycle costs D time units!

Often, each (wireless) node is equipped with an internal clock. Using this clock, it should be possible to divide time into slots, and make each node send (or listen, or sleep, respectively) in the appropriate slots according to the media access control (MAC) layer protocol used.

However, as it turns out, synchronizing clocks in a network is not trivial. As nodes' internal clocks are not perfect, they will run at speeds that are time-dependent. For instance, variations in temperature or supply voltage will affect this *clock drift*. For standard clocks, the drift is in the order of parts per million, i.e., within a second, it will accumulate to a couple of microseconds. Wireless TDMA protocols account for this by introducing *guard times*. Whenever a node knows that it is about to receive a message from a neighbor, it powers up its radio a little bit earlier to make sure that it does not miss the message even when clocks are not perfectly synchronized. If nodes are badly synchronized, messages of different slots might collide.

In the *clock synchronization* problem, we are given a network (graph) with n nodes. The goal for each node is to have a logical clock such that the logical clock values are well synchronized, and close to real time. Each node is equipped with a hardware clock, that ticks more or less in real time, i.e., the time between two pulses is arbitrary between $[1-\epsilon, 1+\epsilon]$, for a constant $\epsilon \ll 1$. Similarly as in our asynchronous model, we assume that messages sent over the edges of the graph have a delivery time between $[0, 1]$. In other words, we have a bounded but variable drift on the hardware clocks and an arbitrary jitter in the delivery times. The goal is to design a message-passing algorithm that ensures that the logical clock skew of adjacent nodes is as small as possible at all times.

Theorem 10.17. *The global clock skew (the logical clock difference between any two nodes in the graph) is $\Omega(D)$, where D is the diameter of the graph.*

Proof. For a node u, let t_u be the logical time of u and let $(u \to v)$ denote a message sent from u to a node v. Let $t(m)$ be the time delay of a message m and let u and v be neighboring nodes. First consider a case where the message delays between u and v are $1/2$. Then all the messages sent by u and v at time i according to the clock of the sender arrive at time $i + 1/2$ according to the clock of the receiver.

Then consider the following cases

- $t_u = t_v + 1/2$, $t(u \to v) = 1$, $t(v \to u) = 0$
- $t_u = t_v - 1/2$, $t(u \to v) = 0$, $t(v \to u) = 1$,

where the message delivery time is always fast for one node and slow for the other and the logical clocks are off by $1/2$. In both scenarios, the messages sent at time i according to the clock of the sender arrive at time $i + 1/2$ according to the logical clock of the receiver. Therefore, for nodes u and v, both cases with clock drift seem the same as the case with perfectly synchronized clocks. Furthermore, in a linked list of D nodes, the left- and rightmost nodes l, r cannot distinguish $t_l = t_r + D/2$ from $t_l = t_r - D/2$. □

Remarks:

- From Theorem 10.17, it directly follows that all the clock synchronization algorithms we studied have a global skew of $\Omega(D)$.
- Many natural algorithms manage to achieve a global clock skew of $\mathcal{O}(D)$.

As both the message jitter and hardware clock drift are bounded by constants, it feels like we should be able to get a constant drift between neighboring nodes. As synchronizer α pays most attention to the local synchronization, we take a look at a protocol inspired by the synchronizer α. A pseudo-code representation for the clock synchronization protocol α is given in Algorithm 10.18.

Algorithm 10.18 Clock synchronization α (at node v)

```
1: repeat
2:    send logical time t_v to all neighbors
3:    if Receive logical time t_u, where t_u > t_v, from any neighbor u then
4:       t_v := t_u
5:    end if
6: until done
```

Lemma 10.19. *The clock synchronization protocol α has a local skew of $\Omega(n)$.*

Proof. Let the graph be a linked list of D nodes. We denote the nodes by $v_1, v_2, \ldots, v_D$ from left to right and the logical clock of node v_i by t_i. Apart from the left-most node v_1 all hardware clocks run with speed 1 (real time). Node v_1 runs at maximum speed, i.e. the time between two pulses is not 1 but $1 - \epsilon$. Assume that initially all message delays are 1. After some time, node v_1 will start to speed up v_2, and after some more time v_2 will speed up v_3, and so on. At some point of time, we will have a clock skew of 1 between any two neighbors. In particular $t_1 = t_D + D - 1$.

Now we start playing around with the message delays. Let $t_1 = T$. First we set the delay between the v_1 and v_2 to 0. Now node v_2 immediately adjusts its logical clock to T. After this event (which is instantaneous in our model) we set the delay between v_2 and v_3 to 0, which results in v_3 setting its logical clock to T as well. We perform this successively to all pairs of nodes until v_{D-2} and v_{D-1}. Now node v_{D-1} sets its logical clock to T, which indicates that the difference between the logical clocks of v_{D-1} and v_D is $T - (T - (D - 1)) = D - 1$. □

Remarks:

- The introduced examples may seem cooked-up, but examples like this exist in all networks, and for all algorithms. Indeed, it was shown that any natural clock synchronization algorithm must have a bad local skew. In particular, a protocol that averages between all neighbors is even worse than the introduced α algorithm. This algorithm has a clock skew of $\Omega(D^2)$ in the linked list, at all times.
- It was shown that the local clock skew is $\Theta(\log D)$, i.e., there is a protocol that achieves this bound, and there is a proof that no algorithm can be better than this bound!
- Note that these are worst-case bounds. In practice, clock drift and message delays may not be the worst possible, typically the speed of hardware clocks changes at a comparatively slow pace and the message transmission times follow a benign probability distribution. If we assume this, better protocols do exist.

Chapter Notes

The idea behind synchronizers is quite intuitive and as such, synchronizers α and β were implicitly used in various asynchronous algorithms [Gal76, Cha79, CL85] before being proposed as separate entities. The general idea of applying synchronizers to run synchronous algorithms in asynchronous networks was first introduced by Awerbuch [Awe85a]. His work also formally introduced the synchronizers α and β. Improved synchronizers that exploit inactive nodes or hypercube networks were presented in [AP90, PU87].

Naturally, as synchronizers are motivated by practical difficulties with local clocks, there are plenty of real life applications. Studies regarding applications can be found in, e.g., [SM86, Awe85b, LTC89, AP90, PU87]. Synchronizers in the presence of network failures have been discussed in [AP88, HS94].

It has been known for a long time that the global clock skew is $\Theta(D)$ [LL84, ST87]. The problem of synchronizing the clocks of nearby nodes was introduced by Fan and Lynch in [LF04]; they proved a surprising lower bound of $\Omega(\log D / \log\log D)$ for the local skew. The first algorithm providing a non-trivial local skew of $\mathcal{O}(\sqrt{D})$ was given in [LW06]. Later, matching upper and lower bounds of $\Theta(\log D)$ were given in [LLW10]. The problem has also been studied in a dynamic setting [KLO09, KLLO10].

Clock synchronization is a well-studied problem in practice, for instance regarding the global clock skew in sensor networks, e.g. [EGE02, GKS03, MKSL04, PSJ04]. One more recent line of work is focussing on the problem of minimizing the local clock skew [BvRW07, SW09, LSW09, FW10, FZTS11].

Bibliography

[AP88] Baruch Awerbuch and David Peleg. Adapting to Asynchronous Dynamic Networks with Polylogarithmic Overhead. In *24th ACM Symposium on Foundations of Computer Science (FOCS)*, pages 206–220, 1988.

[AP90] Baruch Awerbuch and David Peleg. Network Synchronization with Polylogarithmic Overhead. In *Proceedings of the 31st IEEE Symposium on Foundations of Computer Science (FOCS)*, 1990.

[Awe85a] Baruch Awerbuch. Complexity of Network Synchronization. *Journal of the ACM (JACM)*, 32(4):804–823, October 1985.

[Awe85b] Baruch Awerbuch. Reducing Complexities of the Distributed Maxflow and Breadth-first-search Algorithms by Means of Network Synchronization. *Networks*, 15:425–437, 1985.

[BvRW07] Nicolas Burri, Pascal von Rickenbach, and Roger Wattenhofer. Dozer: Ultra-Low Power Data Gathering in Sensor Networks. In *International Conference on Information Processing in Sensor Networks (IPSN), Cambridge, Massachusetts, USA*, April 2007.

[Cha79] E.J.H. Chang. *Decentralized Algorithms in Distributed Systems*. PhD thesis, University of Toronto, 1979.

[CL85] K. Mani Chandy and Leslie Lamport. Distributed Snapshots: Determining Global States of Distributed Systems. *ACM Transactions on Computer Systems*, 1:63–75, 1985.

[EGE02] Jeremy Elson, Lewis Girod, and Deborah Estrin. Fine-grained Network Time Synchronization Using Reference Broadcasts. *ACM SIGOPS Operating Systems Review*, 36:147–163, 2002.

[FW10] Roland Flury and Roger Wattenhofer. Slotted Programming for Sensor Networks. In *International Conference on Information Processing in Sensor Networks (IPSN), Stockholm, Sweden*, April 2010.

[FZTS11] Federico Ferrari, Marco Zimmerling, Lothar Thiele, and Olga Saukh. Efficient Network Flooding and Time Synchronization with Glossy. In *Proceedings of the 10th International Conference on Information Processing in Sensor Networks (IPSN)*, pages 73–84, 2011.

[Gal76] Robert Gallager. Distributed Minimum Hop Algorithms. Technical report, Lab. for Information and Decision Systems, 1976.

[GKS03] Saurabh Ganeriwal, Ram Kumar, and Mani B. Srivastava. Timing-sync Protocol for Sensor Networks. In *Proceedings of the 1st international conference on Embedded Networked Sensor Systems (SenSys)*, 2003.

[HS94] M. Harrington and A. K. Somani. Synchronizing Hypercube Networks in the Presence of Faults. *IEEE Transactions on Computers*, 43(10):1175–1183, 1994.

[KLLO10] Fabian Kuhn, Christoph Lenzen, Thomas Locher, and Rotem Oshman. Optimal Gradient Clock Synchronization in Dynamic Networks. In *29th Symposium on Principles of Distributed Computing (PODC), Zurich, Switzerland*, July 2010.

[KLO09] Fabian Kuhn, Thomas Locher, and Rotem Oshman. Gradient Clock Synchronization in Dynamic Networks. In *21st ACM Symposium on Parallelism in Algorithms and Architectures (SPAA), Calgary, Canada*, August 2009.

[LF04] Nancy Lynch and Rui Fan. Gradient Clock Synchronization. In *Proceedings of the 23rd Annual ACM Symposium on Principles of Distributed Computing (PODC)*, 2004.

[LL84] Jennifer Lundelius and Nancy Lynch. An Upper and Lower Bound for Clock Synchronization. *Information and Control*, 62:190–204, 1984.

[LLW10] Christoph Lenzen, Thomas Locher, and Roger Wattenhofer. Tight Bounds for Clock Synchronization. In *Journal of the ACM, Volume 57, Number 2*, January 2010.

[LSW09] Christoph Lenzen, Philipp Sommer, and Roger Wattenhofer. Optimal Clock Synchronization in Networks. In *7th ACM Conference on Embedded Networked Sensor Systems (SenSys), Berkeley, California, USA*, November 2009.

[LTC89] K. B. Lakshmanan, K. Thulasiraman, and M. A. Comeau. An Efficient Distributed Protocol for Finding Shortest Paths in Networks with Negative Weights. *IEEE Trans. Softw. Eng.*, 15:639–644, 1989.

[LW06] Thomas Locher and Roger Wattenhofer. Oblivious Gradient Clock Synchronization. In *20th International Symposium on Distributed Computing (DISC), Stockholm, Sweden*, September 2006.

[MKSL04] Miklós Maróti, Branislav Kusy, Gyula Simon, and Ákos Lédeczi. The Flooding Time Synchronization Protocol. In *Proceedings of the 2nd international Conference on Embedded Networked Sensor Systems*, SenSys '04, 2004.

[PSJ04] Santashil PalChaudhuri, Amit Kumar Saha, and David B. Johnson. Adaptive Clock Synchronization in Sensor Networks. In *Proceedings of the 3rd International Symposium on Information Processing in Sensor Networks*, IPSN '04, 2004.

[PU87] David Peleg and Jeffrey D. Ullman. An Optimal Synchronizer for the Hypercube. In *Proceedings of the sixth annual ACM Symposium on Principles of Distributed Computing*, PODC '87, pages 77–85, 1987.

[SM86] Baruch Shieber and Shlomo Moran. Slowing Sequential Algorithms for Obtaining Fast Distributed and Parallel Algorithms: Maximum Matchings. In *Proceedings of the fifth annual ACM Symposium on Principles of Distributed Computing*, PODC '86, pages 282–292, 1986.

[ST87] T. K. Srikanth and S. Toueg. Optimal Clock Synchronization. *Journal of the ACM*, 34:626–645, 1987.

[SW09] Philipp Sommer and Roger Wattenhofer. Gradient Clock Synchronization in Wireless Sensor Networks. In *8th ACM/IEEE International Conference on Information Processing in Sensor Networks (IPSN), San Francisco, USA*, April 2009.

Chapter 11

Communication Complexity

This chapter is on "hard" problems in distributed computing. In sequential computing, there are NP-hard problems which are conjectured to take exponential time. Is there something similar in distributed computing? Using flooding/echo (Algorithms 2.9,2.10) from Chapter 2, everything so far was solvable basically in $\mathcal{O}(D)$ time, where D is the diameter of the network.

11.1 Diameter & APSP

But how do we compute the diameter itself!?! With flooding/echo, of course!

Algorithm 11.1 Naive Diameter Construction

1: all nodes compute their radius by synchronous flooding/echo
2: all nodes flood their radius on the constructed BFS tree
3: the maximum radius a node sees is the diameter

Remarks:

- Since all these phases only take $\mathcal{O}(D)$ time, nodes know the diameter in $\mathcal{O}(D)$ time, which is asymptotically optimal.
- However, there is a problem! Nodes are now involved in n parallel flooding/echo operations, thus a node may have to handle many and big messages in one single time step. Although this is not strictly illegal in the message passing model, it still feels like cheating! A natural question is whether we can do the same by just sending short messages in each round.
- In Definition 1.8 of Chapter 1 we postulated that nodes should send only messages of "reasonable" size. In this chapter we strengthen the definition a bit, and require that each message should have at most $\mathcal{O}(\log n)$ bits. This is generally enough to communicate a constant number of ID's or values to neighbors, but not enough to communicate everything a node knows!
- A simple way to avoid large messages is to split them into small messages that are sent using several rounds. This can cause that messages

are getting delayed in some nodes but not in others. The flooding might not use edges of a BFS tree anymore! These floodings might not compute correct distances anymore! On the other hand we know that the maximal message size in Algorithm 11.1 is $\mathcal{O}(n \log n)$. So we could just simulate each of these "big message" rounds by n "small message" rounds using small messages. This yields a runtime of $\mathcal{O}(nD)$ which is not desirable. A third possible approach is "starting each flooding/echo one after each other" and results in $\mathcal{O}(nD)$ in the worst case as well.

- So let us fix the above algorithm! The key idea is to arrange the flooding-echo processes in a more organized way: Start the flooding processes in a certain order and prove that at any time, each node is only involved in one flooding. This is realized in Algorithm 11.3.

Definition 11.2. *(BFS_v) Performing a breadth first search at node v produces spanning tree BFS_v (see Chapter 2). This takes time $\mathcal{O}(D)$ using small messages.*

Remarks:

- A spanning tree of a graph G can be traversed in time $\mathcal{O}(n)$ by sending a pebble over an edge in each time slot.
- This can be done using, e.g., a depth first search (DFS): Start at the root of a tree, recursively visit all nodes in the following way. If the current node still has an unvisited child, then the pebble always visits that child first. Return to the parent only when all children have been visited.
- Algorithm 11.3 works as follows: Given a graph G, first a leader l computes its BFS tree BFS_l. Then we send a pebble P to traverse tree BFS_l. Each time pebble P enters a node v for the first time, P waits one time slot, and then starts a breadth first search (BFS) – using edges in G – from v with the aim of computing the distances from v to all other nodes. Since we start a BFS_v from every node v, each node u learns its distance to all these nodes v during the according execution of BFS_v. There is no need for an echo-process at the end of BFS_u.

Remarks:

- Having all distances is nice, but how do we get the diameter? Well, as before, each node could just flood its radius (its maximum distance) into the network. However, messages are small now and we need to modify this slightly. In each round a node only sends the maximal distance that it is aware of to its neighbors. After D rounds each node will know the maximum distance among all nodes.

Lemma 11.4. *In Algorithm 11.3, at no time a node w is simultaneously active for both BFS_u and BFS_v.*

Algorithm 11.3 Computes APSP on G.

```
1: Assume we have a leader node l (if not, compute one first)
2: compute BFS_l of leader l
3: send a pebble P to traverse BFS_l in a DFS way;
4: while P traverses BFS_l do
5:   if P visits a new node v then
6:     wait one time slot;   // avoid congestion
7:     start BFS_v from node v;   // compute all distances to v
8:     // the depth of node u in BFS_v is d(u, v)
9:   end if
10: end while
```

Proof. Assume a BFS_u is started at time t_u at node u. Then node w will be involved in BFS_u at time $t_u + d(u, w)$. Now, consider a node v whose BFS_v is started at time $t_v > t_u$. According to the algorithm this implies that the pebble visits v after u and took some time to travel from u to v. In particular, the time to get from u to v is at least $d(u, v)$, in addition at least node v is visited for the first time (which involves waiting at least one time slot), and we have $t_v \geq t_u + d(u, v) + 1$. Using this and the triangle inequality, we get that node w is involved in BFS_v strictly after being involved in BFS_u since $t_v + d(v, w) \geq (t_u + d(u, v) + 1) + d(v, w) \geq t_u + d(u, w) + 1 > t_u + d(u, w)$. □

Theorem 11.5. *Algorithm 11.3 computes APSP (all pairs shortest path) in time $\mathcal{O}(n)$.*

Proof. Since the previous lemma holds for any pair of vertices, no two BFS "interfere" with each other, i.e. all messages can be sent on time without congestion. Hence, all BFS stop at most D time slots after they were started. We conclude that the runtime of the algorithm is determined by the time $\mathcal{O}(D)$ we need to build tree BFS_l, plus the time $\mathcal{O}(n)$ that P needs to traverse BFS_l, plus the time $\mathcal{O}(D)$ needed by the last BFS that P initiated. Since $D \leq n$, this is all in $\mathcal{O}(n)$. □

Remarks:

- All of a sudden our algorithm needs $\mathcal{O}(n)$ time, and possibly $n \gg D$. We should be able to do better, right?!
- Unfortunately not! One can show that computing the diameter of a network needs $\Omega(n/\log n)$ time.
- Note that one can check whether a graph has diameter 1 by exchanging some specific information such as degree with the neighbors. However, already checking diameter 2 is difficult.

11.2 Lower Bound Graphs

We define a family $\mathcal{G}$ of graphs that we use to prove a lower bound on the rounds needed to compute the diameter. To simplify our analysis, we assume that $(n-2)$ can be divided by 8. We start by defining four sets of nodes, each

consisting of $q = q(n) := (n-2)/4$ nodes. Throughout this chapter we write $[q]$ as a short version of $\{1, \ldots, q\}$ and define:

$$
\begin{aligned}
\mathbf{L_0} &:= \{l_i \mid i \in [q]\} && \text{// upper left in Figure 11.6}\\
\mathbf{L_1} &:= \{l'_i \mid i \in [q]\} && \text{// lower left}\\
\mathbf{R_0} &:= \{r_i \mid i \in [q]\} && \text{// upper right}\\
\mathbf{R_1} &:= \{r'_i \mid i \in [q]\} && \text{// lower right}
\end{aligned}
$$

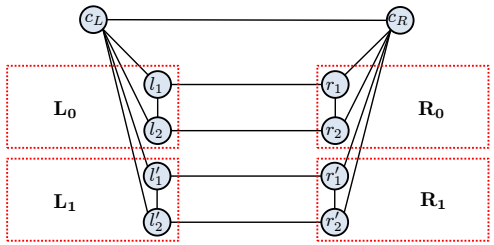


Figure 11.6: The above skeleton G' contains $n = 10$ nodes, such that $q = 2$.

We add node c_L and connect it to all nodes in $\mathbf{L_0}$ and $\mathbf{L_1}$. Then we add node c_R, connected to all nodes in $\mathbf{R_0}$ and $\mathbf{R_1}$. Furthermore, nodes c_L and c_R are connected by an edge. For $i \in [q]$ we connect l_i to r_i and l'_i to r'_i. Also we add edges such that nodes in $\mathbf{L_0}$ are a clique, nodes in $\mathbf{L_1}$ are a clique, nodes in $\mathbf{R_0}$ are a clique, and nodes in $\mathbf{R_1}$ are a clique. The resulting graph is called G'. Graph G' is the skeleton of any graph in family $\mathcal{G}$.

More formally skeleton $G' = (V', E')$ is:

$$V' := \mathbf{L_0} \cup \mathbf{L_1} \cup \mathbf{R_0} \cup \mathbf{R_1} \cup \{c_L, c_R\}$$

$$
\begin{aligned}
E' := & \bigcup_{v \in \mathbf{L_0} \cup \mathbf{L_1}} \{(v, c_L)\} && \text{// connections to } c_L\\
\cup & \bigcup_{v \in \mathbf{R_0} \cup \mathbf{R_1}} \{(v, c_R)\} && \text{// connections to } c_R\\
\cup & \bigcup_{i \in [q]} \{(l_i, r_i), (l'_i, r'_i)\} \cup \{(c_L, c_R)\} && \text{// connects left to right}\\
\cup & \bigcup_{S \in \{\mathbf{L_0}, \mathbf{L_1}, \mathbf{R_0}, \mathbf{R_1}\}} \bigcup_{u \neq v \in S} \{(u, v)\} && \text{// clique edges}
\end{aligned}
$$

To simplify our arguments, we partition G' into two parts: **Part L** is the subgraph induced by nodes $\mathbf{L_0} \cup \mathbf{L_1} \cup \{c_L\}$. **Part R** is the subgraph induced by nodes $\mathbf{R_0} \cup \mathbf{R_1} \cup \{c_R\}$.

Family $\mathcal{G}$ contains any graph G that is derived from G' by adding any combination of edges of the form (l_i, l'_j) resp. (r_i, r'_j) with $l_i \in \mathbf{L_0}$, $l'_j \in \mathbf{L_1}$, $r_i \in \mathbf{R_0}$, and $r'_j \in \mathbf{R_1}$.

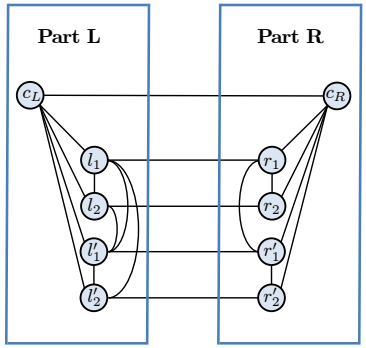


Figure 11.7: The above graph G has $n = 10$ and is a member of family $\mathcal{G}$. What is the diameter of G?

Lemma 11.8. *The diameter of a graph $G = (V, E) \in \mathcal{G}$ is 2 if and only if: For each tuple (i, j) with $i, j \in [q]$, there is either edge (l_i, l'_j) or edge (r_i, r'_j) (or both edges) in E.*

Proof. Note that the distance between most pairs of nodes is at most 2. In particular, the radius of c_L resp. c_R is 2. Thanks to c_L resp. c_R the distance between, any two nodes within **Part L** resp. within **Part R** is at most 2. Because of the cliques $\mathbf{L_0}, \mathbf{L_1}, \mathbf{R_0}, \mathbf{R_1}$, distances between l_i and r_j resp. l'_i and r'_j is at most 2.

The only interesting case is between a node $l_i \in \mathbf{L_0}$ and node $r'_j \in \mathbf{R_1}$ (or, symmetrically, between $l'_j \in \mathbf{L_1}$ and node $r_i \in \mathbf{R_0}$). If either edge (l_i, l'_j) or edge (r_i, r'_j) is present, then this distance is 2, since the path (l_i, l'_j, r'_j) or the path (l_i, r_i, r'_j) exists. If neither of the two edges exist, then the neighborhood of l_i consists of $\{c_L, r_i\}$, all nodes in $\mathbf{L_0}$, and some nodes in $\mathbf{L_1} \setminus \{l'_j\}$, and the neighborhood of r'_j consists of $\{c_R, l'_j\}$, all nodes in $\mathbf{R_1}$, and some nodes in $\mathbf{R_0} \setminus \{r_i\}$ (see for example Figure 11.9 with $i = 2$ and $j = 2$.) Since the two neighborhoods do not share a common node, the distance between l_i and r'_j is (at least) 3. □

Remarks:

- Each part contains up to $q^2 \in \Theta(n^2)$ edges not belonging to the skeleton.
- There are $2q + 1 \in \Theta(n)$ edges connecting the left and the right part. Since in each round we can transmit $\mathcal{O}(\log n)$ bits over each edge (in each direction), the bandwidth between **Part L** and **Part R** is $\mathcal{O}(n \log n)$.
- If we transmit the information of the $\Theta(n^2)$ edges in a naive way with a bandwidth of $\mathcal{O}(n \log n)$, we need $\Omega(n/\log n)$ time. But maybe we can do better?!? Can an algorithm be smarter and only send the information that is really necessary to tell whether the diameter is 2?
- It turns out that any algorithm needs $\Omega(n/\log n)$ rounds, since the information that is really necessary to tell that the diameter is larger than 2 contains basically $\Theta(n^2)$ bits.

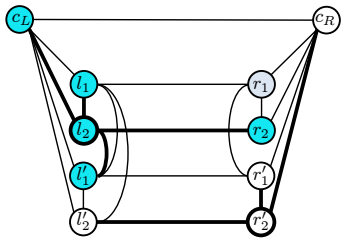


Figure 11.9: Nodes in the neighborhood of l_2 are cyan, the neighborhood of r'_2 is white. Since these neighborhoods do not intersect, the distance of these two nodes is $\mathrm{d}(l_2, r'_2) > 2$. If edge (l_2, l'_2) was included, their distance would be 2.

11.3 Communication Complexity

To prove the last remark formally, we can use arguments from two-party communication complexity. This area essentially deals with a basic version of distributed computation: two parties are given some input each and want to solve a task on this input.

We consider two students (Alice and Bob) at two different universities connected by a communication channel (e.g., via email) and we assume this channel to be reliable. Now Alice and Bob want to check whether they received the same problem set for homework (we assume their professors are lazy and wrote it on the black board instead of putting a nicely prepared document online.) Do Alice and Bob really need to type the whole problem set into their emails? In a more formal way: Alice receives an k-bit string x and Bob another k-bit string y, and the goal is for both of them to compute the equality function.

Definition 11.10. *(Equality.) We define the equality function* EQ *to be:*

$$\mathrm{EQ}(x, y) := \begin{cases} 1 & : x = y \\ 0 & : x \neq y \end{cases}.$$

Remarks:

- In a more general setting, Alice and Bob are interested in computing a certain function $f : \{0,1\}^k \times \{0,1\}^k \to \{0,1\}$ with the least amount of communication between them. Of course they can always succeed by having Alice send her whole k-bit string to Bob, who then computes the function, but the idea here is to find clever ways of calculating f with less than k bits of communication. We measure how clever they can be as follows:

Definition 11.11. *(Communication complexity CC.) The communication complexity of protocol A for function f is $CC(A,f) :=$ minimum number of bits exchanged between Alice and Bob in the worst case when using A. The communication complexity of f is $CC(f) := \min\{CC(A,f) \mid A \text{ solves } f\}$. That is the minimal number of bits that the best protocol needs to send in the worst case.*

Definition 11.12. *For a given function f, we define a $2^k \times 2^k$ matrix M^f representing f. That is $M^f_{x,y} := f(x,y)$.*

Example 11.13. *For* EQ*, in case $k = 3$, matrix M^{EQ} looks like this:*

EQ	000	001	010	011	100	101	110	111	$\leftarrow x$
000	1	0	0	0	0	0	0	0	
001	0	1	0	0	0	0	0	0	
010	0	0	1	0	0	0	0	0	
011	0	0	0	1	0	0	0	0	
100	0	0	0	0	1	0	0	0	
101	0	0	0	0	0	1	0	0	
110	0	0	0	0	0	0	1	0	
111	0	0	0	0	0	0	0	1	
$\uparrow y$									

As a next step we define a (combinatorial) monochromatic rectangle. These are "submatrices" of M^f which contain the same entry.

Definition 11.14. *(monochromatic rectangle.) A set $R \subseteq \{0,1\}^k \times \{0,1\}^k$ is called a monochromatic rectangle, if*

- *whenever $(x_1,y_1) \in R$ and $(x_2,y_2) \in R$ then $(x_1,y_2) \in R$.*
- *there is a fixed z such that $f(x,y) = z$ for all $(x,y) \in R$.*

Example 11.15. *The first three of the following rectangles are monochromatic, the last one is not:*

$R_1 = \{011\} \times \{011\}$

$R_2 = \{011, 100, 101, 110\} \times \{000, 001\}$

$R_3 = \{000, 001, 101\} \times \{011, 100, 110, 111\}$

$R_4 = \{000, 001\} \times \{000, 001\}$

Example 11.13: light gray
Example 11.13: gray
Example 11.13: dark gray
Example 11.13: boxed

Each time Alice and Bob exchange a bit, they can eliminate columns/rows of the matrix M^f and a combinatorial rectangle is left. They can stop communicating when this remaining rectangle is monochromatic. However, maybe there is a more efficient way to exchange information about a given bit string than just naively transmitting contained bits? In order to cover all possible ways of communication, we need the following definition:

Definition 11.16. *(fooling set.) A set $S \subset \{0,1\}^k \times \{0,1\}^k$ fools f if there is a fixed z such that*

- *$f(x,y) = z$ for each $(x,y) \in S$*
- *For any $(x_1,y_1) \neq (x_2,y_2) \in S$, the rectangle $\{x_1,x_2\} \times \{y_1,y_2\}$ is not monochromatic: Either $f(x_1,y_2) \neq z$, $f(x_2,y_1) \neq z$ or both $\neq z$.*

Example 11.17. *Consider $S = \{(000,000),(001,001)\}$. Take a look at the non-monochromatic rectangle R_4 in Example 11.15. Verify that S is indeed a fooling set for* EQ*!*

Remarks:

- Can you find a larger fooling set for *EQ*?
- We assume that Alice and Bob take turns in sending a bit. This results in 2 possible actions (send 0/1) per round and in 2^t *action patterns* during a sequence of t rounds.

Lemma 11.18. *If S is a fooling set for f, then $CC(f) = \Omega(\log|S|)$.*

Proof. We prove the statement via contradiction: fix a protocol A and assume that it needs $t < \log(|S|)$ rounds in the worst case. Then there are 2^t possible action patterns, with $2^t < |S|$. Hence for at least two elements of S, let us call them $(x_1,y_1),(x_2,y_2)$, protocol A produces the same action pattern P. Naturally, the action pattern on the alternative inputs $(x_1,y_2),(x_2,y_1)$ will be P as well: in the first round Alice and Bob have no information on the other party's string and send the same bit that was sent in P. Based on this, they determine the second bit to be exchanged, which will be the same as the second one in P since they cannot distinguish the cases. This continues for all t rounds. We conclude that after t rounds, Alice does not know whether Bob's input is y_1 or y_2 and Bob does not know whether Alice's input is x_1 or x_2. By the definition of fooling sets, either

- $f(x_1,y_2) \neq f(x_1,y_1)$ in which case Alice (with input x_1) does not know the solution yet,

or

- $f(x_2,y_1) \neq f(x_1,y_1)$ in which case Bob (with input y_1) does not know the solution yet.

This contradicts the assumption that A leads to a correct decision for all inputs after t rounds. Therefore at least $\log(|S|)$ rounds are necessary. □

Theorem 11.19. $CC(\mathrm{EQ}) = \Omega(k)$.

Proof. The set $S := \{(x,x) \mid x \in \{0,1\}^k\}$ fools *EQ* and has size 2^k. Now apply Lemma 11.18. □

Definition 11.20. *Denote the negation of a string z by $\overline{z}$ and by $x \circ y$ the concatenation of strings x and y.*

Lemma 11.21. *Let x, y be k-bit strings. Then $x \neq y$ if and only if there is an index $i \in [2k]$ such that the i^{th} bit of $x \circ \overline{x}$ and the i^{th} bit of $\overline{y} \circ y$ are both* 0.

Proof. If $x \neq y$, there is an $j \in [k]$ such that x and y differ in the j^{th} bit. Therefore either the j^{th} bit of both x and $\overline{y}$ is 0, or the j^{th} bit of $\overline{x}$ and y is 0. For this reason, there is an $i \in [2k]$ such that $x \circ \overline{x}$ and $\overline{y} \circ y$ are both 0 at position i.

If $x = y$, then for any $i \in [2k]$ it is always the case that either the i^{th} bit of $x \circ \overline{x}$ is 1 or the i^{th} bit of $\overline{y} \circ y$ (which is the negation of $x \circ \overline{x}$ in this case) is 1. □

Remarks:

- With these insights we get back to the problem of computing the diameter of a graph and relate this problem to *EQ*.

Definition 11.22. *Using the parameter q defined before, we define a bijective map between all pairs x, y of q^2-bit strings and the graphs in $\mathcal{G}$: each pair of strings x, y is mapped to graph $G_{x,y} \in \mathcal{G}$ that is derived from skeleton G' by adding*

- *edge (l_i, l'_j) to* ***Part L*** *if and only if the $(j + q \cdot (i-1))^{th}$ bit of x is* 1.
- *edge (r_i, r'_j) to* ***Part R*** *if and only if the $(j + q \cdot (i-1))^{th}$ bit of y is* 1.

Remarks:

- Clearly, **Part L** of $G_{x,y}$ depends on x only and **Part R** depends on y only.

Lemma 11.23. *Let x and y be $\frac{q^2}{2}$-bit strings given to Alice and Bob.*[1] *Then graph $G := G_{x \circ \overline{x}, \overline{y} \circ y} \in \mathcal{G}$ has diameter* 2 *if and only if $x = y$.*

Proof. By Lemma 11.21 and the construction of G, there is neither edge (l_i, l'_j) nor edge (r_i, r'_j) in $E(G)$ for some (i, j) if and only if $x \neq y$. Applying Lemma 11.8 yields: G has diameter 2 if and only if $x = y$. □

Theorem 11.24. *Any distributed algorithm A that decides whether a graph G has diameter* 2 *needs $\Omega\left(\frac{n}{\log n} + D\right)$ time.*

Proof. Computing D for sure needs time $\Omega(D)$. It remains to prove $\Omega\left(\frac{n}{\log n}\right)$. Assume there is a distributed algorithm A that decides whether the diameter of a graph is 2 in time $o(n/\log n)$. When Alice and Bob are given $\frac{q^2}{2}$-bit inputs x and y, they can simulate A to decide whether $x = y$ as follows: Alice constructs **Part L** of $G_{x \circ \overline{x}, \overline{y} \circ y}$ and Bob constructs **Part R**. As we remarked, both parts are independent of each other such that **Part L** can be constructed by Alice without knowing y and **Part R** can be constructed by Bob without knowing x. Furthermore, $G_{x \circ \overline{x}, \overline{y} \circ y}$ has diameter 2 if and only if $x = y$ (Lemma 11.23.)

Now Alice and Bob simulate the distributed algorithm A round by round: In the first round, they determine which messages the nodes in their part of

[1] Thats why we need that $n - 2$ can be divided by 8.

G would send. Then they use their communication channel to exchange all $2(2q + 1) \in \Theta(n)$ messages that would be sent over edges between **Part L** and **Part R** in this round while executing A on G. Based on this Alice and Bob determine which messages would be sent in round two and so on. For each round simulated by Alice and Bob, they only need to communicate $\mathcal{O}(n \log n)$ bits: $\mathcal{O}(\log n)$ bits for each of $\mathcal{O}(n)$ messages. Since A makes a decision after $o(n/\log n)$ rounds, this yields a total communication of $o(n^2)$ bits. On the other hand, Lemma 11.19 states that to decide whether x equals y, Alice and Bob need to communicate at least $\Omega\left(\frac{q^2}{2}\right) = \Omega(n^2)$ bits. A contradiction. □

Remarks:

- Until now we only considered deterministic algorithms. Can one do better using randomness?

Algorithm 11.25 Randomized evaluation of *EQ*.

1: Alice and Bob use public randomness. That is they both have access to the same random bit string $z \in \{0,1\}^k$
2: Alice sends bit $a := \sum_{i \in [k]} x_i \cdot z_i \mod 2$ to Bob
3: Bob sends bit $b := \sum_{i \in [k]} y_i \cdot z_i \mod 2$ to Alice
4: **if** $a \neq b$ **then**
5: we know $x \neq y$
6: **end if**

Lemma 11.26. *If $x \neq y$, Algorithm 11.25 discovers $x \neq y$ with probability at least* 1/2.

Proof. Note that if $x = y$ we have $a = b$ for sure.

If $x \neq y$, Algorithm 11.25 may not reveal inequality. For instance, for $k = 2$, if $x = 01$, $y = 10$ and $z = 11$ we get $a = b = 1$. In general, let I be the set of indices where $x_i \neq y_i$, i.e. $I := \{i \in [k] \mid x_i \neq y_i\}$. Since $x \neq y$, we know that $|I| > 0$. We have

$$|a - b| \equiv \sum_{i \in I} z_i \pmod 2,$$

and since all z_i with $i \in I$ are random, we get that $a \neq b$ with probability at least 1/2. □

Remarks:

- By excluding the vector $z = 0^k$ we can even get a discovery probability strictly larger than 1/2.
- Repeating the Algorithm 11.25 with different random strings z, the error probability can be reduced arbitrarily.
- Does this imply that there is a fast randomized algorithm to determine the diameter? Unfortunately not!

- Sometimes public randomness is not available, but private randomness is. Here Alice has her own random string and Bob has his own random string. A modified version of Algorithm 11.25 also works with private randomness at the cost of the runtime.

- One can prove an $\Omega(n/\log n)$ lower bound for any randomized distributed algorithm that computes the diameter. To do so one considers the disjointness function *DISJ* instead of equality. Here, Alice is given a subset $X \subseteq [k]$ and and Bob is given a subset $Y \subseteq [k]$ and they need to determine whether $Y \cap X = \emptyset$. (X and Y can be represented by k-bit strings x, y.) The reduction is similar as the one presented above but uses graph $G_{\overline{x},\overline{y}}$ instead of $G_{x \circ \overline{x},\overline{y} \circ y}$. However, the lower bound for the randomized communication complexity of *DISJ* is more involved than the lower bound for $CC(EQ)$.

- Since one can compute the diameter given a solution for APSP, an $\Omega(n/\log n)$ lower bound for APSP is implied. As such, our simple Algorithm 11.3 is almost optimal!

- Many prominent functions allow for a low communication complexity. For instance, $CC(PARITY) = 2$. What is the Hamming distance (number of different entries) of two strings? It is known that $CC(HAM \geq d) = \Omega(d)$. Also, CC(decide whether "$HAM \geq k/2 + \sqrt{k}$" or "$HAM \leq k/2 - \sqrt{k}$") $= \Omega(k)$, even when using randomness. This problem is known as the Gap-Hamming-Distance.

- Lower bounds in communication complexity have many applications. Apart from getting lower bounds in distributed computing, one can also get lower bounds regarding circuit depth or query times for static data structures.

- In the distributed setting with limited bandwidth we showed that computing the diameter has about the same complexity as computing all pairs shortest paths. In contrast, in sequential computing, it is a major open problem whether the diameter can be computed faster than all pairs shortest paths. No nontrivial lower bounds are known, only that $\Omega(n^2)$ steps are needed – partly due to the fact that there can be n^2 edges/distances in a graph. On the other hand the currently best algorithm uses fast matrix multiplication and terminates after $\mathcal{O}(n^{2.3727})$ steps.

11.4 Distributed Complexity Theory

We conclude this chapter with a short overview on the main complexity classes of distributed message passing algorithms. Given a network with n nodes and diameter D, we managed to establish a rich selection of upper and lower bounds regarding how much time it takes to solve or approximate a problem. Currently we know five main distributed complexity classes:

- Strictly *local* problems can be solved in constant $\mathcal{O}(1)$ time, e.g., a constant approximation of a dominating set in a planar graph.

- Just a little bit slower are problems that can be solved in *log-star* $\mathcal{O}(\log^* n)$ time, e.g., many combinatorial optimization problems in special graph classes such as growth bounded graphs. 3-coloring a ring takes $\mathcal{O}(\log^* n)$.

- A large body of problems is *polylogarithmic* (or *pseudo-local*), in the sense that they seem to be strictly local but are not, as they need $\mathcal{O}(\text{polylog } n)$ time, e.g., the maximal independent set problem.

- There are problems which are *global* and need $\mathcal{O}(D)$ time, e.g., to count the number of nodes in the network.

- Finally there are problems which need *polynomial* $\mathcal{O}(\text{poly } n)$ time, even if the diameter D is a constant, e.g., computing the diameter of the network.

Chapter Notes

The linear time algorithm for computing the diameter was discovered independently by [HW12, PRT12]. The presented matching lower bound is by Frischknecht et al. [FHW12], extending techniques by [DHK+11].

Due to its importance in network design, shortest path-problems in general and the APSP problem in particular were among the earliest studied problems in distributed computing. Developed algorithms were immediately used, e.g., as early as in 1969 in the ARPANET (see [Lyn96], p.506). Routing messages via shortest paths were extensively discussed to be beneficial in [Taj77, MS79, MRR80, SS80, CM82] and in many other papers. It is not surprising that there is plenty of literature dealing with algorithms for distributed APSP, but most of them focused on secondary targets such as trading time for message complexity. E.g., papers [AR78, Tou80, Che82] obtain a communication complexity of roughly $\mathcal{O}(n \cdot m)$ bits/messages and still require superlinear runtime. Also a lot of effort was spent to obtain fast sequential algorithms for various versions of computing APSP or related problems such as the diameter problem, e.g., [CW90, AGM91, AMGN92, Sei95, SZ99, BVW08]. These algorithms are based on fast matrix multiplication such that currently the best runtime is $\mathcal{O}(n^{2.3727})$ due to [Wil12].

The problem sets in which one needs to distinguish diameter 2 from 4 are inspired by a combinatorial $(\times, 3/2)$-approximation in a sequential setting by Aingworth et. al. [ACIM99]. The main idea behind this approximation is to distinguish diameter 2 from 4. This part was transferred to the distributed setting in [HW12].

Two-party communication complexity was introduced by Andy Yao in [Yao79]. Later, Yao received the Turing Award. A nice introduction to communication complexity covering techniques such as fooling-sets is the book by Nisan and Kushilevitz [KN97].

This chapter was written in collaboration with Stephan Holzer.

Bibliography

[ACIM99] D. Aingworth, C. Chekuri, P. Indyk, and R. Motwani. Fast Estimation of Diameter and Shortest Paths (Without Matrix Multiplica-

tion). *SIAM Journal on Computing (SICOMP)*, 28(4):1167–1181, 1999.

[AGM91] N. Alon, Z. Galil, and O. Margalit. On the exponent of the all pairs shortest path problem. In *Proceedings of the 32nd Annual IEEE Symposium on Foundations of Computer Science (FOCS)*, pages 569–575, 1991.

[AMGN92] N. Alon, O. Margalit, Z. Galilt, and M. Naor. Witnesses for Boolean Matrix Multiplication and for Shortest Paths. In *Proceedings of the 33rd Annual Symposium on Foundations of Computer Science (FOCS)*, pages 417–426. IEEE Computer Society, 1992.

[AR78] J.M. Abram and IB Rhodes. A decentralized shortest path algorithm. In *Proceedings of the 16th Allerton Conference on Communication, Control and Computing (Allerton)*, pages 271–277, 1978.

[BVW08] G.E. Blelloch, V. Vassilevska, and R. Williams. A New Combinatorial Approach for Sparse Graph Problems. In *Proceedings of the 35th international colloquium on Automata, Languages and Programming, Part I (ICALP)*, pages 108–120. Springer-Verlag, 2008.

[Che82] C.C. Chen. A distributed algorithm for shortest paths. *IEEE Transactions on Computers (TC)*, 100(9):898–899, 1982.

[CM82] K.M. Chandy and J. Misra. Distributed computation on graphs: Shortest path algorithms. *Communications of the ACM (CACM)*, 25(11):833–837, 1982.

[CW90] D. Coppersmith and S. Winograd. Matrix multiplication via arithmetic progressions. *Journal of symbolic computation (JSC)*, 9(3):251–280, 1990.

[DHK+11] A. Das Sarma, S. Holzer, L. Kor, A. Korman, D. Nanongkai, G. Pandurangan, D. Peleg, and R. Wattenhofer. Distributed Verification and Hardness of Distributed Approximation. *Proceedings of the 43rd annual ACM Symposium on Theory of Computing (STOC)*, 2011.

[FHW12] S. Frischknecht, S. Holzer, and R. Wattenhofer. Networks Cannot Compute Their Diameter in Sublinear Time. In *Proceedings of the 23rd annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 1150–1162, January 2012.

[HW12] Stephan Holzer and Roger Wattenhofer. Optimal Distributed All Pairs Shortest Paths and Applications. In *PODC*, page to appear, 2012.

[KN97] E. Kushilevitz and N. Nisan. *Communication complexity*. Cambridge University Press, 1997.

[Lyn96] Nancy A. Lynch. *Distributed Algorithms*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1996.

[MRR80] J. McQuillan, I. Richer, and E. Rosen. The new routing algorithm for the ARPANET. *IEEE Transactions on Communications (TC)*, 28(5):711–719, 1980.

[MS79] P. Merlin and A. Segall. A failsafe distributed routing protocol. *IEEE Transactions on Communications (TC)*, 27(9):1280–1287, 1979.

[PRT12] David Peleg, Liam Roditty, and Elad Tal. Distributed Algorithms for Network Diameter and Girth. In *ICALP*, page to appear, 2012.

[Sei95] R. Seidel. On the all-pairs-shortest-path problem in unweighted undirected graphs. *Journal of Computer and System Sciences (JCSS)*, 51(3):400–403, 1995.

[SS80] M. Schwartz and T. Stern. Routing techniques used in computer communication networks. *IEEE Transactions on Communications (TC)*, 28(4):539–552, 1980.

[SZ99] A. Shoshan and U. Zwick. All pairs shortest paths in undirected graphs with integer weights. In *Proceedings of the 40th Annual IEEE Symposium on Foundations of Computer Science (FOCS)*, pages 605–614. IEEE, 1999.

[Taj77] W.D. Tajibnapis. A correctness proof of a topology information maintenance protocol for a distributed computer network. *Communications of the ACM (CACM)*, 20(7):477–485, 1977.

[Tou80] S. Toueg. An all-pairs shortest-paths distributed algorithm. *Tech. Rep. RC 8327, IBM TJ Watson Research Center, Yorktown Heights, NY 10598, USA*, 1980.

[Wil12] V.V. Williams. Multiplying Matrices Faster Than Coppersmith-Winograd. *Proceedings of the 44th annual ACM Symposium on Theory of Computing (STOC)*, 2012.

[Yao79] A.C.C. Yao. Some complexity questions related to distributive computing. In *Proceedings of the 11th annual ACM symposium on Theory of computing (STOC)*, pages 209–213. ACM, 1979.

Chapter 12

Wireless Protocols

Wireless communication was one of the major success stories of the last decades. Today, different wireless standards such as wireless local area networks (WLAN) are omnipresent. In some sense, from a distributed computing viewpoint wireless networks are quite simple, as they cannot form arbitrary network topologies. Simplistic models of wireless networks include geometric graph models such as the so-called unit disk graph. Modern models are more robust: The network graph is restricted, e.g., the total number of neighbors of a node which are not adjacent is likely to be small. This observation is hard to capture with purely geometric models, and motivates more advanced network connectivity models such as bounded growth or bounded independence.

However, on the other hand, wireless communication is also more difficult than standard message passing, as for instance nodes are not able to transmit a different message to each neighbor at the same time. And if two neighbors are transmitting at the same time, they interfere, and a node may not be able to decipher anything.

In this chapter we deal with the distributed computing principles of wireless communication: We make the simplifying assumption that all n nodes are in the communication range of each other, i.e., the network graph is a clique. Nodes share a synchronous time, in each time slot a node can decide to either transmit or receive (or sleep). However, two or more nodes transmitting in the same time slot will cause interference. Transmitting nodes are never aware if there is interference because they cannot simultaneously transmit and receive.

12.1 Basics

The basic communication protocol in wireless networks is the medium access control (MAC) protocol. Unfortunately it is difficult to claim that one MAC protocol is better than another, because it all depends on the parameters, such as the network topology, the channel characteristics, or the traffic pattern. When it comes to the principles of wireless protocols, we usually want to achieve much simpler goals. One basic and important question is the following: How long does it take until one node can transmit successfully, without interference? This question is often called the wireless leader election problem (Chapter 2), with the node transmitting alone being the leader.

Clearly, we can use node IDs to solve leader election, e.g., a node with ID i transmits in time slot i. However, this may be incredibly slow. There are better deterministic solutions, but by and large the best and simplest algorithms are randomized.

Throughout this chapter, we use a random variable X to denote the number of nodes transmitting in a given slot.

Algorithm 12.1 Slotted Aloha

```
1: Every node v executes the following code:
2: repeat
3:     transmit with probability 1/n
4: until one node has transmitted alone
```

Theorem 12.2. *Using Algorithm 12.1 allows one node to transmit alone (become a leader) after expected time e.*

Proof. The probability for success, i.e., only one node transmitting is

$$Pr[X = 1] = n \cdot \frac{1}{n} \cdot \left(1 - \frac{1}{n}\right)^{n-1} \approx \frac{1}{e},$$

where the last approximation is a result from Theorem 12.29 for sufficiently large n. Hence, if we repeat this process e times, we can expect one success. □

Remarks:

- The origin of the name is the ALOHAnet which was developed at the University of Hawaii.
- How does the leader know that it is the leader? One simple solution is a "distributed acknowledgment". The nodes just continue Algorithm 12.1, including the ID of the the leader in their transmission. So the leader learns that it is the leader.
- One more problem?! Indeed, node v which managed to transmit the acknowledgment (alone) is the only remaining node which does not know that the leader knows that it is the leader. We can fix this by having the leader acknowledge v's successful acknowledgment.
- One can also imagine an unslotted time model. In this model two messages which overlap partially will interfere and no message is received. As everything in this chapter, Algorithm 12.1 also works in an unslotted time model, with a factor 2 penalty, i.e., the probability for a successful transmission will drop from $\frac{1}{e}$ to $\frac{1}{2e}$. Essentially, each slot is divided into t small time slots with $t \to \infty$ and the nodes start a new t-slot long transmission with probability $\frac{1}{2nt}$.

12.2 Initialization

Sometimes we want the n nodes to have the IDs $\{1, 2, \ldots, n\}$. This process is called initialization. Initialization can for instance be used to allow the nodes to transmit one by one without any interference.

12.2.1 Non-Uniform Initialization

Theorem 12.3. *If the nodes know n, we can initialize them in $\mathcal{O}(n)$ time slots.*

Proof. We repeatedly elect a leader using e.g., Algorithm 12.1. The leader gets the next free number and afterwards leaves the process. We know that this works with probability $1/e$. The expected time to finish is hence $e \cdot n$. □

Remarks:

- But this algorithm requires that the nodes know n in order to give them IDs from $1, \ldots, n$! For a more realistic scenario we need a uniform algorithm, i.e, the nodes do not know n.

12.2.2 Uniform Initialization with CD

Definition 12.4 (Collision Detection, CD)**.** *Two or more nodes transmitting concurrently is called interference. In a system with collision detection, a* receiver *can distinguish interference from nobody transmitting. In a system without collision detection, a receiver cannot distinguish the two cases.*

The main idea of the algorithm is to partition nodes iteratively into sets. Each set is identified by a label (a bitstring), and by storing one such bitstring, each node knows in which set it currently is. Initially, all nodes are in a single set, identified by the empty bitstring. This set is then partitioned into two *non-empty* sets, identified by '0' and '1'. In the same way, all sets are iteratively partitioned into two non-empty sets, as long as a set contains more than one node. If a set contains only a single node, this node receives the next free ID. The algorithm terminates once every node is alone in its set. Note that this partitioning process iteratively creates a binary tree which has exactly one node in the set at each leaf, and thus has n leaves.

Algorithm 12.5 Initialization with Collision Detection

```
1:  Every node v executes the following code:
2:  nextId := 0
3:  myBitstring := ''                    ◁ initialize to empty string
4:  bitstringsToSplit := ['']            ◁ a queue with sets to split

5:  while bitstringsToSplit is not empty do
6:    b := bitstringsToSplit.pop()

7:    repeat
8:      if b = myBitstring then
9:        choose r uniformly at random from {0, 1}
10:       in the next two time slots:
11:       transmit in slot r, and listen in other slot
12:     else
13:       it is not my bitstring, just listen in both slots
14:     end if
15:   until there was at least 1 transmission in both slots
16:   if b = myBitstring then
17:     myBitstring := myBitstring + r     ◁ append bit r
18:   end if

19:   for r ∈ {0, 1} do
20:     if some node u transmitted alone in slot r then
21:       node u becomes ID nextId and becomes passive
22:       nextId := nextId + 1
23:     else
24:       bitstringsToSplit.push(b + r)
25:     end if
26:   end for
27: end while
```

Remarks:

- In line 20 a transmitting node needs to know whether it was the only one transmitting. This is achievable in several ways, for instance by adding an acknowledgement round. To notify a node v that it has transmitted alone in round r, every node that was silent in round r sends an acknowledgement in round $r + 1$, while v is silent. If v hears a message or interference in $r + 1$, it knows that it transmitted alone in round r.

Theorem 12.6. *Algorithm 12.5 correctly initializes n nodes in expected time $\mathcal{O}(n)$.*

Proof. A successful split is defined as a split in which both subsets are non-empty. We know that there are exactly $n - 1$ successful splits because we have a binary tree with n leaves and $n - 1$ inner nodes. Let us now calculate the probability for creating two non-empty sets from a set of size $k \geq 2$ as

$$Pr[1 \leq X \leq k-1] = 1 - Pr[X = 0] - Pr[X = k] = 1 - \frac{1}{2^k} - \frac{1}{2^k} \geq \frac{1}{2}.$$

Thus, in expectation we need $\mathcal{O}(n)$ splits. □

Remarks:

- What if we do not have collision detection?

12.2.3 Uniform Initialization without CD

Let us assume that we have a special node ℓ (leader) and let S denote the set of nodes which want to transmit. We now split every time slot from Algorithm 12.5 into two time slots and use the leader to help us distinguish between silence and noise. In the first slot every node from the set S transmits, in the second slot the nodes in $S \cup \{\ell\}$ transmit. This gives the nodes sufficient information to distinguish the different cases (see Table 12.7).

	nodes in S transmit	nodes in $S \cup \{\ell\}$ transmit
$\lvert S\rvert = 0$	✗	✔
$\lvert S\rvert = 1, S = \{\ell\}$	✔	✔
$\lvert S\rvert = 1, S \neq \{\ell\}$	✔	✗
$\lvert S\rvert \geq 2$	✗	✗

Table 12.7: Using a leader to distinguish between noise and silence: ✗ represents noise/silence, ✔ represents a successful transmission.

Remarks:

- As such, Algorithm 12.5 works also without CD, with only a factor 2 overhead.
- More generally, a leader immediately brings CD to any protocol.
- This protocol has an important real life application, for instance when checking out a shopping cart with items which have RFID tags.
- But how do we determine such a leader? And how long does it take until we are "sure" that we have one? Let us repeat the notion of *with high probability*.

12.3 Leader Election

12.3.1 With High Probability

Definition 12.8 (With High Probability)**.** *Some probabilistic event is said to occur* with high probability (w.h.p.)*, if it happens with a probability $p \geq 1 - 1/n^c$, where c is a constant. The constant c may be chosen arbitrarily, but it is considered constant with respect to Big-O notation.*

Theorem 12.9. *Algorithm 12.1 elects a leader w.h.p. in $\mathcal{O}(\log n)$ time slots.*

Proof. The probability for not electing a leader after $c \cdot \log n$ time slots, i.e., $c \log n$ slots without a successful transmission is

$$\left(1 - \frac{1}{e}\right)^{c \ln n} = \left(1 - \frac{1}{e}\right)^{e \cdot c' \ln n} \leq \frac{1}{e^{\ln n \cdot c'}} = \frac{1}{n^{c'}}.$$

□

Remarks:

- What about uniform algorithms, i.e. the number of nodes n is not known?

12.3.2 Uniform Leader Election

Algorithm 12.10 Uniform leader election

```
1: Every node v executes the following code:
2: for k = 1, 2, 3, ... do
3:   for i = 1 to ck do
4:     transmit with probability p := 1/2^k
5:     if node v was the only node which transmitted then
6:       v becomes the leader
7:       break
8:     end if
9:   end for
10: end for
```

Theorem 12.11. *By using Algorithm 12.10 it is possible to elect a leader w.h.p. in $\mathcal{O}(\log^2 n)$ time slots if n is not known.*

Proof. Let us briefly describe the algorithm. The nodes transmit with probability $p = 2^{-k}$ for ck time slots for $k = 1, 2, \ldots$. At first p will be too high and hence there will be a lot of interference. But after $\log n$ phases, we have $k \approx \log n$ and thus the nodes transmit with probability $\approx \frac{1}{n}$. For simplicity's sake, let us assume that n is a power of 2. Using the approach outlined above, we know that after $\log n$ iterations, we have $p = \frac{1}{n}$. Theorem 12.9 yields that we can elect a leader w.h.p. in $\mathcal{O}(\log n)$ slots. Since we have to try $\log n$ estimates until $k \approx n$, the total runtime is $\mathcal{O}(\log^2 n)$. □

Remarks:

- Note that our proposed algorithm has not used collision detection. Can we solve leader election faster in a uniform setting with collision detection?

Algorithm 12.12 Uniform leader election with CD

```
1: Every node v executes the following code:
2: repeat
3:    transmit with probability 1/2
4:    if at least one node transmitted then
5:       all nodes that did not transmit quit the protocol
6:    end if
7: until one node transmits alone
```

12.3.3 Fast Leader Election with CD

Theorem 12.13. *With collision detection we can elect a leader using Algorithm 12.12 w.h.p. in $\mathcal{O}(\log n)$ time slots.*

Proof. The number of active nodes k is monotonically decreasing and always greater than 1 which yields the correctness. A slot is called successful if at most half the active nodes transmit. We can assume that $k \geq 2$ since otherwise we would have already elected a leader. We can calculate the probability that a time slot is successful as

$$Pr\left[1 \leq X \leq \left\lceil \frac{k}{2} \right\rceil\right] = P\left[X \leq \left\lceil \frac{k}{2} \right\rceil\right] - Pr[X = 0] \geq \frac{1}{2} - \frac{1}{2^k} \geq \frac{1}{4}.$$

Since the number of active nodes at least halves in every successful time slot, $\log n$ successful time slots are sufficient to elect a leader. Now let Y be a random variable which counts the number of successful time slots after $8 \cdot c \cdot \log n$ time slots. The expected value is $E[Y] \geq 8 \cdot c \cdot \log n \cdot \frac{1}{4} \geq 2 \cdot c \cdot \log n$. Since all those time slots are independent from each other, we can apply a Chernoff bound (see Theorem 12.28) with $\delta = \frac{1}{2}$ which states

$$Pr[Y < (1-\delta)E[Y]] \leq e^{-\frac{\delta^2}{2}E[Y]} \leq e^{-\frac{1}{8}\cdot 2c\log n} \leq n^{-\alpha}$$

for any constant α. □

Remarks:

- Can we be even faster?

12.3.4 Even Faster Leader Election with CD

Let us first briefly describe an algorithm for this. In the first phase the nodes transmit with probability $1/2^{2^0}, 1/2^{2^1}, 1/2^{2^2}, \ldots$ until no node transmits. This yields a first approximation on the number of nodes. Afterwards, a binary search is performed to determine an even better approximation of n. Finally, the third phase finds a constant approximation of n using a biased random walk. The algorithm stops in any case as soon as only one node is transmitting, which will become the leader.

Lemma 12.15. *If $j > \log n + \log\log n$, then $Pr[X > 1] \leq \frac{1}{\log n}$.*

Algorithm 12.14 Fast uniform leader election

```
1: i := 1
2: repeat
3:    i := 2 · i
4:    transmit with probability 1/2^i
5: until no node transmitted
   {End of Phase 1}
6: l := 2^{i/2}
7: u := 2^i
8: while l + 1 < u do
9:    j := ⌈(l+u)/2⌉
10:   transmit with probability 1/2^j
11:   if no node transmitted then
12:      u := j
13:   else
14:      l := j
15:   end if
16: end while
   {End of Phase 2}
17: k := u
18: repeat
19:   transmit with probability 1/2^k
20:   if no node transmitted then
21:      k := k − 1
22:   else
23:      k := k + 1
24:   end if
25: until exactly one node transmitted
```

Proof. The nodes transmit with probability $1/2^j < 1/2^{\log n + \log\log n} = \frac{1}{n\log n}$. The expected number of nodes transmitting is $E[X] = \frac{n}{n\log n}$. Using Markov's inequality (see Theorem 12.27) yields $Pr[X > 1] \leq Pr[X > E[X] \cdot \log n] \leq \frac{1}{\log n}$. □

Lemma 12.16. *If $j < \log n - \log\log n$, then $P[X = 0] \leq \frac{1}{n}$.*

Proof. The nodes transmit with probability $1/2^j > 1/2^{\log n - \log\log n} = \frac{\log n}{n}$. Thus, the probability that a node is silent is at most $1 - \frac{\log n}{n}$. Hence, the probability for a silent time slot, i.e., $Pr[X = 0]$, is at most $(1 - \frac{\log n}{n})^n = e^{-\log n} = \frac{1}{n}$. □

Corollary 12.17. *If $i > 2\log n$, then $Pr[X > 1] \leq \frac{1}{\log n}$.*

Proof. This follows from Lemma 12.15 since the deviation in this corollary is even larger. □

Corollary 12.18. *If $i < \frac{1}{2}\log n$, then $P[X = 0] \leq \frac{1}{n}$.*

Proof. This follows from Lemma 12.16 since the deviation in this corollary is even larger. □

Lemma 12.19. *Let v be such that $2^{v-1} < n \leq 2^v$, i.e., $v \approx \log n$. If $k > v+2$, then $Pr[X > 1] \leq \frac{1}{4}$.*

Proof. Markov's inequality yields

$$Pr[X > 1] = Pr\left[X > \frac{2^k}{n}E[X]\right] < Pr[X > \frac{2^k}{2^v}E[X]] < Pr[X > 4E[X]] < \frac{1}{4}.$$

□

Lemma 12.20. *If $k < v-2$, then $P[X = 0] \leq \frac{1}{4}$.*

Proof. A similar analysis is possible to upper bound the probability that a transmission fails if our estimate is too small. We know that $k \leq v-2$ and thus

$$Pr[X = 0] = \left(1 - \frac{1}{2^k}\right)^n < e^{-\frac{n}{2^k}} < e^{-\frac{2^{v-1}}{2^k}} < e^{-2} < \frac{1}{4}.$$

□

Lemma 12.21. *If $v-2 \leq k \leq v+2$, then the probability that exactly one node transmits is constant.*

Proof. The transmission probability is $p = \frac{1}{2^{v \pm \Theta(1)}} = \Theta(1/n)$, and the lemma follows with a slightly adapted version of Theorem 12.2.

□

Lemma 12.22. *With probability $1 - \frac{1}{\log n}$ we find a leader in phase 3 in $\mathcal{O}(\log \log n)$ time.*

Proof. For any k, because of Lemmas 12.19 and 12.20, the random walk of the third phase is biased towards the good area. One can show that in $\mathcal{O}(\log \log n)$ steps one gets $\Omega(\log \log n)$ good transmissions. Let Y denote the number of times exactly one node transmitted. With Lemma 12.21 we obtain $E[Y] = \Omega(\log \log n)$. Now a direct application of a Chernoff bound (see Theorem 12.28) yields that these transmissions elect a leader with probability $1 - \frac{1}{\log n}$. □

Theorem 12.23. *The Algorithm 12.14 elects a leader with probability of at least $1 - \frac{\log \log n}{\log n}$ in time $\mathcal{O}(\log \log n)$.*

Proof. From Corollary 12.17 we know that after $\mathcal{O}(\log \log n)$ time slots, the first phase terminates. Since we perform a binary search on an interval of size $\mathcal{O}(\log n)$, the second phase also takes at most $\mathcal{O}(\log \log n)$ time slots. For the third phase we know that $\mathcal{O}(\log \log n)$ slots are sufficient to elect a leader with probability $1 - \frac{1}{\log n}$ by Lemma 12.22. Thus, the total runtime is $\mathcal{O}(\log \log n)$.

Now we can combine the results. We know that the error probability for every time slot in the first two phases is at most $\frac{1}{\log n}$. Using a union bound (see Theorem 12.26), we can upper bound the probability that no error occurred by $\frac{\log \log n}{\log n}$. Thus, we know that after phase 2 our estimate is at most $\log \log n$ away from $\log n$ with probability of at least $1 - \frac{\log \log n}{\log n}$. Hence, we can apply Lemma 12.22 and thus successfully elect a leader with probability of at least $1 - \frac{\log \log n}{\log n}$ (again using a union bound) in time $\mathcal{O}(\log \log n)$.

□

Remarks:

- Tightening this analysis a bit more, one can elect a leader with probability $1 - \frac{1}{\log n}$ in time $\log \log n + o(\log \log n)$.
- Can we be even faster?

12.3.5 Lower Bound

Theorem 12.24. *Any uniform protocol that elects a leader with probability of at least $1 - \frac{1}{2}^t$ must run for at least t time slots.*

Proof. Consider a system with only 2 nodes. The probability that exactly one transmits is at most

$$Pr[X = 1] = 2p \cdot (1-p) \leq \frac{1}{2}.$$

Thus, after t time slots the probability that a leader was elected is at most $1 - \frac{1}{2}^t$. □

Remarks:

- Setting $t = \log \log n$ shows that Algorithm 12.14 is almost tight.

12.3.6 Uniform Asynchronous Wakeup without CD

Until now we have assumed that all nodes start the algorithm in the same time slot. But what happens if this is not the case? How long does it take to elect a leader if we want a uniform and anonymous (nodes do not have an identifier and thus cannot base their decision on it) algorithm?

Theorem 12.25. *If nodes wake up in an arbitrary (worst-case) way, any algorithm may take $\Omega(n/\log n)$ time slots until a single node can successfully transmit.*

Proof. Nodes must transmit at some point, or they will surely never successfully transmit. With a uniform protocol, every node executes the same code. We focus on the first slot where nodes may transmit. No matter what the protocol is, this happens with probability p. Since the protocol is uniform, p must be a constant, independent of n.

The adversary wakes up $w = \frac{c}{p} \ln n$ nodes in each time slot with some constant c. All nodes woken up in the first time slot will transmit with probability p. We study the event E_1 that exactly one of them transmits in that first time slot. Using the inequality $(1 + t/n)^n \leq e^t$ from Lemma 12.29 we get

$$\begin{aligned} Pr[E_1] &= w \cdot p \cdot (1-p)^{w-1} \\ &= c \ln n \, (1-p)^{\frac{1}{p}(c \ln n - p)} \\ &\leq c \ln n \cdot e^{-c \ln + p} \\ &= c \ln n \cdot n^{-c} e^{p} \\ &= n^{-c} \cdot \mathcal{O}(\log n) \\ &< \frac{1}{n^{c-1}} = \frac{1}{n^{c'}}. \end{aligned}$$

In other words, w.h.p. that time slot will not be successful. Since the nodes cannot distinguish noise from silence, the same argument applies to every set of nodes which wakes up. Let E_α be the event that all n/w time slots will not be successful. Using the inequality $1 - p \leq (1 - p/k)^k$ from Lemma 12.30 we get

$$Pr[E_\alpha] = (1 - Pr(E_1))^{n/w} > \left(1 - \frac{1}{n^{c'}}\right)^{\Theta(n/\log n)} > 1 - \frac{1}{n^{c''}}.$$

In other words, w.h.p. it takes more than n/w time slots until some node can transmit alone.

□

12.4 Useful Formulas

In this chapter we have used several inequalities in our proofs. For simplicity's sake we list all of them in this section.

Theorem 12.26. *Boole's inequality or union bound: For a countable set of events $E_1, E_2, E_3, \ldots$, we have*

$$Pr[\bigcup_i E_i] \leq \sum_i Pr[E_i].$$

Theorem 12.27. *Markov's inequality: If X is any random variable and $a > 0$, then*

$$Pr[|X| \geq a] \leq \frac{E[X]}{a}.$$

Theorem 12.28. *Chernoff bound: Let $Y_1, \ldots, Y_n$ be a independent Bernoulli random variables let $Y := \sum_i Y_i$. For any $0 \leq \delta \leq 1$ it holds*

$$Pr[Y < (1-\delta)E[Y]] \leq e^{-\frac{\delta^2}{2}E[Y]}$$

and for $\delta > 0$

$$Pr[Y \geq (1+\delta) \cdot E[Y]] \leq e^{-\frac{\min\{\delta, \delta^2\}}{3} \cdot E[Y]}$$

Theorem 12.29. *We have*

$$e^t \left(1 - \frac{t^2}{n}\right) \leq \left(1 + \frac{t}{n}\right)^n \leq e^t$$

for all $n \in \mathbb{N}, |t| \leq n$. Note that

$$\lim_{n \to \infty} \left(1 + \frac{t}{n}\right)^n = e^t.$$

Theorem 12.30. *For all p, k such that $0 < p < 1$ and $k \geq 1$ we have*

$$1 - p \leq (1 - p/k)^k.$$

Chapter Notes

The Aloha protocol is presented and analyzed in [Abr70, BAK+75, Abr85]; the basic technique that unslotted protocols are twice as bad a slotted protocols is from [Rob75]. The idea to broadcast in a packet radio network by building a tree was first presented in [TM78, Cap79]. This idea is also used in [HNO99] to initialize the nodes. Willard [Wil86] was the first that managed to elect a leader in $\mathcal{O}(\log \log n)$ time in expectation. Looking more carefully at the success rate, it was shown that one can elect a leader with probability $1 - \frac{1}{\log n}$ in time $\log \log n + o(\log \log n)$ [NO98]. Finally, approximating the number of nodes in the network is analyzed in [JKZ02, CGK05, BKK+16]. The lower bound for probabilistic wake-up is published in [JS02]. In addition to single-hop networks, multi-hop networks have been analyzed, e.g. broadcast [BYGI92, KM98, CR06], or deployment [MvRW06].

This chapter was written in collaboration with Philipp Brandes.

Bibliography

[Abr70] Norman Abramson. THE ALOHA SYSTEM: another alternative for computer communications. In *Proceedings of the November 17-19, 1970, fall joint computer conference*, pages 281–285, 1970.

[Abr85] Norman M. Abramson. Development of the ALOHANET. *IEEE Transactions on Information Theory*, 31(2):119–123, 1985.

[BAK+75] R. Binder, Norman M. Abramson, Franklin Kuo, A. Okinaka, and D. Wax. ALOHA packet broadcasting: a retrospect. In *American Federation of Information Processing Societies National Computer Conference (AFIPS NCC)*, 1975.

[BKK+16] Philipp Brandes, Marcin Kardas, Marek Klonowski, Dominik Pajk, and Roger Wattenhofer. Approximating the Size of a Radio Network in Beeping Model. In *23rd International Colloquium on Structural Information and Communication Complexity, Helsinki, Finland*, July 2016.

[BYGI92] Reuven Bar-Yehuda, Oded Goldreich, and Alon Itai. On the Time-Complexity of Broadcast in Multi-hop Radio Networks: An Exponential Gap Between Determinism and Randomization. *J. Comput. Syst. Sci.*, 45(1):104–126, 1992.

[Cap79] J. Capetanakis. Tree algorithms for packet broadcast channels. *IEEE Trans. Inform. Theory*, 25(5):505–515, 1979.

[CGK05] Ioannis Caragiannis, Clemente Galdi, and Christos Kaklamanis. Basic Computations in Wireless Networks. In *International Symposium on Algorithms and Computation (ISAAC)*, 2005.

[CR06] Artur Czumaj and Wojciech Rytter. Broadcasting algorithms in radio networks with unknown topology. *J. Algorithms*, 60(2):115–143, 2006.

[HNO99] Tatsuya Hayashi, Koji Nakano, and Stephan Olariu. Randomized Initialization Protocols for Packet Radio Networks. In *13th International Parallel Processing Symposium & 10th Symposium on Parallel and Distributed Processing (IPPS/SPDP)*, 1999.

[JKZ02] Tomasz Jurdzinski, Miroslaw Kutylowski, and Jan Zatopianski. Energy-Efficient Size Approximation of Radio Networks with No Collision Detection. In *Computing and Combinatorics (COCOON)*, 2002.

[JS02] Tomasz Jurdzinski and Grzegorz Stachowiak. Probabilistic Algorithms for the Wakeup Problem in Single-Hop Radio Networks. In *International Symposium on Algorithms and Computation (ISAAC)*, 2002.

[KM98] Eyal Kushilevitz and Yishay Mansour. An Omega(D log (N/D)) Lower Bound for Broadcast in Radio Networks. *SIAM J. Comput.*, 27(3):702–712, 1998.

[MvRW06] Thomas Moscibroda, Pascal von Rickenbach, and Roger Wattenhofer. Analyzing the Energy-Latency Trade-off during the Deployment of Sensor Networks. In *25th Annual Joint Conference of the IEEE Computer and Communications Societies (INFOCOM), Barcelona, Spain*, April 2006.

[NO98] Koji Nakano and Stephan Olariu. Randomized O (log log n)-Round Leader Election Protocols in Packet Radio Networks. In *International Symposium on Algorithms and Computation (ISAAC)*, 1998.

[Rob75] Lawrence G. Roberts. ALOHA packet system with and without slots and capture. *SIGCOMM Comput. Commun. Rev.*, 5(2):28–42, April 1975.

[TM78] B. S. Tsybakov and V. A. Mikhailov. Slotted multiaccess packet broadcasting feedback channel. *Problemy Peredachi Informatsii*, 14:32–59, October - December 1978.

[Wil86] Dan E. Willard. Log-Logarithmic Selection Resolution Protocols in a Multiple Access Channel. *SIAM J. Comput.*, 15(2):468–477, 1986.

Chapter 13

Stabilization

A large branch of research in distributed computing deals with fault-tolerance. Being able to tolerate a considerable fraction of failing or even maliciously behaving ("Byzantine") nodes while trying to reach *consensus* (on e.g. the output of a function) among the nodes that work properly is crucial for building reliable systems. However, consensus protocols require that a majority of the nodes remains non-faulty all the time.

Can we design a distributed system that survives transient (short-lived) failures, even if *all* nodes are temporarily failing? In other words, can we build a distributed system that *repairs itself*?

13.1 Self-Stabilization

Definition 13.1 (Self-Stabilization)**.** *A distributed system is* self-stabilizing *if, starting from an arbitrary state, it is guaranteed to converge to a legitimate state. If the system is in a legitimate state, it is guaranteed to remain there, provided that no further faults happen. A state is* legitimate *if the state satisfies the specifications of the distributed system.*

Remarks:

- What kind of transient failures can we tolerate? An adversary can crash nodes, or make nodes behave Byzantine. Indeed, temporarily an adversary can do harm in even worse ways, e.g. by corrupting the volatile memory of a node (without the node noticing – not unlike the movie Memento), or by corrupting messages on the fly (without anybody noticing). However, as all failures are transient, eventually all nodes must work correctly again, that is, crashed nodes get resurrected, Byzantine nodes stop being malicious, messages are being delivered reliably, and the memory of the nodes is secure.

- Clearly, the read only memory (ROM) must be taboo at all times for the adversary. No system can repair itself if the program code itself or constants are corrupted. The adversary can only corrupt the variables in the volatile random access memory (RAM).

Definition 13.2 (Time Complexity)**.** *The time complexity of a self-stabilizing system is the time that passed after the last (transient) failure until the system has converged to a legitimate state again, staying legitimate.*

Remarks:

- Self-stabilization enables a distributed system to recover from a transient fault regardless of its nature. A self-stabilizing system does not have to be initialized as it eventually (after convergence) will behave correctly.

- One of the first self-stabilizing algorithms was Dijkstra's token ring network. A token ring is an early form of a local area network where nodes are arranged in a ring, communicating by a token. The system is correct if there is exactly one token in the ring. Let's have a look at a simple solution. Given an oriented ring, we simply call the clockwise neighbor parent (p), and the counterclockwise neighbor child (c). Also, there is a leader node v_0. Every node v is in a state $S(v) \in \{0, 1, \ldots, n\}$, perpetually informing its child about its state. The token is implicitly passed on by nodes switching state. Upon noticing a change of the parent state $S(p)$, node v executes the following code:

Algorithm 13.3 Self-stabilizing Token Ring

```
1: if v = v_0 then
2:   if S(v) = S(p) then
3:     S(v) := S(v) + 1 (mod n)
4:   end if
5: else
6:   S(v) := S(p)
7: end if
```

Theorem 13.4. *Algorithm 13.3 stabilizes correctly.*

Proof: As long as some nodes or edges are faulty, anything can happen. In self-stabilization, we only consider the system after all faults already have happened (at time t_0, however starting in an arbitrary state).

Every node apart from leader v_0 will always attain the state of its parent. It may happen that one node after the other will learn the current state of the leader. In this case the system stabilizes after the leader increases its state at most n time units after time t_0. It may however be that the leader increases its state even if the system is not stable, e.g. because its parent or parent's parent accidentally had the same state at time t_0.

The leader will increase its state possibly multiple times without reaching stability, however, at some point the leader will reach state s, a state that no other node had at time t_0. (Since there are n nodes and n states, this will eventually happen.) At this point the system must stabilize because the leader cannot push for $s + 1 \pmod n$ until every node (including its parent) has s.

After stabilization, there will always be only one node changing its state, i.e., the system remains in a legitimate state.

□

Remarks:

- Although one might think the time complexity of the algorithm is quite bad, it is asymptotically optimal.
- It can be a lot of fun designing self-stabilizing algorithms. Let us try to build a system, where the nodes organize themselves as a maximal independent set (MIS, Chapter 7):

Algorithm 13.5 Self-stabilizing MIS

Require: Node IDs

Every node v executes the following code:

1: **do atomically**
2: Leave MIS if a neighbor with a larger ID is in the MIS
3: Join MIS if no neighbor with larger ID joins MIS
4: Send (node ID, MIS or not MIS) to all neighbors
5: **end do**

Remarks:

- Note that the main idea of Algorithm 13.5 is from Algorithm 7.3, Chapter 7.
- As long as some nodes are faulty, anything can happen: Faulty nodes may for instance decide to join the MIS, but report to their neighbors that they did not join the MIS. Similarly messages may be corrupted during transport. As soon as the system (nodes, messages) is correct, however, the system will converge to a MIS. (The arguments are the same as in Chapter 7).
- Self-stabilizing algorithms always run in an infinite loop, because transient failures can hit the system at any time. Without the infinite loop, an adversary can always corrupt the solution "after" the algorithm terminated.
- The problem of Algorithm 13.5 is its time complexity, which may be linear in the number of nodes. This is not very exciting. We need something better! Since Algorithm 13.5 was just the self-stabilizing variant of the slow MIS Algorithm 7.3, maybe we can hope to "self-stabilize" some of our fast algorithms from Chapter 7?
- Yes, we can! Indeed there is a general transformation that takes any local algorithm (efficient but not fault-tolerant) and turns it into a self-stabilizing algorithm, keeping the same level of efficiency and efficacy. We present the general transformation below.

Theorem 13.6 (Transformation)**.** *We are given a deterministic local algorithm $\mathcal{A}$ that computes a solution of a given problem in k synchronous communication rounds. Using our transformation, we get a self-stabilizing system with time complexity k. In other words, if the adversary does not corrupt the system for k time units, the solution is stable. In addition, if the adversary does not corrupt any node or message closer than distance k from a node u, node u will be stable.*

Proof: In the proof, we present the transformation. First, however, we need to be more formal about the deterministic local algorithm $\mathcal{A}$. In $\mathcal{A}$, each node of the network computes its decision in k phases. In phase i, node u computes its local variables according to its local variables and received messages of the earlier phases. Then node u sends its messages of phase i to its neighbors. Finally node u receives the messages of phase i from its neighbors. The set of local variables of node u in phase i is given by L_u^i. (In the very first phase, node u initializes its local variables with L_u^1.) The message sent from node u to node v in phase i is denoted by $m_{u,v}^i$. Since the algorithm $\mathcal{A}$ is deterministic, node u can compute its local variables L_u^i and messages $m_{u,*}^i$ of phase i from its state of earlier phases, by simply applying functions f_L and f_m. In particular,

$$L_u^i = f_L(u, L_u^{i-1}, m_{*,u}^{i-1}), \text{ for } i > 1, \text{ and} \tag{13.1}$$

$$m_{u,v}^i = f_m(u, v, L_u^i), \text{ for } i \geq 1. \tag{13.2}$$

The self-stabilizing algorithm needs to simulate all the k phases of the local algorithm $\mathcal{A}$ in parallel. Each node u stores its local variables $L_u^1, \ldots, L_u^k$ as well as all messages received $m_{*,u}^1, \ldots, m_{*,u}^k$ in two tables in RAM. For simplicity, each node u also stores all the sent messages $m_{u,*}^1, \ldots, m_{u,*}^k$ in a third table. If a message or a local variable for a particular phase is unknown, the entry in the table will be marked with a special value $\perp$ ("unknown"). Initially, all entries in the table are $\perp$.

Clearly, in the self-stabilizing model, an adversary can choose to change table values at all times, and even reset these values to $\perp$. Our self-stabilizing algorithm needs to constantly work against this adversary. In particular, each node u runs these two procedures constantly:

- For all neighbors: Send each neighbor v a message containing the complete row of messages of algorithm $\mathcal{A}$, that is, send the vector $(m_{u,v}^1, \ldots, m_{u,v}^k)$ to neighbor v. Similarly, if neighbor u receives such a vector from neighbor v, then neighbor u replaces neighbor v's row in the table of incoming messages by the received vector $(m_{v,u}^1, \ldots, m_{v,u}^k)$.
- Because of the adversary, node u must constantly recompute its local variables (including the initialization) and outgoing message vectors using Functions (13.1) and (13.2) respectively.

The proof is by induction. Let $N^i(u)$ be the i-neighborhood of node u (that is, all nodes within distance i of node u). We assume that the adversary has not corrupted any node in $N^k(u)$ since time t_0. At time t_0 all nodes in $N^k(u)$ will check and correct their initialization. Following Equation (13.2), at time t_0 all nodes in $N^k(u)$ will send the correct message entry for the first round $(m_{*,*}^1)$ to all neighbors. Asynchronous messages take at most 1 time unit to be received at a destination. Hence, using the induction with Equations (13.1) and (13.2) it follows that at time $t_0 + i$, all nodes in $N^{k-i}(u)$ have received the correct messages $m_{*,*}^1, \ldots, m_{*,*}^i$. Consequently, at time $t_0 + k$ node u has received all messages of local algorithm $\mathcal{A}$ correctly, and will compute the same result value as in $\mathcal{A}$. □

Remarks:

- Using our transformation (also known as "local checking"), designing self-stabilizing algorithms just turned from art to craft.
- As we have seen, many local algorithms are randomized. This brings two additional problems. Firstly, one may not exactly know how long the algorithm will take. This is not really a problem since we can simply send around all the messages needed, until the algorithm is finished. The transformation of Theorem 13.6 works also if nodes just send all messages that are not $\perp$. Secondly, we must be careful about the adversary. In particular we need to restrict the adversary such that a node can produce a reproducible sufficiently long string of random bits. This can be achieved by storing the sufficiently long string along with the program code in the read only memory (ROM). Alternatively, the algorithm might not store the random bit string in its ROM, but only the seed for a random bit generator. We need this in order to keep the adversary from reshuffling random bits until the bits become "bad", and the expected (or with high probability) efficacy or efficiency guarantees of the original local algorithm $\mathcal{A}$ cannot be guaranteed anymore.
- Since most local algorithms have only a few communication rounds, and only exchange small messages, the memory overhead of the transformation is usually bearable. In addition, information can often be compressed in a suitable way so that for many algorithms message size will remain polylogarithmic. For example, the information of the fast MIS algorithm (Algorithm 7.12) consists of a series of random values (one for each round), plus two boolean values per round. These boolean values represent whether the node joins the MIS, or whether a neighbor of the node joins the MIS. The order of the values tells in which round a decision is made. Indeed, the series of random bits can even be compressed just into the random seed value, and the neighbors can compute the random values of each round themselves.
- There is hope that our transformation as well gives good algorithms for mobile networks, that is for networks where the topology of the network may change. Indeed, for deterministic local approximation algorithms, this is true: If the adversary does not change the topology of a node's k-neighborhood in time k, the solution will locally be stable again.
- For randomized local approximation algorithms however, this is not that simple. Assume for example, that we have a randomized local algorithm for the dominating set problem. An adversary can constantly switch the topology of the network, until it finds a topology for which the random bits (which are not really random because these random bits are in ROM) give a solution with a bad approximation ratio. By defining a weaker adversarial model, we can fix this problem. Essentially, the adversary needs to be oblivious, in the sense that it cannot see the solution. Then it will not be possible for the adversary to restart the random computation if the solution is "too good".

- Self-stabilization is the original approach, and self-organization may be the general theme, but new buzzwords pop up every now and then, e.g. self-configuration, self-management, self-regulation, self-repairing, self-healing, self-optimization, self-adaptivity, or self-protection. Generally all these are summarized as "self-*". One computing giant coined the term "autonomic computing" to reflect the trend of self-managing distributed systems.

13.2 Advanced Stabilization

We finish the chapter with a non-trivial example beyond self-stabilization, showing the beauty and potential of the area: In a small town, every evening each citizen calls all his (or her) friends, asking them whether they will vote for the Democratic or the Republican party at the next election.[1] In our town citizens listen to their friends, and everybody re-chooses his or her affiliation according to the majority of friends.[2] Is this process going to "stabilize" (in one way or another)?

Remarks:

- Is eventually everybody voting for the same party? No.
- Will each citizen eventually stay with the same party? No.
- Will citizens that stayed with the same party for some time, stay with that party forever? No.
- And if their friends also constantly root for the same party? No.
- Will this beast stabilize at all?!? Yes!

Eventually every citizen will either stay with the same party for the rest of her life, or switch her opinion every day.

Theorem 13.7 (Dems & Reps)**.** *Eventually every citizen is rooting for the same party every other day.*

Proof: To prove that the opinions eventually become fixed or cycle every other day, think of each friendship as a pair of (directed) edges, one in each direction. Let us say an edge is currently *bad* if the party of the *advising* friend differs from the next-day's party of the *advised* friend. In other words, the edge is bad if the advised friend did not follow the advisor's opinion (which means that the advisor was in the minority). An edge that is not bad, is *good*.

Consider the out-edges of citizen u on day t, during which (say) u roots for the Democrats. Assume that on day t, g out-edges of u are good, and b out-edges are bad. Note that $g + b$ is the degree of u. Since g out-edges are good, g friends of u root for the Democrats on day $t + 1$. Likewise, b friends of u root for the Republicans on day $t + 1$. In other words, on the evening of day $t + 1$ citizen u will receive g recommendations for Democrats, and b for Republicans. We distinguish two cases:

[1] We are in the US, and as we know from The Simpsons, you "throw your vote away" if you vote for somebody else. As a consequence our example has two parties only.

[2] Assume for the sake of simplicity that everybody has an odd number of friends.

- $g > b$: In this case, citizen u will again root for the Democrats on day $t + 2$. Note that this means, on day $t + 1$, exactly g in-edges of u are good, and exactly b in-edges are bad. In other words, the number of bad out-edges on day t is exactly the number of bad in-edges on day $t + 1$.

- $g < b$: In this case, citizen u will root for the Republicans on day $t + 2$. Please note that on day $t+1$, exactly b in-edges of u are good, and exactly g in-edges are bad. In other words, the number of bad out-edges on day t was exactly the number of good in-edges on day $t + 1$ (and vice versa). This means that the number of bad out-edges on day t is strictly larger than the number of bad in-edges on day $t + 1$.

We can summarize these two cases by the following observation. If a citizen u votes for the same party on day t as on day $t + 2$, the number of her bad out-edges on day t is the same as the number of her bad in-edges on day $t + 1$. If a citizen u votes for different parties on the days t and $t + 2$, the number of her bad out-edges on day t is strictly larger than the number of her bad in-edges on day $t + 1$.

We now account for the total number of bad edges. We denote the total number of bad out-edges on day t with BO_t and by the total number of bad in-edges on day t with BI_t. Using the analysis of the two cases, and summing up for all citizens, we know that $BO_t \geq BI_{t+1}$. Moreover, each out-edge of a citizen is an in-edge for another citizen, hence $BO_t = BI_t$. In fact, if any citizen switches its party from day t to $t + 2$, we know that the total number of bad edges strictly decreases, i.e., $BO_{t+1} = BI_{t+1} < BO_t$. But BO cannot decrease forever. Once $BO_{t+1} = BO_t$, every citizen u votes for the same party on day $t+2$ as u voted on day t, and the system stabilizes in the sense that every citizen will either stick with his or her party forever or flip-flop every day. □

Remarks:

- The model can be generalized considerably by, for example, adding weights to vertices (meaning some citizens' opinions are more important than others), adding weights to edges (meaning the influence between some citizens is stronger than between others), allowing loops (citizens who consider their own current opinions as well), allowing tie-breaking mechanisms, and even allowing different thresholds for party changes.

- How long does it take until the system stabilizes?

- Some may be reminded of Conway's Game of Life: We are given an infinite two-dimensional grid of cells, each of which is in one of two possible states, *dead* or *alive*. Every cell interacts with its eight neighbors. In each round, the following transitions occur: Any live cell with fewer than two live neighbors dies, as if caused by loneliness. Any live cell with more than three live neighbors dies, as if by overcrowding. Any live cell with two or three live neighbors lives on to the next generation. Any dead cell with exactly three live neighbors is "born" and becomes a live cell. The initial pattern constitutes the "seed" of the system. The first generation is created by applying the above rules simultaneously to every cell in the seed, births and deaths happen simultaneously, and the discrete moment at which this happens is sometimes called a tick. (In other words, each generation is a pure function of the one before.) The rules continue to be applied repeatedly to create further generations. John Conway figured that these rules were enough to generate interesting situations, including "breeders" with create "guns" which in turn create "gliders". As such Life in some sense answers an old question by John von Neumann, whether there can be a simple machine that can build copies of itself. In fact Life is Turing complete, that is, as powerful as any computer.

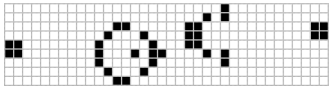


Figure 13.8: A "glider gun"...

Figure 13.9: ... in action.

Chapter Notes

Self-stabilization was first introduced in a paper by Edsger W. Dijkstra in 1974 [Dij74], in the context of a token ring network. It was shown that the ring stabilizes in time $\Theta(n)$. For his work Dijkstra received the 2002 ACM PODC Influential Paper Award. Shortly after receiving the award he passed away. With Dijkstra being such an eminent person in distributed computing (e.g. concurrency, semaphores,mutual exclusion, deadlock, finding shortest paths in graphs, fault-tolerance, self-stabilization), the award was renamed Edsger W. Dijkstra Prize in Distributed Computing. In 1991 Awerbuch et al. showed that any algorithm can be modified into a self-stabilizing algorithm that stabilizes in the same time that is needed to compute the solution from scratch [APSV91].

The Republicans vs. Democrats problem was popularized by Peter Winkler, in his column "Puzzled" [Win08]. Goles et al. already proved in [GO80] that any configuration of any such system with symmetric edge weights will end up in a situation where each citizen votes for the same party every second day.

Winkler additionally proved that the time such a system takes to stabilize is bounded by $\mathcal{O}(n^2)$. Frischknecht et al. constructed a worst case graph which takes $\Omega(n^2/\log^2 n)$ rounds to stabilize [FKW13]. Keller et al. generalized this results in [KPW14], showing that a graph with symmetric edge weights stabilizes in $\mathcal{O}(W(G))$, where $W(G)$ is the sum of edge weights in graph G. They also constructed a weighted graph with exponential stabilization time. Closely related to this puzzle is the well known Game of Life which was described by the mathematician John Conway and made popular by Martin Gardner [Gar70]. In the Game of Life cells can be either dead or alive and change their states according to the number of alive neighbors.

Bibliography

[APSV91] Baruch Awerbuch, Boaz Patt-Shamir, and George Varghese. Self-Stabilization By Local Checking and Correction. In *In Proceedings of IEEE Symposium on Foundations of Computer Science (FOCS)*, 1991.

[Dij74] Edsger W. Dijkstra. Self-stabilizing systems in spite of distributed control. *Communications of the ACM*, 17(11):943–644, November 1974.

[FKW13] Silvio Frischknecht, Barbara Keller, and Roger Wattenhofer. Convergence in (Social) Influence Networks. In *27th International Symposium on Distributed Computing (DISC), Jerusalem, Israel*, October 2013.

[Gar70] M. Gardner. Mathematical Games: The fantastic combinations of John Conway's new solitaire game Life. *Scientific American*, 223:120–123, October 1970.

[GO80] E. Goles and J. Olivos. Periodic behavior of generalized threshold functions. *Discrete Mathematics*, 30:187–189, 1980.

[KPW14] Barbara Keller, David Peleg, and Roger Wattenhofer. How even Tiny Influence can have a Big Impact! In *7th International Conference on Fun with Algorithms (FUN), Lipari Island, Italy*, July 2014.

[Win08] P. Winkler. Puzzled. *Communications of the ACM*, 51(9):103–103, August 2008.

Chapter 14

Labeling Schemes

Imagine you want to repeatedly query a huge graph, e.g., a social or a road network. For example, you might need to find out whether two nodes are connected, or what the distance between two nodes is. Since the graph is so large, you distribute it among multiple servers in your data center.

14.1 Adjacency

Theorem 14.1. *It is possible to assign labels of size $2\log n$ bits to nodes in a tree so that for every pair u, v of nodes, it is easy to tell whether u is adjacent to v by just looking at u and v's labels.*

Proof. Choose a root in the tree arbitrarily so that every non-root node has a parent. The label of each node u consists of two parts: The ID of u (from 1 to n), and the ID of u's parent (or nothing if u is the root). □

Remarks:

- What we have constructed above is called a *labeling scheme*, more precisely a labeling scheme for adjacency in trees. Formally, a labeling scheme is defined as follows.

Definition 14.2. *A* labeling scheme *consists of an* encoder e *and a* decoder d*. The encoder e assigns to each node v a label $e(v)$. The decoder d receives the labels of the nodes in question and returns an answer to some query. The largest size (in bits) of a label assigned to a node is called the* label size *of the labeling scheme.*

Remarks:

- In Theorem 14.1, the decoder receives two node labels $e(u)$ and $e(v)$, and its answer is Yes or No, depending on whether u and v are adjacent or not. The label size is $2\log n$.
- The label size is the complexity measure we are going to focus on in this chapter. The run-time of the encoder and the decoder are two other complexity measures that are studied in the literature.

- There is an interesting connection between labeling schemes for adjacency and so-called *induced-universal graphs*: Let $\mathcal{F}$ be a family of graphs. The graph $U(n)$ is called *n-induced-universal for $\mathcal{F}$* if all $G \in \mathcal{F}$ with at most n nodes appear as a node-induced subgraph in $U(n)$. (A node-induced subgraph of $U(n) = (V, E)$ is any graph that can be obtained by taking a subset V' of V and all edges from E which have both endpoints in V'.)
- In the movie Good Will Hunting, the big open question was to find all graphs of the family of homeomorphically irreducible (non-isomorphic, no node with degree 2) trees with 10 nodes, $\mathcal{T}_{10}$. What is the smallest induced-universal graph for $\mathcal{T}_{10}$?
- If a graph family $\mathcal{F}$ allows a labeling scheme for adjacency with label size $f(n)$, then there are n-induced-universal graphs for $\mathcal{F}$ so that the size of $U(n)$ is at most $2^{f(n)}$. Since the size of $U(n)$ is exponential in f it is interesting to study the label size carefully: If f is $\log n$, the size of $U(n)$ is n, whereas if f is $2\log n$ the size of $U(n)$ becomes n^2!
- What about adjacency in general graphs?

Theorem 14.3. *Any labeling scheme for adjacency in general graphs has a label size of at least $\Omega(n)$ bits.*

Proof. Let $\mathcal{G}_n$ denote the family of graphs with n nodes, and assume there is a labeling scheme for adjacency in graphs from $\mathcal{G}_n$ with label size s. First, we argue that the encoder e must be injective on $\mathcal{G}_n$: Since the labeling scheme is for adjacency, e cannot assign the same labels to two different graphs.

There are 2^s possible labels for any node, and for every $G \in \mathcal{G}_n$ we can choose n of them. Thus, we obtain that

$$|\mathcal{G}_n| \leq \left(\!\!\binom{2^s}{n}\!\!\right) = \binom{2^s + n - 1}{n}$$

Moreover, a graph in $\mathcal{G}_n$ can have at most $\binom{n}{2}$ edges, and thus $|\mathcal{G}_n| \geq 2^{\binom{n}{2}}/n!$ when taking into account that the order of the nodes is irrelevant. Canceling out the $n!$ term and taking the logarithm on both sides of the inequality we conclude that $s \in \Omega(n)$. □

Remarks:

- The lower bound for general graphs is a bit discouraging; we wanted to use labeling schemes for queries on large graphs!
- The situation is less dire if the graph is not arbitrary. For instance, in degree-bounded graphs, in planar graphs, and in trees, the bounds change to $\Theta(\log n)$ bits.
- What about other queries, e.g., distance?
- Next, we will focus on rooted trees.

14.2 Rooted Trees

Theorem 14.4. *There is a $2\log n$ labeling scheme for ancestry, i.e., for two nodes u and v, find out if u is an ancestor of v in the rooted tree T.*

Proof. Traverse the tree with a depth first search, and consider the obtained pre-ordering of the nodes, i.e., enumerate the nodes in the order in which they are first visited. For a node u denote by $l(u)$ the index in the pre-order. Our encoder assigns labels $e(u) = (l(u), r(u))$ to each node u, where $r(u)$ is the largest value $l(v)$ that appears at any node v in the sub-tree rooted at u. With the labels assigned in this manner, we can find out whether u is an ancestor of v by checking if $l(v)$ is contained in the interval $(l(u), r(u)]$. □

Algorithm 14.5 Naïve-Distance-Labeling(T)

```
Let l be the label of the root r of T
Let T_1, ..., T_δ be the sub-trees rooted at each of the δ children of r
for i = 1, ..., δ do
    The root of T_i gets the label obtained by appending i to l
    Naïve-Distance-Labeling(T_i)
end for
```

Theorem 14.6. *There is an $\mathcal{O}(n \log n)$ labeling scheme for distance in trees.*

Proof. Apply the encoder algorithm Naïve-Distance-Labeling(T) to label the tree T. The encoder assigns to every node v a sequence $(l_1, l_2 \dots)$. The length of a sequence $e(v)$ is at most n, and each entry in the sequence requires at most $\log n$ bits. A label $(l_1, \dots, l_k)$ of a node v corresponds to a path from r to v in T, and the nodes on the path are labeled $(l_1), (l_1, l_2), (l_1, l_2, l_3)$ and so on. The distance between u and v in T is obtained by reconstructing the paths from $e(u)$ and $e(v)$. □

Remarks:

- We can assign the labels more carefully to obtain a smaller label size. For that, we use the following *heavy-light decomposition*.

Algorithm 14.7 Heavy-Light-Decomposition(T)

```
Node r is the root of T
Let T_1, ..., T_δ be the sub-trees rooted at each of the δ children of r
Let T_max be a largest tree in {T_1, ..., T_δ} in terms of number of nodes
Mark the edge (r, T_max) as heavy
Mark all edges to other children of r as light
Assign the names 1, ..., δ − 1 to the light edges of r
for i = 1, ..., δ do
    Heavy-Light-Decomposition(T_i)
end for
```

Theorem 14.8. *There is an $\mathcal{O}(\log^2 n)$ labeling scheme for distance in trees.*

Proof. For our proof, use Heavy-Light-Decomposition(T) to partition T's edges into heavy and light edges. All heavy edges form a collection of paths, called the *heavy paths*. Moreover, every node is reachable from the root through a sequence of heavy paths connected with light edges. Instead of storing the whole path to reach a node, we only store the information about heavy paths and light edges that were taken to reach a node from the root.

For instance, if node u can be reached by first using 2 heavy edges, then the 7^{th} light edge, then 3 heavy edges, and then the light edges 1 and 4, then we assign to v the label $(\mathbf{2}, 7, \mathbf{3}, 1, 4)$. For any node u, the path $p(u)$ from the root to u is now specified by the label. The distance between any two nodes can be computed using the paths.

Since every parent has at most $\Delta < n$ children, the name of a light edge has at most $\log n$ bits. The size (number of nodes in the sub-tree) of a light child is at most half the size of its parent, so a path can have at most $\log n$ light edges. Between any two light edges, there could be a heavy path, so we can have up to $\log n$ heavy paths in a label. The length of such a heavy path can be described with $\log n$ bits as well, since no heavy path has more than n nodes. Altogether we therefore need at most $\mathcal{O}(\log^2 n)$ bits. □

Remarks:

- One can show that any labeling scheme for distance in trees needs to use labels of size at least $\Omega(\log^2 n)$.
- The distance encoder from Theorem 14.8 also supports decoders for other queries. To check for ancestry, it therefore suffices to check if $p(u)$ is a prefix of $p(v)$ or vice versa.
- The nearest common ancestor is the last node that is on both $p(u)$ and $p(v)$, and the separation level is the length of the path to that node.
- Two nodes are siblings if their distance is 2 but they are not ancestors.
- The heavy-light decomposition can be used to shave off a few bits in other labeling schemes, e.g., ancestry or adjacency.

14.3 Road Networks

Labeling schemes are used to quickly find shortest paths in road networks.

Remarks:

- A naïve approach is to store at every node u the shortest paths to all other nodes v. This requires an impractical amount of memory. For example, the road network for Western Europe has 18 million nodes and 44 million directed edges, and the USA road network has 24 million nodes and 58 million directed edges.
- What if we only store the next node on the shortest path to all targets? In a worst case this stills requires $\Omega(n)$ bits per node. Moreover, answering a single query takes many invocations of the decoder.

- For simplicity, let us focus on answering distance queries only. Even if we only want to know the distance, storing the full table of n^2 distances costs more than 1000TB, too much for storing it in RAM.
- The idea for the encoder is to compute a set S of *hub* nodes that lie on many shortest paths. We then store at each node u only the distance to the hub nodes that appear on shortest paths originating or ending in u.
- Given two labels $e(u)$ and $e(v)$, let $H(u,v)$ denote the set of hub nodes that appear in both labels. The decoder now simply returns $d(u,v) = \min\{\text{dist}(u,h) + \text{dist}(h,v) : h \in H(u,v)\}$, all of which can be computed from the two labels.
- The key in finding a good labeling scheme now lies in finding good hub nodes.

Algorithm 14.9 Naïve-Hub-Labeling(G)

```
Let P be the set of all n^2 shortest paths
while P ≠ ∅ do
    Let h be a node which is on a maximum number of paths in P
    for all paths p = (u, ..., v) ∈ P do
        if h is on p then
            Add h with the distance dist(u, h) to the label of u
            Add h with the distance dist(h, v) to the label of v
            Remove p from P
        end if
    end for
end while
```

Remarks:

- Unfortunately, algorithm 14.9 takes a prohibitively long time to compute.
- Another approach computes the set S as follows. The encoder (Algorithm 14.10) first constructs so-called *shortest path covers*. The node set S_i is a shortest path cover if S_i contains a node on every shortest path of length between 2^{i-1} and 2^i. At node v only the hub nodes in S_i that are within the ball of radius 2^i around v (denoted by $B(v,2^i)$) are stored.

Algorithm 14.10 Hub-Labeling(G)

```
for i = 1, ..., log D do
    Compute the shortest path cover S_i
end for
for all v ∈ V do
    Let F_i(v) be the set S_i ∩ B(v, 2^i)
    Let F(v) be the set F_1(v) ∪ F_2(v) ∪ ...
    The label of v consists of the nodes in F(v), with their distance to v
end for
```


Remarks:

- The size of the shortest path covers will determine how space efficient the solution will be. It turns out that real-world networks allow for small shortest path covers: The parameter h is the so-called *highway dimension* of G, is defined as $h = \max_{i,v} F_i(v)$, and h is conjectured to be small for road networks.
- Computing S_i with a minimal number of hubs is NP-hard, but one can compute a $\mathcal{O}(\log n)$ approximation of S_i in polynomial time. Consequently, the label size is at most $\mathcal{O}(h \log n \log D)$. By ordering the nodes in each label by their ID, the decoder can scan through both node lists in parallel in time $\mathcal{O}(h \log n \log D)$.
- While this approach yields good theoretical bounds, the encoder is *still* too slow in practice. Therefore, before computing the shortest path covers, the graph is contracted by introducing *shortcuts* first.
- Based on this approach a distance query on a continent-sized road network can be answered in less that 1μs on current hardware, orders of magnitude faster than a single random disk access. Storing all the labels requires roughly 20 GB of RAM.
- The method can be extended to support shortest path queries, e.g., by storing the path to/from the hub nodes, or by recursively querying for nodes that lie on the shortest path to the hub.

Chapter Notes

Adjacency labelings were first studied by Breuer and Folkman [BF67]. The $\log n + \mathcal{O}(\log^* n)$ upper bound for trees is due to [AR02] using a clustering technique. In contrast, it was shown that for general graphs the size of universal graphs is at least $2^{(n-1)/2}$! Since graphs of arboricity d can be decomposed into d forests [NW61], the labeling scheme from [AR02] can be used to label graphs of arboricity d with $d \log n + \mathcal{O}(\log n)$ bit labels. For a thorough survey on labeling schemes for rooted trees please check [AHR].

Universal graphs were studied already by Ackermann [Ack37], and later by Erdős, Rényi, and Rado [ER63, Rad64]. The connection between labeling schemes and universal graphs [KNR88] was investigated thoroughly. Our adjacency lower bound follows the presentation in [AKTZ14], which also summarizes recent results in this field of research.

Distance labeling schemes were first studied by Peleg [Pel00]. The notion of highway dimension was introduced by [AFGW10] in an attempt to explain the good performance of many heuristics to speed up shortest path computations, e.g., Transit Node Routing [BFSS07]. Their suggestions to modify the SHARC heuristic [BD08] lead to the hub labeling scheme and were implemented and evaluated [ADGW11], and later refined [DGSW14]. The $\Omega(n)$ label size lower bound for routing (shortest paths) with stretch smaller than 3 is due to [GG01].

This chapter was written in collaboration with Jochen Seidel. Thanks to Noy Rotbart for suggesting the topic.

Bibliography

[Ack37] Wilhelm Ackermann. Die Widerspruchsfreiheit der allgemeinen Mengenlehre. *Mathematische Annalen*, 114(1):305–315, 1937.

[ADGW11] Ittai Abraham, Daniel Delling, Andrew V. Goldberg, and Renato Fonseca F. Werneck. A hub-based labeling algorithm for shortest paths in road networks. In *SEA*, 2011.

[AFGW10] Ittai Abraham, Amos Fiat, Andrew V. Goldberg, and Renato Fonseca F. Werneck. Highway dimension, shortest paths, and provably efficient algorithms. In *SODA*, 2010.

[AHR] Stephen Alstrup, Esben Bistrup Halvorsen, and Noy Rotbart. A survey on labeling schemes for trees. To appear.

[AKTZ14] Stephen Alstrup, Haim Kaplan, Mikkel Thorup, and Uri Zwick. Adjacency labeling schemes and induced-universal graphs. *CoRR*, abs/1404.3391, 2014.

[AR02] Stephen Alstrup and Theis Rauhe. Small induced-universal graphs and compact implicit graph representations. In *FOCS*, 2002.

[BD08] Reinhard Bauer and Daniel Delling. SHARC: fast and robust unidirectional routing. In *ALENEX*, 2008.

[BF67] Melvin A Breuer and Jon Folkman. An unexpected result in coding the vertices of a graph. *Journal of Mathematical Analysis and Applications*, 20(3):583 – 600, 1967.

[BFSS07] Holger Bast, Stefan Funke, Peter Sanders, and Dominik Schultes. Fast routing in road networks with transit nodes. *Science*, 316(5824):566, 2007.

[DGSW14] Daniel Delling, Andrew V. Goldberg, Ruslan Savchenko, and Renato F. Werneck. Hub labels: Theory and practice. In *SEA*, 2014.

[ER63] P. Erdős and A. Rényi. Asymmetric graphs. *Acta Mathematica Academiae Scientiarum Hungarica*, 14(3-4):295–315, 1963.

[GG01] Cyril Gavoille and Marc Gengler. Space-efficiency for routing schemes of stretch factor three. *J. Parallel Distrib. Comput.*, 61(5):679–687, 2001.

[KNR88] Sampath Kannan, Moni Naor, and Steven Rudich. Implicit representation of graphs. In *STOC*, 1988.

[NW61] C. St. J. A. Nash-Williams. Edge-disjoint spanning trees of finite graphs. *J. London Math. Soc.*, 36:445–450, 1961.

[Pel00] David Peleg. Proximity-preserving labeling schemes. *Journal of Graph Theory*, 33(3):167–176, 2000.

[Rad64] Richard Rado. Universal graphs and universal functions. *Acta Arith.*, 9:331–340, 1964.

Chapter 15

Fault-Tolerance & Paxos

How do you create a fault-tolerant distributed system? In this chapter we start out with simple questions, and, step by step, improve our solutions until we arrive at a system that works even under adverse circumstances, Paxos.

15.1 Client/Server

Definition 15.1 (node)**.** *We call a single actor in the system* node*. In a computer network the computers are the nodes, in the classical client-server model both the server and the client are nodes, and so on. If not stated otherwise, the total number of nodes in the system is n.*

Model 15.2 (message passing)**.** *In the* message passing model *we study distributed systems that consist of a set of nodes. Each node can perform local computations, and can send messages to every other node.*

Remarks:

- We start with two nodes, the smallest number of nodes in a distributed system. We have a *client* node that wants to "manipulate" data (e.g., store, update, ...) on a remote *server* node.

Algorithm 15.3 Naïve Client-Server Algorithm

1: Client sends commands one at a time to server

Model 15.4 (message loss)**.** *In the message passing model with* message loss*, for* any *specific message, it is not guaranteed that it will arrive safely at the receiver.*

Remarks:

- A related problem is message corruption, i.e., a message is received but the content of the message is corrupted. In practice, in contrast to message loss, message corruption can be handled quite well, e.g. by including additional information in the message, such as a checksum.

- Algorithm 15.3 does not work correctly if there is message loss, so we need a little improvement.

Algorithm 15.5 Client-Server Algorithm with Acknowledgments

1: Client sends commands one at a time to server
2: Server acknowledges every command
3: If the client does not receive an acknowledgment within a reasonable time, the client resends the command

Remarks:

- Sending commands "one at a time" means that when the client sent command c, the client does not send any new command c' until it received an acknowledgment for c.
- Since not only messages sent by the client can be lost, but also acknowledgments, the client might resend a message that was already received and executed on the server. To prevent multiple executions of the same command, one can add a *sequence number* to each message, allowing the receiver to identify duplicates.
- This simple algorithm is the basis of many reliable protocols, e.g. TCP.
- The algorithm can easily be extended to work with multiple servers: The client sends each command to every server, and once the client received an acknowledgment from each server, the command is considered to be executed successfully.
- What about multiple clients?

Model 15.6 (variable message delay)**.** *In practice, messages might experience different transmission times, even if they are being sent between the same two nodes.*

Remarks:

- Throughout this chapter, we assume the variable message delay model.

Theorem 15.7. *If Algorithm 15.5 is used with multiple clients and multiple servers, the servers might see the commands in different order, leading to an inconsistent state.*

Proof. Assume we have two clients u_1 and u_2, and two servers s_1 and s_2. Both clients issue a command to update a variable x on the servers, initially $x = 0$. Client u_1 sends command $x = x + 1$ and client u_2 sends $x = 2 \cdot x$.

Let both clients send their message at the same time. With variable message delay, it can happen that s_1 receives the message from u_1 first, and s_2 receives the message from u_2 first.[1] Hence, s_1 computes $x = (0 + 1) \cdot 2 = 2$ and s_2 computes $x = (0 \cdot 2) + 1 = 1$. □

[1] For example, u_1 and s_1 are (geographically) located close to each other, and so are u_2 and s_2.

Definition 15.8 (state replication)**.** *A set of nodes achieves* state replication*, if all nodes execute a (potentially infinite) sequence of commands $c_1, c_2, c_3, \ldots$, in the same order.*

Remarks:

- State replication is a fundamental property for distributed systems.
- For people working in the financial tech industry, state replication is often synonymous with the term blockchain. The Bitcoin blockchain we will discuss in Chapter 20 is indeed one way to implement state replication. However, as we will see in all the other chapters, there are many alternative concepts that are worth knowing, with different properties.
- Since state replication is trivial with a single server, we can designate a single server as a *serializer*. By letting the serializer distribute the commands, we automatically order the requests and achieve state replication!

Algorithm 15.9 State Replication with a Serializer

1: Clients send commands one at a time to the serializer
2: Serializer forwards commands one at a time to all other servers
3: Once the serializer received all acknowledgments, it notifies the client about the success

Remarks:

- This idea is sometimes also referred to as *master-slave replication*.
- What about node failures? Our serializer is a single point of failure!
- Can we have a more *distributed* approach of solving state replication? Instead of directly establishing a consistent order of commands, we can use a different approach: We make sure that there is always at most one client sending a command; i.e., we use *mutual exclusion*, respectively *locking*.

Algorithm 15.10 Two-Phase Protocol

Phase 1

1: Client asks all servers for the lock

Phase 2

2: **if** client receives lock from every server **then**
3: Client sends command reliably to each server, and gives the lock back
4: **else**
5: Clients gives the received locks back
6: Client waits, and then starts with Phase 1 again
7: **end if**

Remarks:

- This idea appears in many contexts and with different names, usually with slight variations, e.g. *two-phase locking (2PL)*.
- Another example is the *two-phase commit (2PC)* protocol, typically presented in a database environment. The first phase is called the *preparation* of a transaction, and in the second phase the transaction is either *committed* or *aborted*. The 2PC process is not started at the client but at a designated server node that is called the *coordinator*.
- It is often claimed that 2PL and 2PC provide better consistency guarantees than a simple serializer if nodes can *recover* after crashing. In particular, alive nodes might be kept consistent with crashed nodes, for transactions that started while the crashed node was still running. This benefit was even improved in a protocol that uses an additional phase (3PC).
- The problem with 2PC or 3PC is that they are not well-defined if exceptions happen.
- Does Algorithm 15.10 really handle node crashes well? No! In fact, it is even worse than the simple serializer approach (Algorithm 15.9): Instead of having a only one node which must be available, Algorithm 15.10 requires *all* servers to be responsive!
- Does Algorithm 15.10 also work if we only get the lock from a subset of servers? Is a majority of servers enough?
- What if two or more clients concurrently try to acquire a majority of locks? Do clients have to abandon their already acquired locks, in order not to run into a deadlock? How? And what if they crash before they can release the locks? Do we need a slightly different concept?

15.2 Paxos

Definition 15.11 (ticket)**.** *A* ticket *is a weaker form of a lock, with the following properties:*

- **Reissuable:** *A server can issue a ticket, even if previously issued tickets have not yet been returned.*
- **Ticket expiration:** *If a client sends a message to a server using a previously acquired ticket t, the server will only accept t, if t is the most recently issued ticket.*

Remarks:

- There is no problem with crashes: If a client crashes while holding a ticket, the remaining clients are not affected, as servers can simply issue new tickets.
- Tickets can be implemented with a counter: Each time a ticket is requested, the counter is increased. When a client tries to use a ticket, the server can determine if the ticket is expired.
- What can we do with tickets? Can we simply replace the locks in Algorithm 15.10 with tickets? We need to add at least one additional phase, as only the client knows if a majority of the tickets have been valid in Phase 2.

Algorithm 15.12 Naïve Ticket Protocol

Phase 1

1: Client asks all servers for a ticket

Phase 2

2: **if** a majority of the servers replied **then**
3: Client sends command together with ticket to each server
4: Server stores command only if ticket is still valid, and replies to client
5: **else**
6: Client waits, and then starts with Phase 1 again
7: **end if**

Phase 3

8: **if** client hears a positive answer from a majority of the servers **then**
9: Client tells servers to execute the stored command
10: **else**
11: Client waits, and then starts with Phase 1 again
12: **end if**

Remarks:

- There are problems with this algorithm: Let u_1 be the first client that successfully stores its command c_1 on a majority of the servers. Assume that u_1 becomes very slow just before it can notify the servers (Line 7), and a client u_2 updates the stored command in some servers to c_2. Afterwards, u_1 tells the servers to execute the command. Now some servers will execute c_1 and others c_2!
- How can this problem be fixed? We know that every client u_2 that updates the stored command after u_1 must have used a newer ticket than u_1. As u_1's ticket was accepted in Phase 2, it follows that u_2 must have acquired its ticket after u_1 already stored its value in the respective server.
- Idea: What if a server, instead of only handing out tickets in Phase 1, also notifies clients about its currently stored command? Then, u_2 learns that u_1 already stored c_1 and instead of trying to store c_2, u_2 could support u_1 by also storing c_1. As both clients try to store and execute the same command, the order in which they proceed is no longer a problem.
- But what if not all servers have the same command stored, and u_2 learns multiple stored commands in Phase 1. What command should u_2 support?
- Observe that it is always safe to support the most recently stored command. As long as there is no majority, clients can support any command. However, once there is a majority, clients need to support this value.

- So, in order to determine which command was stored most recently, servers can remember the ticket number that was used to store the command, and afterwards tell this number to clients in Phase 1.
- If every server uses its own ticket numbers, the newest ticket does not necessarily have the largest number. This problem can be solved if clients suggest the ticket numbers themselves!

Algorithm 15.13 Paxos

	Client (Proposer)	**Server (Acceptor)**
	Initialization	
	c ◁ *command to execute* $t = 0$ ◁ *ticket number to try*	$T_{\max} = 0$ ◁ *largest issued ticket* $C = \perp$ ◁ *stored command* $T_{\text{store}} = 0$ ◁ *ticket used to store C*
	Phase 1	
1:	$t = t + 1$	
2:	Ask all servers for ticket t	
3:		**if** $t > T_{\max}$ **then**
4:		$T_{\max} = t$
5:		Answer with $\texttt{ok}(T_{\text{store}}, C)$
6:		**end if**
	Phase 2	
7:	**if** a majority answers `ok` **then**	
8:	Pick (T_{store}, C) with largest T_{store}	
9:	**if** $T_{\text{store}} > 0$ **then**	
10:	$c = C$	
11:	**end if**	
12:	Send $\texttt{propose}(t, c)$ to same majority	
13:	**end if**	
14:		**if** $t = T_{\max}$ **then**
15:		$C = c$
16:		$T_{\text{store}} = t$
17:		Answer `success`
18:		**end if**
	Phase 3	
19:	**if** a majority answers `success` **then**	
20:	Send $\texttt{execute}(c)$ to every server	
21:	**end if**	

Remarks:

- Unlike previously mentioned algorithms, there is no step where a client explicitly decides to start a new attempt and jumps back to Phase 1. Note that this is not necessary, as a client can decide to abort the current attempt and start a new one *at any point* in the algorithm. This has the advantage that we do not need to be careful about selecting "good" values for timeouts, as correctness is independent of the decisions when to start new attempts.
- The performance can be improved by letting the servers send negative

replies in phases 1 and 2 if the ticket expired.

- The contention between different clients can be alleviated by randomizing the waiting times between consecutive attempts.

Lemma 15.14. *We call a message* propose(t,c) *sent by clients on Line 12 a* proposal for (t,c). *A proposal for (t,c) is* chosen, *if it is stored by a majority of servers (Line 15). For every issued* propose(t',c') *with $t' > t$ holds that $c' = c$, if there was a chosen* propose(t,c).

Proof. Observe that there can be at most one proposal for every ticket number τ since clients only send a proposal if they received a majority of the tickets for τ (Line 7). Hence, every proposal is uniquely identified by its ticket number τ.

Assume that there is at least one propose(t',c') with $t' > t$ and $c' \neq c$; of such proposals, consider the proposal with the smallest ticket number t'. Since both this proposal and also the propose(t,c) have been sent to a majority of the servers, we can denote by S the non-empty intersection of servers that have been involved in both proposals. Recall that since propose(t,c) has been chosen, this means that that at least one server $s \in S$ must have stored command c; thus, when the command was stored, the ticket number t was still valid. Hence, s must have received the request for ticket t' *after* it already stored propose(t,c), as the request for ticket t' invalidates ticket t.

Therefore, the client that sent propose(t',c') must have learned from s that a client already stored propose(t,c). Since a client adapts its proposal to the command that is stored with the highest ticket number so far (Line 8), the client must have proposed c as well. There is only one possibility that would lead to the client not adapting c: If the client received the information from a server that some client stored propose(t^*,c^*), with $c^* \neq c$ and $t^* > t$. But in that case, a client must have sent propose(t^*,c^*) with $t < t^* < t'$, but this contradicts the assumption that t' is the smallest ticket number of a proposal issued after t. □

Theorem 15.15. *If a command c is executed by some servers, all servers (eventually) execute c.*

Proof. From Lemma 15.14 we know that once a proposal for c is chosen, every subsequent proposal is for c. As there is exactly one first propose(t,c) that is chosen, it follows that all successful proposals will be for the command c. Thus, only proposals for a single command c can be chosen, and since clients only tell servers to execute a command, when it is chosen (Line 20), each client will eventually tell every server to execute c. □

Remarks:

- If the client with the first successful proposal does not crash, it will directly tell every server to execute c.
- However, if the client crashes before notifying any of the servers, the servers will execute the command only once the next client is successful. Once a server received a request to execute c, it can inform every client that arrives later that there is already a chosen command, so that the client does not waste time with the proposal process.

- Note that Paxos cannot make progress if half (or more) of the servers crash, as clients cannot achieve a majority anymore.
- The original description of Paxos uses three roles: Proposers, acceptors and learners. Learners have a trivial role: They do nothing, they just learn from other nodes which command was chosen.
- We assigned every node only one role. In some scenarios, it might be useful to allow a node to have multiple roles. For example in a peer-to-peer scenario nodes need to act as both client and server.
- Clients (Proposers) must be trusted to follow the protocol strictly. However, this is in many scenarios not a reasonable assumption. In such scenarios, the role of the proposer can be executed by a set of servers, and clients need to contact proposers, to propose values in their name.
- So far, we only discussed how a set of nodes can reach decision for a single command with the help of Paxos. We call such a single decision an *instance* of Paxos.
- If we want to execute multiple commands, we can extend each instance with an instance number, that is sent around with every message. Once a command is chosen, any client can decide to start a new instance with the next number. If a server did not realize that the previous instance came to a decision, the server can ask other servers about the decisions to catch up.

Chapter Notes

Two-phase protocols have been around for a long time, and it is unclear if there is a single source of this idea. One of the earlier descriptions of this concept can found in the book of Gray [Gra78].

Leslie Lamport introduced Paxos in 1989. But why is it called Paxos? Lamport described the algorithm as the solution to a problem of the parliament of a fictitious Greek society on the island Paxos. He even liked this idea so much, that he gave some lectures in the persona of an Indiana-Jones-style archaeologist! When the paper was submitted, many readers were so distracted by the descriptions of the activities of the legislators, they did not understand the meaning and purpose of the algorithm. The paper was rejected. But Lamport refused to rewrite the paper, and he later wrote that he *"was quite annoyed at how humorless everyone working in the field seemed to be"*. A few years later, when the need for a protocol like Paxos arose again, Lamport simply took the paper out of the drawer and gave it to his colleagues. They liked it. So Lamport decided to submit the paper (in basically unaltered form!) again, 8 years after he wrote it – and it got accepted! But as this paper [Lam98] is admittedly hard to read, he had mercy, and later wrote a simpler description of Paxos [Lam01].

This chapter was written in collaboration with David Stolz.

Bibliography

[Gra78] James N Gray. *Notes on data base operating systems*. Springer, 1978.

[Lam98] Leslie Lamport. The part-time parliament. *ACM Transactions on Computer Systems (TOCS)*, 16(2):133–169, 1998.

[Lam01] Leslie Lamport. Paxos made simple. *ACM Sigact News*, 32(4):18–25, 2001.

Chapter 16

Consensus

16.1 Two Friends

Alice wants to arrange dinner with Bob, and since both of them are very reluctant to use the "call" functionality of their phones, she sends a text message suggesting to meet for dinner at 6pm. However, texting is unreliable, and Alice cannot be sure that the message arrives at Bob's phone, hence she will only go to the meeting point if she receives a confirmation message from Bob. But Bob cannot be sure that his confirmation message is received; if the confirmation is lost, Alice cannot determine if Bob did not even receive her suggestion, or if Bob's confirmation was lost. Therefore, Bob demands a confirmation message from Alice, to be sure that she will be there. But as this message can also be lost...

You can see that such a message exchange continues forever, if both Alice and Bob want to be sure that the other person will come to the meeting point!

Remarks:

- Such a protocol cannot terminate: Assume that there are protocols which lead to agreement, and P is one of the protocols which require the least number of messages. As the last confirmation might be lost and the protocol still needs to guarantee agreement, we can simply decide to always omit the last message. This gives us a new protocol P' which requires less messages than P, contradicting the assumption that P required the minimal amount of messages.
- Can Alice and Bob use Paxos?

16.2 Consensus

In Chapter 15 we studied a problem that we vaguely called agreement. We will now introduce a formally specified variant of this problem, called *consensus*.

Definition 16.1 (consensus)**.** *There are n nodes, of which at most f might crash, i.e., at least $n - f$ nodes are* correct*. Node i starts with an input value v_i. The nodes must decide for one of those values, satisfying the following properties:*

- **Agreement** *All correct nodes decide for the same value.*
- **Termination** *All correct nodes terminate in finite time.*
- **Validity** *The decision value must be the input value of a* node.

Remarks:

- We assume that every node can send messages to every other node, and that we have reliable links, i.e., a message that is sent will be received.
- There is no broadcast medium. If a node wants to send a message to multiple nodes, it needs to send multiple individual messages.
- Does Paxos satisfy all three criteria? If you study Paxos carefully, you will notice that Paxos does not guarantee termination. For example, the system can be stuck forever if two clients continuously request tickets, and neither of them ever manages to acquire a majority.

16.3 Impossibility of Consensus

Model 16.2 (asynchronous)**.** *In the* asynchronous model*, algorithms are event based ("upon receiving message ..., do ..."). Nodes do not have access to a synchronized wall-clock. A message sent from one node to another will arrive in a finite but unbounded time.*

Remarks:

- The asynchronous time model is a widely used formalization of the variable message delay model (Model 15.6).

Definition 16.3 (asynchronous runtime)**.** *For algorithms in the asynchronous model, the* runtime *is the number of time units from the start of the execution to its completion in the worst case (every legal input, every execution scenario), assuming that each message has a delay of* at most *one time unit.*

Remarks:

- The maximum delay cannot be used in the algorithm design, i.e., the algorithm must work independent of the actual delay.
- Asynchronous algorithms can be thought of as systems, where local computation is significantly faster than message delays, and thus can be done in no time. Nodes are only active once an event occurs (a message arrives), and then they perform their actions "immediately".
- We will show now that crash failures in the asynchronous model can be quite harsh. In particular there is no deterministic fault-tolerant consensus algorithm in the asynchronous model, not even for binary input.

Definition 16.4 (configuration)**.** *We say that a system is fully defined (at any point during the execution) by its* configuration C*. The configuration includes the state of every node, and all messages that are in transit (sent but not yet received).*

Definition 16.5 (univalent)**.** *We call a configuration C* univalent*, if the decision value is determined independently of what happens afterwards.*

Remarks:

- We call a configuration that is univalent for value v *v-valent*.
- Note that a configuration can be univalent, even though no single node is aware of this. For example, the configuration in which all nodes start with value 0 is 0-valent (due to the validity requirement).
- As we restricted the input values to be binary, the decision value of any consensus algorithm will also be binary (due to the validity requirement).

Definition 16.6 (bivalent)**.** *A configuration C is called* bivalent *if the nodes might decide for* 0 *or* 1*.*

Remarks:

- The decision value depends on the order in which messages are received or on crash events. I.e., the decision is not yet made.
- We call the initial configuration of an algorithm C_0. When nodes are in C_0, all of them executed their initialization code and possibly sent some messages, and are now waiting for the first message to arrive.

Lemma 16.7. *There is at least one selection of input values V such that the according initial configuration C_0 is bivalent, if $f \geq 1$.*

Proof. Note that C_0 only depends on the input values of the nodes, as no event occurred yet. Let $V = [v_0, v_1, \ldots, v_{n-1}]$ denote the array of input values, where v_i is the input value of node i.

We construct $n+1$ arrays $V_0, V_1, \ldots, V_n$, where the index i in V_i denotes the position in the array up to which all input values are 1. So, $V_0 = [0, 0, 0, \ldots, 0]$, $V_1 = [1, 0, 0, \ldots, 0]$, and so on, up to $V_n = [1, 1, 1, \ldots, 1]$.

Note that the configuration corresponding to V_0 must be 0-valent so that the validity requirement is satisfied. Analogously, the configuration corresponding to V_n must be 1-valent. Assume that all initial configurations with starting values V_i are univalent. Therefore, there must be at least one index b, such that the configuration corresponding to V_b is 0-valent, and configuration corresponding to V_{b+1} is 1-valent. Observe that only the input value of the b^{th} node differs from V_b to V_{b+1}.

Since we assumed that the algorithm can tolerate at least one failure, i.e., $f \geq 1$, we look at the following execution: All nodes except b start with their initial value according to V_b respectively V_{b+1}. Node b is "extremely slow"; i.e., all messages sent by b are scheduled in such a way, that all other nodes must assume that b crashed, in order to satisfy the termination requirement.

Since the nodes cannot determine the value of b, and we assumed that all initial configurations are univalent, they will decide for a value v independent of the initial value of b. Since V_b is 0-valent, v must be 0. However we know that V_{b+1} is 1-valent, thus v must be 1. Since v cannot be both 0 and 1, we have a contradiction.

□

Definition 16.8 (transition)**.** *A* transition *from configuration C to a following configuration C_τ is characterized by an event $\tau = (u, m)$, i.e., node u receiving message m.*

Remarks:

- Transitions are the formally defined version of the "events" in the asynchronous model we described before.
- A transition $\tau = (u, m)$ is only applicable to C, if m was still in transit in C.
- C_τ differs from C as follows: m is no longer in transit, u has possibly a different state (as u can update its state based on m), and there are (potentially) new messages in transit, sent by u.

Definition 16.9 (configuration tree)**.** *The* configuration tree *is a directed tree of configurations. Its root is the configuration C_0 which is fully characterized by the input values V. The edges of the tree are the transitions; every configuration has all applicable transitions as outgoing edges.*

Remarks:

- For any algorithm, there is exactly *one* configuration tree for every selection of input values.
- Leaves are configurations where the execution of the algorithm terminated. Note that we use termination in the sense that the system as a whole terminated, i.e., there will not be any transition anymore.
- Every path from the root to a leaf is one possible asynchronous execution of the algorithm.
- Leaves must be univalent, or the algorithm terminates without agreement.
- If a node u crashes when the system is in C, all transitions $(u, *)$ are removed from C in the configuration tree.

Lemma 16.10. *Assume two transitions $\tau_1 = (u_1, m_1)$ and $\tau_2 = (u_2, m_2)$ for $u_1 \neq u_2$ are both applicable to C. Let $C_{\tau_1\tau_2}$ be the configuration that follows C by first applying transition τ_1 and then τ_2, and let $C_{\tau_2\tau_1}$ be defined analogously. It holds that $C_{\tau_1\tau_2} = C_{\tau_2\tau_1}$.*

Proof. Observe that τ_2 is applicable to C_{τ_1}, since m_2 is still in transit and τ_1 cannot change the state of u_2. With the same argument τ_1 is applicable to C_{τ_2}, and therefore both $C_{\tau_1\tau_2}$ and $C_{\tau_2\tau_1}$ are well-defined. Since the two transitions are completely independent of each other, meaning that they consume the same messages, lead to the same state transitions and to the same messages being sent, it follows that $C_{\tau_1\tau_2} = C_{\tau_2\tau_1}$. □

Definition 16.11 (critical configuration)**.** *We say that a configuration C is* critical*, if C is bivalent, but all configurations that are direct children of C in the configuration tree are univalent.*

Remarks:

- Informally, C is critical, if it is the last moment in the execution where the decision is not yet clear. As soon as the next message is processed by any node, the decision will be determined.

Lemma 16.12. *If a system is in a bivalent configuration, it must reach a critical configuration within finite time, or it does not always solve consensus.*

Proof. Recall that there is at least one bivalent initial configuration (Lemma 16.7). Assuming that this configuration is not critical, there must be at least one bivalent following configuration; hence, the system may enter this configuration. But if this configuration is not critical as well, the system may afterwards progress into another bivalent configuration. As long as there is no critical configuration, an unfortunate scheduling (selection of transitions) can always lead the system into another bivalent configuration. The only way how an algorithm can *enforce* to arrive in a univalent configuration is by reaching a critical configuration.

Therefore we can conclude that a system which does not reach a critical configuration has at least one possible execution where it will terminate in a bivalent configuration (hence it terminates without agreement), or it will not terminate at all.

□

Lemma 16.13. *If a configuration tree contains a critical configuration, crashing a single node can create a bivalent leaf; i.e., a crash prevents the algorithm from reaching agreement.*

Proof. Let C denote critical configuration in a configuration tree, and let T be the set of transitions applicable to C. Let $\tau_0 = (u_0, m_0) \in T$ and $\tau_1 = (u_1, m_1) \in T$ be two transitions, and let C_{τ_0} be 0-valent and C_{τ_1} be 1-valent. Note that T must contain these transitions, as C is a critical configuration.

Assume that $u_0 \neq u_1$. Using Lemma 16.10 we know that C has a following configuration $C_{\tau_0\tau_1} = C_{\tau_1\tau_0}$. Since this configuration follows C_{τ_0} it must be 0-valent. However, this configuration also follows C_{τ_1} and must hence be 1-valent. This is a contradiction and therefore $u_0 = u_1$ must hold.

Therefore we can pick one particular node u for which there is a transition $\tau = (u, m) \in T$ which leads to a 0-valent configuration. As shown before, all transitions in T which lead to a 1-valent configuration must also take place on u. Since C is critical, there must be at least one such transition. Applying the same argument again, it follows that all transitions in T that lead to a 0-valent

configuration must take place on u as well, and since C is critical, there is no transition in T that leads to a bivalent configuration. Therefore *all* transitions applicable to C take place on the *same* node u!

If this node u crashes while the system is in C, *all transitions are removed*, and therefore the system is stuck in C, i.e., it terminates in C. But as C is critical, and therefore bivalent, the algorithm fails to reach an agreement.

□

Theorem 16.14. *There is no deterministic algorithm which always achieves consensus in the asynchronous model, with $f > 0$.*

Proof. We assume that the input values are binary, as this is the easiest non-trivial possibility. From Lemma 16.7 we know that there must be at least one bivalent initial configuration C. Using Lemma 16.12 we know that if an algorithm solves consensus, all executions starting from the bivalent configuration C must reach a critical configuration. But if the algorithm reaches a critical configuration, a single crash can prevent agreement (Lemma 16.13). □

Remarks:

- If $f = 0$, then each node can simply send its value to all others, wait for all values, and choose the minimum.
- But if a single node may crash, there is no deterministic solution to consensus in the asynchronous model.
- How can the situation be improved? For example by giving each node access to randomness, i.e., we allow each node to toss a coin.

16.4 Randomized Consensus

Algorithm 16.15 Randomized Consensus (Ben-Or)

```
1:  v_i ∈ {0,1}                    ◁ input bit
2:  round = 1
3:  decided = false
4:  Broadcast myValue(v_i, round)
5:  while true do
        Propose
6:      Wait until a majority of myValue messages of current round arrived
7:      if all messages contain the same value v then
8:          Broadcast propose(v, round)
9:      else
10:         Broadcast propose(⊥, round)
11:     end if
12:     if decided then
13:         Broadcast myValue(v_i, round+1)
14:         Decide for v_i and terminate
15:     end if
        Adapt
16:     Wait until a majority of propose messages of current round arrived
17:     if all messages propose the same value v then
18:         v_i = v
19:         decide = true
20:     else if there is at least one proposal for v then
21:         v_i = v
22:     else
23:         Choose v_i randomly, with Pr[v_i = 0] = Pr[v_i = 1] = 1/2
24:     end if
25:     round = round + 1
26:     Broadcast myValue(v_i, round)
27: end while
```

Remarks:

- The idea of Algorithm 16.15 is very simple: Either all nodes start with the same input bit, which makes consensus easy. Otherwise, nodes toss a coin until a large number of nodes get – by chance – the same outcome.

Lemma 16.16. *As long as no node sets* decided *to true, Algorithm 16.15 always makes progress, independent of which nodes crash.*

Proof. The only two steps in the algorithm when a node waits are in Lines 6 and 15. Since a node only waits for a majority of the nodes to send a message,

and since $f < n/2$, the node will always receive enough messages to continue, as long as no correct node set its value decided to true and terminates. □

Lemma 16.17. *Algorithm 16.15 satisfies the validity requirement.*

Proof. Observe that the validity requirement of consensus, when restricted to binary input values, corresponds to: If all nodes start with v, then v must be chosen; otherwise, either 0 or 1 is acceptable, and the validity requirement is automatically satisfied.

Assume that all nodes start with v. In this case, all nodes propose v in the first round. As all nodes only hear proposals for v, all nodes decide for v (Line 17) and exit the loop in the following round. □

Lemma 16.18. *Algorithm 16.15 satisfies the agreement requirement.*

Proof. Observe that proposals for both 0 and 1 cannot occur in the same round, as nodes only send a proposal for v, if they hear a *majority* for v in Line 8.

Let u be the first node that decides for a value v in round r. Hence, it received a majority of proposals for v in r (Line 17). Note that once a node receives a majority of proposals for a value, it will adapt this value and terminate in the next round. Since there cannot be a proposal for any other value in r, it follows that no node decides for a different value in r.

In Lemma 16.16 we only showed that nodes make progress as long as no node decides, thus we need to be careful that no node gets stuck if u terminates.

Any node $u' \neq u$ can experience one of two scenarios: Either it also receives a majority for v in round r and decides, or it does not receive a majority. In the first case, the agreement requirement is directly satisfied, and also the node cannot get stuck. Let us study the latter case. Since u heard a majority of proposals for v, it follows that every node hears *at least one* proposal for v. Hence, all nodes set their value v_i to v in round r. Therefore, all nodes will broadcast v at the end of round r, and thus all nodes will propose v in round $r+1$. The nodes that already decided in round r will terminate in $r+1$ and send one additional `myValue` message (Line 13). All other nodes will receive a majority of proposals for v in $r+1$, and will set decided to true in round $r+1$, and also send a `myValue` message in round $r+1$. Thus, in round $r+2$ some nodes have already terminated, and others hear enough `myValue` messages to make progress in Line 6. They send another `propose` and a `myValue` message and terminate in $r+2$, deciding for the same value v. □

Lemma 16.19. *Algorithm 16.15 satisfies the termination requirement, i.e., all nodes terminate in expected time $O(2^n)$.*

Proof. We know from the proof of Lemma 16.18 that once a node hears a majority of proposals for a value, all nodes will terminate at most two rounds later. Hence, we only need to show that a node receives a majority of proposals for the same value within expected time $O(2^n)$.

Assume that no node receives a majority of proposals for the same value. In such a round, some nodes may update their value to v based on a proposal (Line 20). As shown before, all nodes that update the value based on a proposal, adapt the same value v. The rest of the nodes choses 0 or 1 randomly. The probability that all nodes choose the same value v in one round is hence at least $1/2^n$. Therefore, the expected number of rounds is bounded by $O(2^n)$. As

every round consists of two message exchanges, the asymptotic runtime of the algorithm is equal to the number of rounds. □

Theorem 16.20. *Algorithm 16.15 achieves binary consensus with expected runtime $O(2^n)$ if up to $f < n/2$ nodes crash.*

Remarks:

- How good is a fault tolerance of $f < n/2$?

Theorem 16.21. *There is no consensus algorithm for the asynchronous model that tolerates $f \geq n/2$ many failures.*

Proof. Assume that there is an algorithm that can handle $f = n/2$ many failures. We partition the set of all nodes into two sets N, N' both containing $n/2$ many nodes. Let us look at three different selection of input values: In V_0 all nodes start with 0. In V_1 all nodes start with 1. In V_{half} all nodes in N start with 0, and all nodes in N' start with 1.

Assume that nodes start with V_{half}. Since the algorithm must solve consensus independent of the scheduling of the messages, we study the scenario where all messages sent from nodes in N to nodes in N' (or vice versa) are heavily delayed. Note that the nodes in N cannot determine if they started with V_0 or V_{half}. Analogously, the nodes in N' cannot determine if they started in V_1 or V_{half}. Hence, if the algorithm terminates before any message from the other set is received, N must decide for 0 and N' must decide for 1 (to satisfy the validity requirement, as they could have started with V_0 respectively V_1). Therefore, the algorithm would fail to reach agreement.

The only possibility to overcome this problem is to wait for at least one message sent from a node of the other set. However, as $f = n/2$ many nodes can crash, the entire other set could have crashed before they sent any message. In that case, the algorithm would wait forever and therefore not satisfy the termination requirement.

□

Remarks:

- Algorithm 16.15 solves consensus with optimal fault-tolerance – but it is awfully slow. The problem is rooted in the individual coin tossing: If all nodes toss the same coin, they could terminate in a constant number of rounds.
- Can this problem be fixed by simply always choosing 1 at Line 22?!
- This cannot work: Such a change makes the algorithm deterministic, and therefore it cannot achieve consensus (Theorem 16.14). Simulating what happens by always choosing 1, one can see that it might happen that there is a majority for 0, but a minority with value 1 prevents the nodes from reaching agreement.
- Nevertheless, the algorithm can be improved by tossing a so-called *shared coin*. A shared coin is a random variable that is 0 for all nodes with constant probability, and 1 with constant probability. Of course, such a coin is not a magic device, but it is simply an algorithm. To improve the expected runtime of Algorithm 16.15, we replace Line 22 with a function call to the shared coin algorithm.

16.5 Shared Coin

Algorithm 16.22 Shared Coin (code for node u)

```
1:  Choose local coin c_u = 0 with probability 1/n, else c_u = 1
2:  Broadcast myCoin(c_u)
3:  Wait for n − f coins and store them in the local coin set C_u
4:  Broadcast mySet(C_u)
5:  Wait for n − f coin sets
6:  if at least one coin is 0 among all coins in the coin sets then
7:      return 0
8:  else
9:      return 1
10: end if
```

Remarks:

- Since at most f nodes crash, all nodes will always receive $n - f$ coins respectively coin sets in Lines 3 and 5. Therefore, all nodes make progress and termination is guaranteed.
- We show the correctness of the algorithm for $f < n/3$. To simplify the proof we assume that $n = 3f + 1$, i.e., we assume the worst case.

Lemma 16.23. *Let u be a node, and let W be the set of coins that u received in at least $f + 1$ different coin sets. It holds that $|W| \geq f + 1$.*

Proof. Let C be the multiset of coins received by u. Observe that u receives exactly $|C| = (n-f)^2$ many coins, as u waits for $n-f$ coin sets each containing $n - f$ coins.

Assume that the lemma does not hold. Then, at most f coins are in all $n-f$ coin sets, and all other coins $(n - f)$ are in at most f coin sets. In other words, the number of total of coins that u received is bounded by

$$|C| \leq f \cdot (n - f) + (n - f) \cdot f = 2f(n - f).$$

Our assumption was that $n > 3f$, i.e., $n - f > 2f$. Therefore $|C| \leq 2f(n-f) < (n - f)^2 = |C|$, which is a contradiction. □

Lemma 16.24. *All coins in W are seen by all correct nodes.*

Proof. Let $w \in W$ be such a coin. By definition of W we know that w is in at least $f + 1$ sets received by u. Since every other node also waits for $n - f$ sets before terminating, each node will receive at least one of these sets, and hence w must be seen by every node that terminates. □

Theorem 16.25. *If $f < n/3$ nodes crash, Algorithm 16.22 implements a shared coin.*

Proof. Let us first bound the probability that the algorithm returns 1 for all nodes. With probability $(1-1/n)^n \approx 1/e \approx 0.37$ all nodes chose their local coin equal to 1 (Line 1), and in that case 1 will be decided. This is only a lower bound on the probability that all nodes return 1, as there are also other scenarios based on message scheduling and crashes which lead to a global decision for 1. But a probability of 0.37 is good enough, so we do not need to consider these scenarios.

With probability $1-(1-1/n)^{|W|}$ there is at least one 0 in W. Using Lemma 16.23 we know that $|W| \geq f+1 \approx n/3$, hence the probability is about $1-(1-1/n)^{n/3} \approx 1-(1/e)^{1/3} \approx 0.28$. We know that this 0 is seen by all nodes (Lemma 16.24), and hence everybody will decide 0. Thus Algorithm 16.22 implements a shared coin. □

Remarks:

- We only proved the worst case. By choosing f fairly small, it is clear that $f+1 \not\approx n/3$. However, Lemma 16.23 can be proved for $|W| \geq n-2f$. To prove this claim you need to substitute the expressions in the contradictory statement: At most $n-2f-1$ coins can be in all $n-f$ coin sets, and $n-(n-2f-1) = 2f+1$ coins can be in at most f coin sets. The remainder of the proof is analogous, the only difference is that the math is not as neat. Using the modified Lemma we know that $|W| \geq n/3$, and therefore Theorem 16.25 also holds for *any* $f < n/3$.

- We implicitly assumed that message scheduling was random; if we need a 0 but the nodes that want to propose 0 are "slow", nobody is going to see these 0's, and we do not have progress.

Theorem 16.26. *Plugging Algorithm 16.22 into Algorithm 16.15 we get a randomized consensus algorithm which terminates in a constant expected number of rounds tolerating up to $f < n/3$ crash failures.*

Chapter Notes

The problem of two friends arranging a meeting was presented and studied under many different names; nowadays, it is usually referred to as the *Two Generals Problem*. The impossibility proof was established in 1975 by Akkoyunlu et al. [AEH75].

The proof that there is no deterministic algorithm that always solves consensus is based on the proof of Fischer, Lynch and Paterson [FLP85], known as FLP, which they established in 1985. This result was awarded the 2001 PODC Influential Paper Award (now called Dijkstra Prize). The idea for the randomized consensus algorithm was originally presented by Ben-Or [Ben83]. The concept of a shared coin was introduced by Bracha [Bra87].

This chapter was written in collaboration with David Stolz.

Bibliography

[AEH75] EA Akkoyunlu, K Ekanadham, and RV Huber. Some constraints and tradeoffs in the design of network communications. In *ACM SIGOPS Operating Systems Review*, volume 9, pages 67–74. ACM, 1975.

[Ben83] Michael Ben-Or. Another advantage of free choice (extended abstract): Completely asynchronous agreement protocols. In *Proceedings of the second annual ACM symposium on Principles of distributed computing*, pages 27–30. ACM, 1983.

[Bra87] Gabriel Bracha. Asynchronous byzantine agreement protocols. *Information and Computation*, 75(2):130–143, 1987.

[FLP85] Michael J. Fischer, Nancy A. Lynch, and Mike Paterson. Impossibility of Distributed Consensus with One Faulty Process. *J. ACM*, 32(2):374–382, 1985.

Chapter 17

Byzantine Agreement

In order to make flying safer, researchers studied possible failures of various sensors and machines used in airplanes. While trying to model the failures, they were confronted with the following problem: Failing machines did not just crash, instead they sometimes showed arbitrary behavior before stopping completely. With these insights researchers modeled failures as arbitrary failures, not restricted to any patterns.

Definition 17.1 (Byzantine)**.** *A node which can have arbitrary behavior is called* byzantine*. This includes "anything imaginable", e.g., not sending any messages at all, or sending different and wrong messages to different neighbors, or lying about the input value.*

Remarks:

- Byzantine behavior also includes collusion, i.e., all byzantine nodes are being controlled by the same adversary.
- We assume that any two nodes communicate directly, and that no node can forge an incorrect sender address. This is a requirement, such that a single byzantine node cannot simply impersonate all nodes!
- We call non-byzantine nodes *correct* nodes.

Definition 17.2 (Byzantine Agreement)**.** *Finding consensus as in Definition 16.1 in a system with byzantine nodes is called* byzantine agreement*. An algorithm is f-resilient if it still works correctly with f byzantine nodes.*

Remarks:

- As for consensus (Definition 16.1) we also need agreement, termination and validity. Agreement and termination are straight-forward, but what about validity?

17.1 Validity

Definition 17.3 (Any-Input Validity)**.** *The decision value must be the input value of any node.*

Remarks:

- This is the validity definition we implicitly used for consensus, in Definition 16.1.
- Does this definition still make sense in the presence of byzantine nodes? What if byzantine nodes lie about their inputs?
- We would wish for a validity definition which differentiates between byzantine and correct inputs.

Definition 17.4 (Correct-Input Validity)**.** *The decision value must be the input value of a* correct *node.*

Remarks:

- Unfortunately, implementing correct-input validity does not seem to be easy, as a byzantine node following the protocol but lying about its input value is indistinguishable from a correct node. Here is an alternative.

Definition 17.5 (All-Same Validity)**.** *If all correct nodes start with the same input v, the decision value must be v.*

Remarks:

- If the decision values are binary, then correct-input validity is induced by all-same validity.
- If the input values are not binary, but for example from sensors that deliever values in $\mathbb{R}$, all-same validity is in most scenarios not really useful.

Definition 17.6 (Median Validity)**.** *If the input values are orderable, e.g. $v \in \mathbb{R}$, byzantine outliers can be prevented by agreeing on a value close to the median of the correct input values, where close is a function of the number of byzantine nodes f.*

Remarks:

- Is byzantine agreement possible? If yes, with what validity condition?
- Let us try to find an algorithm which tolerates 1 single byzantine node, first restricting to the so-called synchronous model.

Model 17.7 (synchronous)**.** *In the* synchronous model*, nodes operate in synchronous rounds. In each round, each node may send a message to the other nodes, receive the messages sent by the other nodes, and do some local computation.*

Definition 17.8 (synchronous runtime)**.** *For algorithms in the synchronous model, the* runtime *is simply the number of rounds from the start of the execution to its completion in the worst case (every legal input, every execution scenario).*

17.2 How Many Byzantine Nodes?

Algorithm 17.9 Byzantine Agreement with $f = 1$.

1: Code for node u, with input value x:

Round 1

2: Send $\texttt{tuple}(u, x)$ to all other nodes
3: Receive $\texttt{tuple}(v, y)$ from all other nodes v
4: Store all received $\texttt{tuple}(v, y)$ in a set S_u

Round 2

5: Send set S_u to all other nodes
6: Receive sets S_v from all nodes v
7: T = set of $\texttt{tuple}(v, y)$ seen in at least two sets S_v, including own S_u
8: Let $\texttt{tuple}(v, y) \in T$ be the tuple with the smallest value y
9: Decide on value y

Remarks:

- Byzantine nodes may not follow the protocol and send syntactically incorrect messages. Such messages can easily be deteced and discarded. It is worse if byzantine nodes send syntactically correct messages, but with a bogus content, e.g., they send different messages to different nodes.
- Some of these mistakes cannot easily be detected: For example, if a byzantine node sends different values to different nodes in the first round; such values will be put into S_u. However, some mistakes can and must be detected: Observe that all nodes only relay information in Round 2, and do not say anything about their own value. So, if a byzantine node sends a set S_v which contains a $\texttt{tuple}(v, y)$, this tuple must be removed by u from S_v upon receiving it (Line 6).
- Recall that we assumed that nodes cannot forge their source address; thus, if a node receives $\texttt{tuple}(v, y)$ in Round 1, it is guaranteed that this message was sent by v.

Lemma 17.10. *If $n \geq 4$, all correct nodes have the same set T.*

Proof. With $f = 1$ and $n \geq 4$ we have at least 3 correct nodes. A correct node will see every correct value at least twice, once directly from another correct node, and once through the third correct node. So all correct values are in T. If the byzantine node sends the same value to at least 2 other (correct) nodes, all correct nodes will see the value twice, so all add it to set T. If the byzantine node sends all different values to the correct nodes, none of these values will end up in any set T. □

Theorem 17.11. *Algorithm 17.9 reaches byzantine agreement if $n \geq 4$.*

Proof. We need to show agreement, any-input validity and termination. With Lemma 17.10 we know that all correct nodes have the same set T, and therefore agree on the same minimum value. The nodes agree on a value proposed by any node, so any-input validity holds. Moreover, the algorithm terminates after two rounds. □

Remarks:

- If $n > 4$ the byzantine node can put multiple values into T.
- The idea of this algorithm can be generalized for any f and $n > 3f$. In the generalization, every node sends in every of $f + 1$ rounds all information it learned so far to all other nodes. In other words, message size increases exponentially with f.
- Does Algorithm 17.9 also work with $n = 3$?

Theorem 17.12. *Three nodes cannot reach byzantine agreement with all-same validity if one node among them is byzantine.*

Proof. We have three nodes u, v, w. In order to achieve all-same validity, a correct node must decide on its own value if another node supports that value. The third node might disagree, but that node could be byzantine. If correct node u has input 0 and correct node v has input 1, the byzantine node w can fool them by telling u that its value is 0 and simultaneously telling v that its value is 1. This leads to u and v deciding on their own values, which results in violating the agreement condition. Even if u talks to v, and they figure out that they have different assumptions about w's value, u cannot distinguish whether w or v is byzantine. □

Theorem 17.13. *A network with n nodes cannot reach byzantine agreement with $f \geq n/3$ byzantine nodes.*

Proof. Let us assume (for the sake of contradiction) that there exists an algorithm A that reaches byzantine agreement for n nodes with $f \geq n/3$ byzantine nodes. With A, we can solve byzantine agreement with 3 nodes. For simplicity, we call the 3 nodes u, v, w supernodes.

Each supernode simulates $n/3$ nodes, either $\lfloor n/3 \rfloor$ or $\lceil n/3 \rceil$, if n is not divisible by 3. Each simulated node starts with the input of its supernode. Now the three supernodes simulate algorithm A. The single byzantine supernode simulates $\lceil n/3 \rceil$ byzantine nodes. As algorithm A promises to solve byzantine agreement for $f \geq n/3$, A has to be able to handle $\lceil n/3 \rceil$ byzantine nodes. Algorithm A guarantees that the correct nodes simulated by the correct two supernodes will achieve byzantine agreement. So the two correct supernodes can just take the value of their simulated nodes (these values have to be the same by the agreement property), and we have achieved agreement for three supernodes, one of them byzantine. This contradicts Lemma 17.12, hence algorithm A cannot exist. □

17.3 The King Algorithm

Algorithm 17.14 King Algorithm (for $f < n/3$)

```
1:  x = my input value
2:  for phase = 1 to f + 1 do
        Round 1
3:      Broadcast value(x)
        Round 2
4:      if some value(y) at least n − f times then
5:          Broadcast propose(y)
6:      end if
7:      if some propose(z) received more than f times then
8:          x = z
9:      end if
        Round 3
10:     Let node v_i be the predefined king of this phase i
11:     The king v_i broadcasts its current value w
12:     if received strictly less than n − f propose(x) then
13:         x = w
14:     end if
15: end for
```

Lemma 17.15. *Algorithm 17.14 fulfills the all-same validity.*

Proof. If all correct nodes start with the same value, all correct nodes propose it in Round 2. All correct nodes will receive at least $n - f$ proposals, i.e., all correct nodes will stick with this value, and never change it to the king's value. This holds for all phases. □

Lemma 17.16. *If a correct node proposes x, no other correct node proposes y, with $y \neq x$, if $n > 3f$.*

Proof. Assume (for the sake of contradiction) that a correct node proposes value x and another correct node proposes value y. Since a good node only proposes a value if it heard at least $n-f$ `value` messages, we know that both nodes must have received their value from at least $n-2f$ distinct correct nodes (as at most f nodes can behave byzantine and send x to one node and y to the other one). Hence, there must be a total of at least $2(n-2f)+f = 2n-3f$ nodes in the system. Using $3f < n$, we have $2n - 3f > n$ nodes, a contradiction. □

Lemma 17.17. *There is at least one phase with a correct king.*

Proof. There are $f + 1$ phases, each with a different king. As there are only f byzantine nodes, one king must be correct. □

Lemma 17.18. *After a round with a correct king, the correct nodes will not change their values v anymore, if $n > 3f$.*

Proof. If all correct nodes change their values to the king's value, all correct nodes have the same value. If some correct node does not change its value to the king's value, it received a proposal at least $n - f$ times, therefore at least $n-2f$ correct nodes broadcasted this proposal. Thus, all correct nodes received it at least $n - 2f > f$ times (using $n > 3f$), therefore all correct nodes set their value to the proposed value, including the correct king. Note that only one value can be proposed more than f times, which follows from Lemma 17.16. With Lemma 17.15, no node will change its value after this round. □

Theorem 17.19. *Algorithm 17.14 solves byzantine agreement.*

Proof. The king algorithm reaches agreement as either all correct nodes start with the same value, or they agree on the same value latest after the phase where a correct node was king according to Lemmas 17.17 and 17.18. Because of Lemma 17.15 we know that they will stick with this value. Termination is guaranteed after $3(f + 1)$ rounds, and all-same validity is proved in Lemma 17.15. □

Remarks:

- Algorithm 17.14 requires $f + 1$ predefined kings. We assume that the kings (and their order) are given. Finding the kings indeed would be a byzantine agreement task by itself, so this must be done before the execution of the King algorithm.
- Do algorithms exist which do not need predefined kings? Yes, see Section 17.5.
- Can we solve byzantine agreement (or at least consensus) in less than $f + 1$ rounds?

17.4 Lower Bound on Number of Rounds

Theorem 17.20. *A synchronous algorithm solving consensus in the presence of f crashing nodes needs at least $f+1$ rounds, if nodes decide for the minimum seen value.*

Proof. Let us assume (for the sake of contradiction) that some algorithm A solves consensus in f rounds. Some node u_1 has the smallest input value x, but in the first round u_1 can send its information (including information about its value x) to only some other node u_2 before u_1 crashes. Unfortunately, in the second round, the only witness u_2 of x also sends x to exactly one other node u_3 before u_2 crashes. This will be repeated, so in round f only node u_{f+1} knows about the smallest value x. As the algorithm terminates in round f, node u_{f+1} will decide on value x, all other surviving (correct) nodes will decide on values larger than x. □

Remarks:

- A general proof without the restriction to decide for the minimum value exists as well.
- Since byzantine nodes can also just crash, this lower bound also holds for byzantine agreement, so Algorithm 17.14 has an asymptotically optimal runtime.
- So far all our byzantine agreement algorithms assume the synchronous model. Can byzantine agreement be solved in the asynchronous model?

17.5 Asynchronous Byzantine Agreement

Algorithm 17.21 Asynchronous Byzantine Agreement (Ben-Or, for $f < n/9$)

```
1:  x_i ∈ {0,1}                    ◁ input bit
2:  r = 1                          ◁ round
3:  decided = false
4:  Broadcast propose(x_i,r)
5:  repeat
6:     Wait until n − f propose messages of current round r arrived
7:     if at least n − 2f propose messages contain the same value x then
8:        x_i = x, decided = true
9:     else if at least n − 4f propose messages contain the same value x then
10:       x_i = x
11:    else
12:       choose x_i randomly, with Pr[x_i = 0] = Pr[x_i = 1] = 1/2
13:    end if
14:    r = r + 1
15:    Broadcast propose(x_i,r)
16: until decided (see Line 8)
17: decision = x_i
```

Lemma 17.22. *Assume $n > 9f$. If a correct node chooses value x in Line 10, then no other correct node chooses value $y \neq x$ in Line 10.*

Proof. For the sake of contradiction, assume that both 0 and 1 are chosen in Line 10. This means that both 0 and 1 had been proposed by at least $n - 5f$ correct nodes. In other words, we have a total of at least $2(n-5f)+f = n+(n-9f) > n$ nodes. Contradiction! □

Theorem 17.23. *Algorithm 17.21 solves binary byzantine agreement as in Definition 17.2 for up to $f < n/9$ byzantine nodes.*

Proof. First note that it is not a problem to wait for $n - f$ propose messages in Line 6, since at most f nodes are byzantine. If all correct nodes have the same input value x, then all (except the f byzantine nodes) will propose the same value x. Thus, every node receives at least $n-2f$ propose messages containing x, deciding on x in the first round already. We have established all-same validity!

If the correct nodes have different (binary) input values, the validity condition becomes trivial as any result is fine.

What about agreement? Let u be the first node to decide on value x (in Line 8). Due to asynchrony another node v received messages from a different subset of the nodes, however, at most f senders may be different. Taking into account that byzantine nodes may lie (send different propose messages to different nodes), f additional propose messages received by v may differ from those received by u. Since node u had at least $n - 2f$ propose messages with value x, node v has at least $n - 4f$ propose messages with value x. Hence every correct node will propose x in the next round, and then decide on x.

So we only need to worry about termination: We have already seen that as soon as one correct node terminates (Line 8) everybody terminates in the next round. So what are the chances that some node u terminates in Line 8? Well, we can hope that all correct nodes randomly propose the same value (in Line 12). Maybe there are some nodes not choosing at random (entering Line 10 instead of 12), but according to Lemma 17.22 they will all propose the same.

Thus, at worst all $n - f$ correct nodes need to randomly choose the same bit, which happens with probability $2^{-(n-f)+1}$. If so, all correct nodes will send the same propose message, and the algorithm terminates. So the expected running time is exponential in the number of nodes n. □

Remarks:

- This Algorithm is a proof of concept that asynchronous byzantine agreement can be achieved. Unfortunately this algorithm is not useful in practice, because of its runtime.
- For a long time, there was no algorithm with subexponential runtime. The currently fastest algorithm has an expected runtime of $O(n^{2.5})$ but only tolerates $f \leq 1/500n$ many byzantine nodes. This algorithm works along the lines of the shared coin algorithm; additionally nodes try to detect which nodes are byzantine.

Chapter Notes

The project which started the study of byzantine failures was called SIFT and was founded by NASA [WLG+78], and the research regarding byzantine agreement started to get significant attention with the results by Pease, Shostak, and Lamport [PSL80, LSP82]. In [PSL80] they presented the generalized version of Algorithm 17.9 and also showed that byzantine agreement is unsolvable for $n \leq 3f$. The algorithm presented in that paper is nowadays called *Exponential Information Gathering (EIG)*, due to the exponential size of the messages.

There are many algorithms for the byzantine agreement problem. For example the Queen Algorithm [BG89] which has a better runtime than the King algorithm [BGP89], but tolerates less failures. That byzantine agreement requires at least $f + 1$ many rounds was shown by Dolev and Strong [DS83], based on a more complicated proof from Fischer and Lynch [FL82].

While many algorithms for the synchronous model have been around for a long time, the asynchronous model is a lot harder. The only results were by Ben-Or and Bracha. Ben-Or [Ben83] was able to tolerate $f < n/5$. Bracha [BT85]

improved this tolerance to $f < n/3$. The first algorithm with a polynomial expected runtime was found by King and Saia [KS13] just recently.

Nearly all developed algorithms only satisfy all-same validity. There are a few exceptions, e.g., correct-input validity [FG03], available if the initial values are from a finite domain, or median validity [SW15] if the input values are orderable.

Before the term *byzantine* was coined, the terms Albanian Generals or Chinese Generals were used in order to describe malicious behavior. When the involved researchers met people from these countries they moved – for obvious reasons – to the historic term byzantine [LSP82].

This chapter was written in collaboration with Barbara Keller.

Bibliography

[Ben83] Michael Ben-Or. Another advantage of free choice (extended abstract): Completely asynchronous agreement protocols. In *Proceedings of the second annual ACM symposium on Principles of distributed computing*, pages 27–30. ACM, 1983.

[BG89] Piotr Berman and Juan A Garay. *Asymptotically optimal distributed consensus*. Springer, 1989.

[BGP89] Piotr Berman, Juan A. Garay, and Kenneth J. Perry. Towards optimal distributed consensus (extended abstract). In *30th Annual Symposium on Foundations of Computer Science, Research Triangle Park, North Carolina, USA, 30 October - 1 November 1989*, pages 410–415, 1989.

[BT85] Gabriel Bracha and Sam Toueg. Asynchronous consensus and broadcast protocols. *Journal of the ACM (JACM)*, 32(4):824–840, 1985.

[DS83] Danny Dolev and H. Raymond Strong. Authenticated algorithms for byzantine agreement. *SIAM Journal on Computing*, 12(4):656–666, 1983.

[FG03] Matthias Fitzi and Juan A Garay. Efficient player-optimal protocols for strong and differential consensus. In *Proceedings of the twenty-second annual symposium on Principles of distributed computing*, pages 211–220. ACM, 2003.

[FL82] Michael J. Fischer and Nancy A. Lynch. A lower bound for the time to assure interactive consistency. 14(4):183–186, June 1982.

[KS13] Valerie King and Jared Saia. Byzantine agreement in polynomial expected time:[extended abstract]. In *Proceedings of the forty-fifth annual ACM symposium on Theory of computing*, pages 401–410. ACM, 2013.

[LSP82] Leslie Lamport, Robert E. Shostak, and Marshall C. Pease. The byzantine generals problem. *ACM Trans. Program. Lang. Syst.*, 4(3):382–401, 1982.

[PSL80] Marshall C. Pease, Robert E. Shostak, and Leslie Lamport. Reaching agreement in the presence of faults. *J. ACM*, 27(2):228–234, 1980.

[SW15] David Stolz and Roger Wattenhofer. Byzantine Agreement with Median Validity. In *19th International Conference on Priniciples of Distributed Systems (OPODIS), Rennes, France*, 2015.

[WLG+78] John H. Wensley, Leslie Lamport, Jack Goldberg, Milton W. Green, Karl N. Levitt, P. M. Melliar-Smith, Robert E. Shostak, and Charles B. Weinstock. Sift: Design and analysis of a fault-tolerant computer for aircraft control. In *Proceedings of the IEEE*, pages 1240–1255, 1978.

Chapter 18

Authenticated Agreement

Byzantine nodes are able to lie about their inputs as well as received messages. Can we detect certain lies and limit the power of byzantine nodes? Possibly, the authenticity of messages may be validated using signatures?

18.1 Agreement with Authentication

Definition 18.1 (Signature)**.** *If a node never signs a message, then no correct node ever accepts that message. We denote a message* $\texttt{msg}(x)$ *signed by node* u *with* $\texttt{msg}(x)_u$*.*

Remarks:

- Algorithm 18.2 shows an agreement protocol for binary inputs relying on signatures. We assume there is a designated "primary" node p. The goal is to decide on p's value.

Algorithm 18.2 Byzantine Agreement with Authentication

Code for primary p:

```
1: if input is 1 then
2:    broadcast value(1)_p
3:    decide 1 and terminate
4: else
5:    decide 0 and terminate
6: end if
```

Code for all other nodes v:

```
7: for all rounds i ∈ 1, ..., f + 1 do
8:    S is the set of accepted messages value(1)_u.
9:    if |S| ≥ i and value(1)_p ∈ S then
10:      broadcast S ∪ {value(1)_v}
11:      decide 1 and terminate
12:   end if
13: end for
14: decide 0 and terminate
```

Theorem 18.3. *Algorithm 18.2 can tolerate* $f < n$ *byzantine failures while terminating in* $f + 1$ *rounds.*

Proof. Assuming that the primary p is not byzantine and its input is 1, then p broadcasts $\texttt{value}(1)_p$ in the first round, which will trigger all correct nodes to decide for 1. If p's input is 0, there is no signed message $\texttt{value}(1)_p$, and no node can decide for 1.

If primary p is byzantine, we need all correct nodes to decide for the same value for the algorithm to be correct. Let us assume that p convinces a correct node v that its value is 1 in round i with $i < f + 1$. We know that v received i signed messages for value 1. Then, v will broadcast $i + 1$ signed messages for value 1, which will trigger all correct nodes to also decide for 1. If p tries to convince some node v late (in round $i = f + 1$), v must receive $f + 1$ signed messages. Since at most f nodes are byzantine, at least one correct node u signed a message $\texttt{value}(1)_u$ in some round $i < f + 1$, which puts us back to the previous case. □

Remarks:

- The algorithm only takes $f + 1$ rounds, which is optimal as described in Theorem 17.20.
- Using signatures, Algorithm 18.2 solves consensus for any number of failures! Does this contradict Theorem 17.12? Recall that in the proof of Theorem 17.12 we assumed that a byzantine node can distribute contradictory information about its own input. If messages are signed, correct nodes can detect such behavior – a node u signing two contradicting messages proves to all nodes that node u is byzantine.
- Does Algorithm 18.2 satisfy any of the validity conditions introduced in Section 17.1? No! A byzantine primary can dictate the decision

value. Can we modify the algorithm such that the correct-input validity condition is satisfied? Yes! We can run the algorithm in parallel for $2f+1$ primary nodes. Either 0 or 1 will occur at least $f+1$ times, which means that one correct process had to have this value in the first place. In this case, we can only handle $f < \frac{n}{2}$ byzantine nodes.

- In reality, a primary will usually be correct. If so, Algorithm 18.2 only needs two rounds! Can we make it work with arbitrary inputs? Also, relying on synchrony limits the practicality of the protocol. What if messages can be lost or the system is asynchronous?

- Zyzzyva uses authenticated messages to achieve state replication, as in Definition 15.8. It is designed to run fast when nodes run correctly, and it will slow down to fix failures!

18.2 Zyzzyva

Definition 18.4 (View)**.** *A view V describes the current state of a replicated system, enumerating the $3f+1$ replicas. The view V also marks one of the replicas as the primary p.*

Definition 18.5 (Command)**.** *If a client wants to update (or read) data, it sends a suitable command c in a* `Request` *message to the primary p. Apart from the command c itself, the* `Request` *message also includes a timestamp t. The client signs the message to guarantee authenticity.*

Definition 18.6 (History)**.** *The history h is a sequence of commands $c_1, c_2, \ldots$ in the order they are executed by Zyzzyva. We denote the history up to c_k with h_k.*

Remarks:

- In Zyzzyva, the primary p is used to order commands submitted by clients to create a history h.

- Apart from the globally accepted history, node u may also have a local history, which we denote as h^u or h^u_k.

Definition 18.7 (Complete command)**.** *If a command completes, it will remain in its place in the history h even in the presence of failures.*

Remarks:

- As long as clients wait for the completion of their commands, clients can treat Zyzzyva like one single computer even if there are up to f failures.

In the Absence of Failures

Algorithm 18.8 Zyzzyva: No failures

1: At time t client u wants to execute command c
2: Client u sends request $\texttt{R} = \texttt{Request}(c,t)_u$ to primary p
3: Primary p appends c to its local history, i.e., $h^p = (h^p, c)$
4: Primary p sends $\texttt{OR} = \texttt{OrderedRequest}(h^p, c, \texttt{R})_p$ to all replicas
5: Each replica r appends command c to local history $h^r = (h^r, c)$ and checks whether $h^r = h^p$
6: Each replica r runs command c_k and obtains result a
7: Each replica r sends $\texttt{Response}(a,\texttt{OR})_r$ to client u
8: Client u collects the set S of received $\texttt{Response}(a,\texttt{OR})_r$ messages
9: Client u checks if all histories h^r are consistent
10: **if** $|S| = 3f+1$ **then**
11: Client u considers command c to be complete
12: **end if**

Remarks:

- Since the client receives $3f+1$ consistent responses, all correct replicas have to be in the same state.

- Only three communication rounds are required for the command c to complete.

- Note that replicas have no idea which commands are considered complete by clients! How can we make sure that commands that are considered complete by a client are actually executed? We will see in Theorem 18.23.

- Commands received from clients should be ordered according to timestamps to preserve the causal order of commands.

- There is a lot of optimization potential. For example, including the entire command history in most messages introduces prohibitively large overhead. Rather, old parts of the history that are agreed upon can be truncated. Also, sending a hash value of the remainder of the history is enough to check its consistency across replicas.

- What if a client does not receive $3f+1$ $\texttt{Response}(a,\texttt{OR})_r$ messages? A byzantine replica may omit sending anything at all! In practice, clients set a timeout for the collection of `Response` messages. Does this mean that Zyzzyva only works in the synchronous model? Yes and no. We will discuss this in Lemma 18.26 and Lemma 18.27.

Byzantine Replicas

Algorithm 18.9 Zyzzyva: Byzantine Replicas (append to Algorithm 18.8)

```
1: if 2f + 1 ≤ |S| < 3f + 1 then
2:     Client u sends Commit(S)_u to all replicas
3:     Each replica r replies with a LocalCommit(S)_r message to u
4:     Client u collects at least 2f + 1 LocalCommit(S)_r messages and considers
       c to be complete
5: end if
```

Remarks:

- If replicas fail, a client u may receive less than $3f + 1$ consistent responses from the replicas. Client u can only assume command c to be complete if all correct replicas r eventually append command c to their local history h^r.

Definition 18.10 (Commit Certificate)**.** *A commit certificate S contains $2f+1$ consistent and signed* $\texttt{Response}(a,\texttt{OR})_r$ *messages from $2f + 1$ different replicas r.*

Remarks:

- The set S is a commit certificate which proves the execution of the command on $2f + 1$ replicas, of which at least $f + 1$ are correct. This commit certificate S must be acknowledged by $2f + 1$ replicas before the client considers the command to be complete.
- Why do clients have to distribute this commit certificate to $2f + 1$ replicas? We will discuss this in Theorem 18.21.
- What if $|S| < 2f + 1$, or what if the client receives $2f + 1$ messages but some have inconsistent histories? Since at most f replicas are byzantine, the primary itself must be byzantine! Can we resolve this?

Byzantine Primary

Definition 18.11 (Proof of Misbehavior)**.** *Proof of misbehavior of some node can be established by a set of contradicting signed messages.*

Remarks:

- For example, if a client u receives two $\texttt{Response}(a,\texttt{OR})_r$ messages that contain inconsistent `OR` messages signed by the primary, client u can prove that the primary misbehaved. Client u broadcasts this proof of misbehavior to all replicas r which initiate a view change by broadcasting a $\texttt{IHatePrimary}_r$ message to all replicas.

Algorithm 18.12 Zyzzyva: Byzantine Primary (append to Algorithm 18.9)

```
1: if |S| < 2f + 1 then
2:     Client u sends the original R = Request(c,t)_u to all replicas
3:     Each replica r sends a ConfirmRequest(R)_r message to p
4:     if primary p replies with OR then
5:         Replica r forwards OR to all replicas
6:         Continue as in Algorithm 18.8, Line 5
7:     else
8:         Replica r initiates view change by broadcasting IHatePrimary_r to all
           replicas
9:     end if
10: end if
```

Remarks:

- A faulty primary can slow down Zyzzyva by not sending out the `OrderedRequest` messages in Algorithm 18.8, repeatedly escalating to Algorithm 18.12.
- Line 5 in the Algorithm is necessary to ensure liveness. We will discuss this in Theorem 18.27.
- Again, there is potential for optimization. For example, a replica might already know about a command that is requested by a client. In that case, it can answer without asking the primary. Furthermore, the primary might already know the message `R` requested by the replicas. In that case, it sends the old `OR` message to the requesting replica.

Safety

Definition 18.13 (Safety)**.** *We call a system safe if the following condition holds: If a command with sequence number j and a history h_j completes, then for any command that completed earlier (with a smaller sequence number $i < j$), the history h_i is a prefix of history h_j.*

Remarks:

- In Zyzzyva a command can only complete in two ways, either in Algorithm 18.8 or in Algorithm 18.9.
- If a system is safe, complete commands cannot be reordered or dropped. So is Zyzzyva so far safe?

Lemma 18.14. *Let c_i and c_j be two different complete commands. Then c_i and c_j must have different sequence numbers.*

Proof. If a command c completes in Algorithm 18.8, $3f + 1$ replicas sent a $\texttt{Response}(a,\texttt{OR})_r$ to the client. If the command c completed in Algorithm 18.9, at least $2f + 1$ replicas sent a $\texttt{Response}(a,\texttt{OR})_r$ message to the client. Hence, a client has to receive at least $2f + 1$ $\texttt{Response}(a,\texttt{OR})_r$ messages.

Both c_i and c_j are complete. Therefore there must be at least $2f+1$ replicas that responded to c_i with a $\texttt{Response}(a,\texttt{OR})_r$ message. But there are also at least

$2f+1$ replicas that responded to c_j with a $\texttt{Response}(a,\texttt{OR})_r$ message. Because there are only $3f+1$ replicas, there is at least one correct replica that sent a $\texttt{Response}(a,\texttt{OR})_r$ message for both c_i and c_j. A correct replica only sends one $\texttt{Response}(a,\texttt{OR})_r$ message for each sequence number, hence the two commands must have different sequence numbers. □

Lemma 18.15. *Let c_i and c_j be two complete commands with sequence numbers $i < j$. The history h_i is a prefix of h_j.*

Proof. As in the proof of Lemma 18.14, there has to be at least one correct replica that sent a $\texttt{Response}(a,\texttt{OR})_r$ message for both c_i and c_j.

A correct replica r that sent a $\texttt{Response}(a,\texttt{OR})_r$ message for c_i will only accept c_j if the history for c_j provided by the primary is consistent with the local history of replica r, including c_i. □

Remarks:

- A byzantine primary can cause the system to never complete any command. Either by never sending any messages or by inconsistently ordering client requests. In this case, replicas have to replace the primary.

View Changes

Definition 18.16 (View Change)**.** *In Zyzzyva, a view change is used to replace a byzantine primary with another (hopefully correct) replica. View changes are initiated by replicas sending $\texttt{IHatePrimary}_r$ to all other replicas. This only happens if a replica obtains a valid proof of misbehavior from a client or after a replica fails to obtain an* `OR` *message from the primary in Algorithm 18.12.*

Remarks:

- How can we safely decide to initiate a view change, i.e. demote a byzantine primary? Note that byzantine nodes should not be able to trigger a view change!

Algorithm 18.17 Zyzzyva: View Change Agreement

1: All replicas continuously collect the set H of $\texttt{IHatePrimary}_r$ messages
2: **if** a replica r received $|H| > f$ messages or a valid `ViewChange` message **then**
3: Replica r broadcasts $\texttt{ViewChange}(H^r,h^r,S_l^r)_r$
4: Replica r stops participating in the current view
5: Replica r switches to the next primary "$p = p+1$"
6: **end if**

Remarks:

- The $f+1$ $\texttt{IHatePrimary}_r$ messages in set H prove that at least one correct replica initiated a view change. This proof is broadcast to all replicas to make sure that once the first correct replica stopped acting in the current view, all other replicas will do so as well.
- S_l^r is the most recent commit certificate that the replica obtained in the ending view as described in Algorithm 18.9. S_l^r will be used to recover the correct history before the new view starts. The local histories h^r are included in the $\texttt{ViewChange}(H^r,h^r,S_l^r)_r$ message such that commands that completed after a correct client received $3f+1$ responses from replicas can be recovered as well.
- In Zyzzyva, a byzantine primary starts acting as a normal replica after a view change. In practice, all machines eventually break and rarely fix themselves after that. Instead, one could consider to replace a byzantine primary with a fresh replica that was not in the previous view.

Algorithm 18.18 Zyzzyva: View Change Execution

1: The new primary p collects the set C of $\texttt{ViewChange}(H^r,h^r,S_l^r)_r$ messages
2: **if** new primary p collected $|C| \geq 2f+1$ messages **then**
3: New primary p sends $\texttt{NewView}(C)_p$ to all replicas
4: **end if**

5: **if** a replica r received a $\texttt{NewView}(C)_p$ message **then**
6: Replica r recovers new history $h_{\texttt{new}}$ as shown in Algorithm 18.20
7: Replica r broadcasts $\texttt{ViewConfirm}(h_{\texttt{new}})_r$ message to all replicas
8: **end if**

9: **if** a replica r received $2f+1$ $\texttt{ViewConfirm}(h_{\texttt{new}})_r$ messages **then**
10: Replica r accepts $h^r = h_{\texttt{new}}$ as the history of the new view
11: Replica r starts participating in the new view
12: **end if**

Remarks:

- Analogously to Lemma 18.15, commit certificates are ordered. For two commit certificates S_i and S_j with sequence numbers $i < j$, the history h_i certified by S_i is a prefix of the history h_j certified by S_j.
- Zyzzyva collects the most recent commit certificate and the local history of $2f+1$ replicas. This information is distributed to all replicas, and used to recover the history for the new view h_{new}.
- If a replica does not receive the $\texttt{NewView}(C)_p$ or the $\texttt{ViewConfirm}(h_{\texttt{new}})_r$ message in time, it triggers another view change by broadcasting $\texttt{IHatePrimary}_r$ to all other replicas.

- How is the history recovered exactly? It seems that the set of histories included in C can be messy. How can we be sure that complete commands are not reordered or dropped?

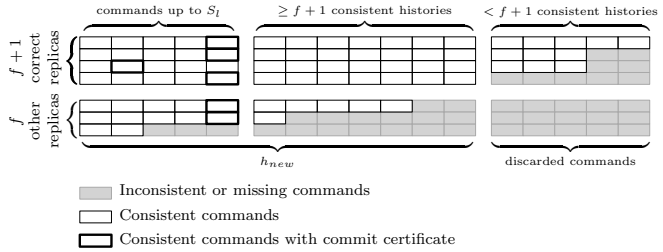


Figure 18.19: The structure of the data reported by different replicas in C. Commands up to the last commit certificate S_l were completed in either Algorithm 18.8 or Algorithm 18.9. After the last commit certificate S_l there may be commands that completed at a correct client in Algorithm 18.8. Algorithm 18.20 shows how the new history h_{new} is recovered such that no complete commands are lost.

Algorithm 18.20 Zyzzyva: History Recovery

```
1: C = set of 2f + 1 ViewChange(H^r,h^r,S^r)_r messages in NewView(C)_p
2: R = set of replicas included in C
3: S_l = most recent commit certificate S_l^r reported in C
4: h_new = history h_l contained in S_l
5: k = l + 1, next sequence number
6: while command c_k exists in C do
7:    if c_k is reported by at least f + 1 replicas in R then
8:       Remove replicas from R that do not support c_k
9:       h_new = (h_new, c_k)
10:   end if
11:   k = k + 1
12: end while
13: return h_new
```

Remarks:

- Commands up to S_l are included into the new history h_{new}.
- If at least $f+1$ replicas share a consistent history after the last commit certificate S_l, also the commands after that are included.
- Even if $f+1$ correct replicas consistently report a command c after the last commit certificate S_l, c may not be considered complete by a client, e.g., because one of the responses to the client was lost. Such a command is included in the new history h_{new}. When the client retries executing c, the replicas will be able to identify the same command c using the timestamp included in the client's request, and avoid duplicate execution of the command.
- Can we be sure that all commands that completed at a correct client are carried over into the new view?

Lemma 18.21. *The globally most recent commit certificate S_l is included in C.*

Proof. Any two sets of $2f+1$ replicas share at least one correct replica. Hence, at least one correct replica which acknowledged the most recent commit certificate S_l also sent a $\texttt{LocalCommit}(S_l)_r$ message that is in C. □

Lemma 18.22. *Any command and its history that completes after S_l has to be reported in C at least $f+1$ times.*

Proof. A command c can only complete in Algorithm 18.8 after S_l. Hence, $3f+1$ replicas sent a $\texttt{Response}(a,\texttt{OR})_r$ message for c. C includes the local histories of $2f+1$ replicas of which at most f are byzantine. As a result, c and its history is consistently found in at least $f+1$ local histories in C. □

Lemma 18.23. *If a command c is considered complete by a client, command c remains in its place in the history during view changes.*

Proof. We have shown in Lemma 18.21 that the most recent commit certificate is contained in C, and hence any command that terminated in Algorithm 18.9 is included in the new history after a view change. Every command that completed before the last commit certificate S_l is included in the history as a result. Commands that completed in Algorithm 18.8 after the last commit certificate are supported by at least $f+1$ correct replicas as shown in Lemma 18.22. Such commands are added to the new history as described in Algorithm 18.20. Algorithm 18.20 adds commands sequentially until the histories become inconsistent. Hence, complete commands are not lost or reordered during a view change. □

Theorem 18.24. *Zyzzyva is safe even during view changes.*

Proof. Complete commands are not reordered within a view as described in Lemma 18.15. Also, no complete command is lost or reordered during a view change as shown in Lemma 18.23. Hence, Zyzzyva is safe. □

Remarks:

- So Zyzzyva correctly handles complete commands even in the presence of failures. We also want Zyzzyva to make progress, i.e., commands issued by correct clients should complete eventually.
- If the network is broken or introduces arbitrarily large delays, commands may never complete.
- Can we be sure commands complete in periods in which delays are bounded?

Definition 18.25 (Liveness)**.** *We call a system* live *if every command eventually completes.*

Lemma 18.26. *Zyzzyva is live during periods of synchrony if the primary is correct and a command is requested by a correct client.*

Proof. The client receives a $\texttt{Response}(a,\texttt{OR})_r$ message from all correct replicas. If it receives $3f+1$ messages, the command completes immediately in Algorithm 18.8. If the client receives fewer than $3f+1$ messages, it will at least receive $2f+1$, since there are at most f byzantine replicas. All correct replicas will answer the client's $\texttt{Commit}(S)_u$ message with a correct $\texttt{LocalCommit}(S)_r$ message after which the command completes in Algorithm 18.9. □

Lemma 18.27. *If, during a period of synchrony, a request does not complete in Algorithm 18.8 or Algorithm 18.9, a view change occurs.*

Proof. If a command does not complete for a sufficiently long time, the client will resend the $\texttt{R} = \texttt{Request}(c,t)_u$ message to all replicas. After that, if a replica's $\texttt{ConfirmRequest}(\texttt{R})_r$ message is not answered in time by the primary, it broadcasts an $\texttt{IHatePrimary}_r$ message. If a correct replica gathers $f+1$ $\texttt{IHatePrimary}_r$ messages, the view change is initiated. If no correct replica collects more than f $\texttt{IHatePrimary}_r$ messages, at least one correct replica received a valid $\texttt{OrderedRequest}(h^p, c, \texttt{R})_p$ message from the primary which it forwards to all other replicas. In that case, the client is guaranteed to receive at least $2f+1$ $\texttt{Response}(a,\texttt{OR})_r$ messages from the correct replicas and can complete the command by assembling a commit certificate. □

Remarks:

- If the newly elected primary is byzantine, the view change may never terminate. However, we can detect if the new primary does not assemble C correctly as all contained messages are signed. If the primary refuses to assemble C, replicas initiate another view change after a timeout.

Chapter Notes

Algorithm 18.2 was introduced by Dolev et al. [DFF+82] in 1982. Byzantine fault tolerant state machine replication (BFT) is a problem that gave rise to various protocols. Castro and Liskov [MC99] introduced the Practical Byzantine Fault Tolerance (PBFT) protocol in 1999, applications such as Farsite [ABC+02] followed. This triggered the development of, e.g., Q/U [AEMGG+05] and HQ [CML+06]. Zyzzyva [KAD+07] improved on performance especially in the case of no failures, while Aardvark [CWA+09] improved performance in the presence of failures. Guerraoui at al. [GKQV10] introduced a modular system which allows to more easily develop BFT protocols that match specific applications in terms of robustness or best case performance.

This chapter was written in collaboration with Pascal Bissig.

Bibliography

[ABC+02] Atul Adya, William J. Bolosky, Miguel Castro, Gerald Cermak, Ronnie Chaiken, John R. Douceur, Jon Howell, Jacob R. Lorch, Marvin Theimer, and Roger P. Wattenhofer. Farsite: Federated, available, and reliable storage for an incompletely trusted environment. *SIGOPS Oper. Syst. Rev.*, 36(SI):1–14, December 2002.

[AEMGG+05] Michael Abd-El-Malek, Gregory R Ganger, Garth R Goodson, Michael K Reiter, and Jay J Wylie. Fault-scalable byzantine fault-tolerant services. *ACM SIGOPS Operating Systems Review*, 39(5):59–74, 2005.

[CML+06] James Cowling, Daniel Myers, Barbara Liskov, Rodrigo Rodrigues, and Liuba Shrira. Hq replication: A hybrid quorum protocol for byzantine fault tolerance. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation*, OSDI '06, pages 177–190, Berkeley, CA, USA, 2006. USENIX Association.

[CWA+09] Allen Clement, Edmund L Wong, Lorenzo Alvisi, Michael Dahlin, and Mirco Marchetti. Making byzantine fault tolerant systems tolerate byzantine faults. In *NSDI*, volume 9, pages 153–168, 2009.

[DFF+82] Danny Dolev, Michael J Fischer, Rob Fowler, Nancy A Lynch, and H Raymond Strong. An efficient algorithm for byzantine agreement without authentication. *Information and Control*, 52(3):257–274, 1982.

[GKQV10] Rachid Guerraoui, Nikola Knežević, Vivien Quéma, and Marko Vukolić. The next 700 bft protocols. In *Proceedings of the 5th European conference on Computer systems*, pages 363–376. ACM, 2010.

[KAD+07] Ramakrishna Kotla, Lorenzo Alvisi, Mike Dahlin, Allen Clement, and Edmund Wong. Zyzzyva: speculative byzantine fault tolerance. In *ACM SIGOPS Operating Systems Review*, volume 41, pages 45–58. ACM, 2007.

[MC99] Barbara Liskov Miguel Castro. Practical byzantine fault tolerance. In *OSDI*, volume 99, pages 173–186, 1999.

Chapter 19

Quorum Systems

What happens if a single server is no longer powerful enough to service all your customers? The obvious choice is to add more servers and to use the majority approach (e.g. Paxos, Chapter 15) to guarantee consistency. However, even if you buy one million servers, a client still has to access more than half of them per request! While you gain fault-tolerance, your efficiency can at most be doubled. Do we have to give up on consistency?

Let us take a step back: We used majorities because majority sets always overlap. But are majority sets the only sets that guarantee overlap? In this chapter we study the theory behind overlapping sets, known as quorum systems.

Definition 19.1 (quorum, quorum system)**.** *Let $V = \{v_1, \ldots, v_n\}$ be a set of nodes. A **quorum** $Q \subseteq V$ is a subset of these nodes. A **quorum system** $\mathcal{S} \subset 2^V$ is a set of quorums s.t. every two quorums intersect, i.e., $Q_1 \cap Q_2 \neq \emptyset$ for all $Q_1, Q_2 \in \mathcal{S}$.*

Remarks:

- When a quorum system is being used, a client selects a quorum, acquires a lock (or ticket) on all nodes of the quorum, and when done releases all locks again. The idea is that no matter which quorum is chosen, its nodes will intersect with the nodes of every other quorum.
- What can happen if two quorums try to lock their nodes at the same time?
- A quorum system $\mathcal{S}$ is called **minimal** if $\forall Q_1, Q_2 \in \mathcal{S} : Q_1 \not\subset Q_2$.
- The simplest quorum system imaginable consists of just one quorum, which in turn just consists of one server. It is known as **Singleton**.
- In the **Majority** quorum system, every quorum has $\lfloor \frac{n}{2} \rfloor + 1$ nodes.
- Can you think of other simple quorum systems?

19.1 Load and Work

Definition 19.2 (access strategy)**.** *An **access strategy** Z defines the probability $P_Z(Q)$ of accessing a quorum $Q \in \mathcal{S}$ s.t. $\sum_{Q \in \mathcal{S}} P_Z(Q) = 1$.*

Definition 19.3 (load)**.**

- *The **load** of access strategy Z on a node v_i is $L_Z(v_i) = \sum_{Q \in \mathcal{S}; v_i \in Q} P_Z(Q)$.*
- *The **load** induced by access strategy Z on a quorum system $\mathcal{S}$ is the maximal load induced by Z on any node in $\mathcal{S}$, i.e., $L_Z(\mathcal{S}) = \max_{v_i \in \mathcal{S}} L_Z(v_i)$.*
- *The **load** of a quorum system $\mathcal{S}$ is $L(\mathcal{S}) = \min_Z L_Z(\mathcal{S})$.*

Definition 19.4 (work)**.**

- *The **work** of a quorum $Q \in \mathcal{S}$ is the number of nodes in Q, $W(Q) = |Q|$.*
- *The **work** induced by access strategy Z on a quorum system $\mathcal{S}$ is the expected number of nodes accessed, i.e., $W_Z(\mathcal{S}) = \sum_{Q \in \mathcal{S}} P_Z(Q) \cdot W(Q)$.*
- *The **work** of a quorum system $\mathcal{S}$ is $W(\mathcal{S}) = \min_Z W_Z(\mathcal{S})$.*

Remarks:

- Note that you cannot choose different access strategies Z for work and load, you have to pick a single Z for both.
- We illustrate the above concepts with a small example. Let $V = \{v_1, v_2, v_3, v_4, v_5\}$ and $\mathcal{S} = \{Q_1, Q_2, Q_3, Q_4\}$, with $Q_1 = \{v_1, v_2\}$, $Q_2 = \{v_1, v_3, v_4\}$, $Q_3 = \{v_2, v_3, v_5\}$, $Q_4 = \{v_2, v_4, v_5\}$. If we choose the access strategy Z s.t. $P_Z(Q_1) = 1/2$ and $P_Z(Q_2) = P_Z(Q_3) = P_Z(Q_4) = 1/6$, then the node with the highest load is v_2 with $L_Z(v_2) = 1/2 + 1/6 + 1/6 = 5/6$, i.e., $L_Z(\mathcal{S}) = 5/6$. Regarding work, we have $W_Z(\mathcal{S}) = 1/2 \cdot 2 + 1/6 \cdot 3 + 1/6 \cdot 3 + 1/6 \cdot 3 = 15/6$.
- Can you come up with a better access strategy for $\mathcal{S}$?
- If every quorum Q in a quorum system $\mathcal{S}$ has the same number of elements, $\mathcal{S}$ is called *uniform*.
- What is the minimum load a quorum system can have?

Primary Copy vs. Majority		**Singleton**	**Majority**
How many nodes need to be accessed?	(Work)	**1**	$> n/2$
What is the load of the busiest node?	(Load)	1	$> \mathbf{1/2}$

Table 19.5: First comparison of the Singleton and Majority quorum systems. Note that the Singleton quorum system can be a good choice when the failure probability of every single node is $> 1/2$.

Theorem 19.6. *Let $\mathcal{S}$ be a quorum system. Then $L(\mathcal{S}) \geq 1/\sqrt{n}$ holds.*

Proof. Let $Q = \{v_1, \ldots, v_q\}$ be a quorum of minimal size in $\mathcal{S}$, with sizes $|Q| = q$ and $|\mathcal{S}| = s$. Let Z be an access strategy for $\mathcal{S}$. Every other quorum in $\mathcal{S}$ intersects in at least one element with this quorum Q. Each time a quorum is accessed, at least one node in Q is accessed as well, yielding a lower bound of $L_Z(v_i) \geq 1/q$ for some $v_i \in Q$.

Furthermore, as Q is minimal, at least q nodes need to be accessed, yielding $W(\mathcal{S}) \geq q$. Thus, $L_Z(v_i) \geq q/n$ for some $v_i \in Q$, as each time q nodes are accessed, the load of the most accessed node is at least q/n.

Combining both ideas leads to $L_Z(\mathcal{S}) \geq \max(1/q, q/n) \Rightarrow L_Z(\mathcal{S}) \geq 1/\sqrt{n}$. Thus, $L(\mathcal{S}) \geq 1/\sqrt{n}$, as Z can be *any* access strategy. □

Remarks:

- Can we achieve this load?

19.2 Grid Quorum Systems

Definition 19.7 (Basic Grid quorum system)**.** *Assume $\sqrt{n} \in \mathbb{N}$, and arrange the n nodes in a square matrix with side length of $\sqrt{n}$, i.e., in a grid. The basic **Grid** quorum system consists of $\sqrt{n}$ quorums, with each containing the full row i and the full column i, for $1 \leq i \leq \sqrt{n}$.*

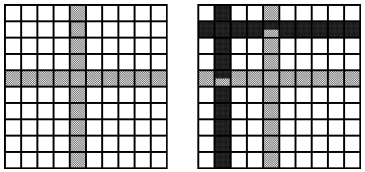


Figure 19.8: The basic version of the Grid quorum system, where each quorum Q_i with $1 \leq i \leq \sqrt{n}$ uses row i and column i. The size of each quorum is $2\sqrt{n}-1$ and two quorums overlap in exactly two nodes. Thus, when the access strategy Z is uniform (i.e., the probability of each quorum is $1/\sqrt{n}$), the work is $2\sqrt{n}-1$, and the load of every node is in $\Theta(1/\sqrt{n})$.

Remarks:

- Consider the right picture in Figure 19.8: The two quorums intersect in two nodes. If both quorums were to be accessed at the same time, it is not guaranteed that at least one quorum will lock all of its nodes, as they could enter a deadlock!
- In the case of just two quorums, one could solve this by letting the quorums just intersect in one node, see Figure 19.9. However, already with three quorums the same situation could occur again, progress is not guaranteed!

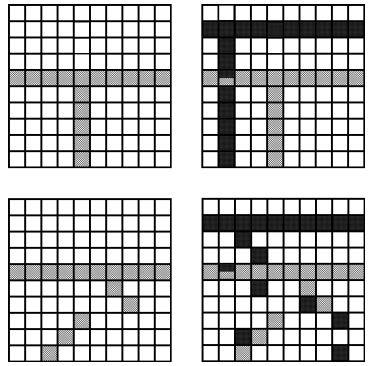


Figure 19.9: There are other ways to choose quorums in the grid s.t. pairwise different quorums only intersect in one node. The size of each quorum is between $\sqrt{n}$ and $2\sqrt{n}-1$, i.e., the work is in $\Theta(\sqrt{n})$. When the access strategy Z is uniform, the load of every node is in $\Theta(1/\sqrt{n})$.

Algorithm 19.10 Sequential Locking Strategy for a Quorum Q

1: Attempt to lock the nodes one by one, ordered by their identifiers
2: Should a node be already locked, release all locks and start over

- However, by deviating from the "access all at once" strategy, we can guarantee progress if the nodes are totally ordered!

Theorem 19.11. *If each quorum is accessed by Algorithm 19.10, at least one quorum will obtain a lock for all of its nodes.*

Proof. We prove the theorem by contradiction. Assume no quorum can make progress, i.e., for every quorum we have: At least one of its nodes is locked by another quorum. Let v be the node with the highest identifier that is locked by some quorum Q. Observe that Q already locked all of its nodes with a smaller identifier than v, otherwise Q would have restarted. As all nodes with a higher identifier than v are not locked, Q either has locked all of its nodes or can make progress – a contradiction. As the set of nodes is finite, one quorum will eventually be able to lock all of its nodes. □

Remarks:

- But now we are back to sequential accesses in a distributed system? Let's do it concurrently with the same idea, i.e., resolving conflicts by the ordering of the nodes. Then, a quorum that locked the highest identifier so far can always make progress!

Theorem 19.13. *If the nodes and quorums use Algorithm 19.12, at least one quorum will obtain a lock for all of its nodes.*

Algorithm 19.12 Concurrent Locking Strategy for a Quorum Q

Invariant: Let $v_Q \in Q$ be the highest identifier of a node locked by Q s.t. all nodes $v_i \in Q$ with $v_i < v_Q$ are locked by Q as well. Should Q not have any lock, then v_Q is set to 0.

```
1: repeat
2:   Attempt to lock all nodes of the quorum Q
3:   for each node v ∈ Q that was not able to be locked by Q do
4:     exchange v_Q and v_Q' with the quorum Q' that locked v
5:     if v_Q > v_Q' then
6:       Q' releases lock on v and Q acquires lock on v
7:     end if
8:   end for
9: until all nodes of the quorum Q are locked
```

Proof. The proof is analogous to the proof of Theorem 19.11: Assume for contradiction that no quorum can make progress. However, at least the quorum with the highest v_Q can always make progress – a contradiction! As the set of nodes is finite, at least one quorum will eventually be able to acquire a lock on all of its nodes. □

Remarks:

- What if a quorum locks all of its nodes and then crashes? Is the quorum system dead now? This issue can be prevented by, e.g., using leases instead of locks: leases have a timeout, i.e., a lock is released eventually.

19.3 Fault Tolerance

Definition 19.14 (resilience)**.** *If any f nodes from a quorum system $\mathcal{S}$ can fail s.t. there is still a quorum $Q \in \mathcal{S}$ without failed nodes, then $\mathcal{S}$ is f-*resilient*. The largest such f is the **resilience** $R(\mathcal{S})$.*

Theorem 19.15. *Let $\mathcal{S}$ be a Grid quorum system where each of the n quorums consists of a full row and a full column. $\mathcal{S}$ has a resilience of $\sqrt{n}-1$.*

Proof. If all $\sqrt{n}$ nodes on the diagonal of the grid fail, then every quorum will have at least one failed node. Should less than $\sqrt{n}$ nodes fail, then there is a row and a column without failed nodes. □

Definition 19.16 (failure probability)**.** *Assume that every node works with a fixed probability p (in the following we assume concrete values, e.g. $p > 1/2$). The **failure probability** $F_p(\mathcal{S})$ of a quorum system $\mathcal{S}$ is the probability that at least one node of every quorum fails.*

Remarks:

- The **asymptotic failure probability** is $F_p(\mathcal{S})$ for $n \to \infty$.

Facts 19.17. *A version of a **Chernoff bound** states the following:*
Let $x_1, \ldots, x_n$ be independent Bernoulli-distributed random variables with $Pr[x_i = 1] = p_i$ and $Pr[x_i = 0] = 1 - p_i = q_i$, then for $X := \sum_{i=1}^{n} x_i$ and $\mu := \mathbb{E}[X] = \sum_{i=1}^{n} p_i$ the following holds:

$$\text{for all } 0 < \delta < 1\text{: } Pr[X \leq (1-\delta)\mu] \leq e^{-\mu\delta^2/2} \ .$$

Theorem 19.18. *The asymptotic failure probability of the Majority quorum system is 0.*

Proof. In a Majority quorum system each quorum contains exactly $\lfloor \frac{n}{2} \rfloor + 1$ nodes and each subset of nodes with cardinality $\lfloor \frac{n}{2} \rfloor + 1$ forms a quorum. The Majority quorum system fails, if only $\lfloor \frac{n}{2} \rfloor$ nodes work. Otherwise there is at least one quorum available. In order to calculate the failure probability we define the following random variables:

$$x_i = \begin{cases} 1, & \text{if node } i \text{ works, happens with probability } p \\ 0, & \text{if node } i \text{ fails, happens with probability } q = 1-p \end{cases}$$

and $X := \sum_{i=1}^{n} x_i$, with $\mu = np$,

whereas X corresponds to the number of working nodes. To estimate the probability that the number of working nodes is less than $\lfloor \frac{n}{2} \rfloor + 1$ we will make use of the Chernoff inequality from above. By setting $\delta = 1 - \frac{1}{2p}$ we obtain $F_P(\mathcal{S}) = Pr[X \leq \lfloor \frac{n}{2} \rfloor] \leq Pr[X \leq \frac{n}{2}] = Pr[X \leq (1-\delta)\mu]$.

With $\delta = 1 - \frac{1}{2p}$ we have $0 < \delta \leq 1/2$ due to $1/2 < p \leq 1$. Thus, we can use the Chernoff bound and get $F_P(\mathcal{S}) \leq e^{-\mu\delta^2/2} \in e^{-\Omega(n)}$. □

Theorem 19.19. *The asymptotic failure probability of the Grid quorum system is 1.*

Proof. Consider the $n = d \cdot d$ nodes to be arranged in a $d \times d$ grid. A quorum always contains one full row. In this estimation we will make use of the Bernoulli inequality which states that for all $n \in \mathbb{N}, x \geq -1 : (1+x)^n \geq 1 + nx$.

The system fails, if in each row at least one node fails (which happens with probability $1 - p^d$ for a particular row, as all nodes work with probability p^d). Therefore we can bound the failure probability from below with:

$F_p(\mathcal{S}) \geq Pr[\text{at least one failure per row}] = (1-p^d)^d \geq 1 - dp^d \underset{n\to\infty}{\longrightarrow} 1$. □

Remarks:

- Now we have a quorum system with optimal load (the Grid) and one with fault-tolerance (Majority), but what if we want both?

Definition 19.20 (B-Grid quorum system)**.** *Consider $n = dhr$ nodes, arranged in a rectangular grid with $h \cdot r$ rows and d columns. Each group of r rows is a band, and r elements in a column restricted to a band are called a mini-column. A quorum consists of one mini-column in every band and one element from each mini-column of one band; thus every quorum has $d + hr - 1$ elements. The **B-Grid** quorum system consists of all such quorums.*

Theorem 19.22. *The asymptotic failure probability of the B-Grid quorum system is 0.*

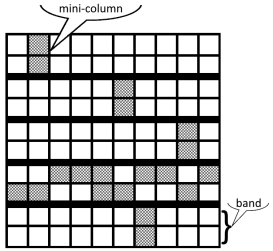


Figure 19.21: A B-Grid quorum system with $n = 100$ nodes, $d = 10$ columns, $h \cdot r = 10$ rows, $h = 5$ bands, and $r = 2$. The depicted quorum has a $d+hr-1 = 10 + 5 \cdot 2 - 1 = 19$ nodes. If the access strategy Z is chosen uniformly, then we have a work of $d + hr - 1$ and a load of $\frac{d+hr-1}{n}$. By setting $d = \sqrt{n}$ and $r = \log n$, we obtain a work of $\Theta(\sqrt{n})$ and a load of $\Theta(1/\sqrt{n})$.

Proof. Suppose $n = dhr$ and the elements are arranged in a grid with d columns and $h \cdot r$ rows. The B-Grid quorum system does fail if in each band a complete mini-column fails, because then it is not possible to choose a band where in each mini-column an element is still working. It also fails if in a band an element in each mini-column fails. Those events may not be independent of each other, but with the help of the union bound, we can upper bound the failure probability with the following equation:

$$\begin{aligned} F_p(\mathcal{S}) &\leq Pr[\text{in every band a complete mini-column fails}] \\ &\quad + Pr[\text{in a band at least one element of every m.-col. fails}] \\ &\leq (d(1-p)^r)^h + h(1-p^r)^d \end{aligned}$$

We use $d = \sqrt{n}, r = \ln d$, and $0 \leq (1-p) \leq 1/3$. Using $n^{\ln x} = x^{\ln n}$, we have $d(1-p)^r \leq d \cdot d^{\ln 1/3} \approx d^{-0.1}$, and hence for large enough d the whole first term is bounded from above by $d^{-0.1h} \ll 1/d^2 = 1/n$.

Regarding the second term, we have $p \geq 2/3$, and $h = d/\ln d < d$. Hence we can bound the term from above by $d(1 - d^{\ln 2/3})^d \approx d(1 - d^{-0.4})^d$. Using $(1 + t/n)^n \leq e^t$, we get (again, for large enough d) an upper bound of $d(1 - d^{-0.4})^d = d(1 - d^{0.6}/d)^d \leq d \cdot e^{-d^{0.6}} = d^{(-d^{0.6}/\ln d)+1} \ll d^{-2} = 1/n$. In total, we have $F_p(\mathcal{S}) \in O(1/n)$. □

	Singleton	**Majority**	**Grid**	**B-Grid***
Work	**1**	$> n/2$	$\Theta(\sqrt{n})$	$\Theta(\sqrt{n})$
Load	1	$> 1/2$	$\Theta(\mathbf{1}/\sqrt{\mathbf{n}})$	$\Theta(\mathbf{1}/\sqrt{\mathbf{n}})$
Resilience	0	$< \mathbf{n/2}$	$\Theta(\sqrt{n})$	$\Theta(\sqrt{n})$
F. Prob.**	$1-p$	$\to \mathbf{0}$	$\to 1$	$\to \mathbf{0}$

Table 19.23: Overview of the different quorum systems regarding resilience, work, load, and their asymptotic failure probability. The best entries in each row are set in bold. * Setting $d = \sqrt{n}$ and $r = \log n$ **Assuming prob. $q = (1-p)$ is constant but significantly less than $1/2$

19.4 Byzantine Quorum Systems

While failed nodes are bad, they are still easy to deal with: just access another quorum where all nodes can respond! Byzantine nodes make life more difficult however, as they can pretend to be a regular node, i.e., one needs more sophisticated methods to deal with them. We need to ensure that the intersection of two quorums always contains a non-byzantine (correct) node and furthermore, the byzantine nodes should not be allowed to infiltrate every quorum. In this section we study three counter-measures of increasing strength, and their implications on the load of quorum systems.

Definition 19.24 (f-disseminating)**.** *A quorum system $\mathcal{S}$ is* f-disseminating *if* (1) *the intersection of two different quorums always contains $f + 1$ nodes, and* (2) *for any set of f byzantine nodes, there is at least one quorum without byzantine nodes.*

Remarks:

- Thanks to (2), even with f byzantine nodes, the byzantine nodes cannot stop all quorums by just pretending to have crashed. At least one quorum will survive. We will also keep this assumption for the upcoming more advanced byzantine quorum systems.
- Byzantine nodes can also do something worse than crashing - they could falsify data! Nonetheless, due to (1), there is at least one non-byzantine node in every quorum intersection. If the data is self-verifying by, e.g., authentication, then this one node is enough.
- If the data is not self-verifying, then we need another mechanism.

Definition 19.25 (f-masking)**.** *A quorum system $\mathcal{S}$ is* f-masking *if* (1) *the intersection of two different quorums always contains $2f + 1$ nodes, and* (2) *for any set of f byzantine nodes, there is at least one quorum without byzantine nodes.*

Remarks:

- Note that except for the second condition, an f-masking quorum system is the same as a $2f$-disseminating system. The idea is that the non-byzantine nodes (at least $f + 1$ can outvote the byzantine ones (at most f), but only if all non-byzantine nodes are up-to-date!
- This raises an issue not covered yet in this chapter. If we access some quorum and update its values, this change still has to be disseminated to the other nodes in the byzantine quorum system. Opaque quorum systems deal with this issue, which are discussed at the end of this section.
- f-disseminating quorum systems need more than $3f$ nodes and f-masking quorum systems need more than $4f$ nodes. Essentially, the quorums may not contain too many nodes, and the different intersection properties lead to the different bounds.

Theorem 19.26. *Let $\mathcal{S}$ be a f-disseminating quorum system. Then $L(\mathcal{S}) \geq \sqrt{(f+1)/n}$ holds.*

Theorem 19.27. *Let $\mathcal{S}$ be a f-masking quorum system. Then $L(\mathcal{S}) \geq \sqrt{(2f+1)/n}$ holds.*

Proofs of Theorems 19.26 and 19.27. The proofs follow the proof of Theorem 19.6, by observing that now not just one element is accessed from a minimal quorum, but $f+1$ or $2f+1$, respectively. □

Definition 19.28 (f-masking Grid quorum system)**.** *A f-masking* Grid *quorum system is constructed as the grid quorum system, but each quorum contains one full column and $f+1$ rows of nodes, with $2f+1 \leq \sqrt{n}$.*

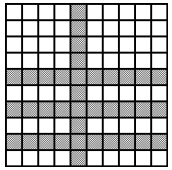


Figure 19.29: An example how to choose a quorum in the f-masking Grid with $f = 2$, i.e., $2+1 = 3$ rows. The load is in $\Theta(f/\sqrt{n})$ when the access strategy is chosen to be uniform. Two quorums overlap by their columns intersecting each other's rows, i.e., they overlap in at least $2f+2$ nodes.

Remarks:

- The f-masking Grid nearly hits the lower bound for the load of f-masking quorum systems, but not quite. A small change and we will be optimal asymptotically.

Definition 19.30 (M-Grid quorum system)**.** *The* M-Grid *quorum system is constructed as the grid quorum as well, but each quorum contains $\sqrt{f+1}$ rows and $\sqrt{f+1}$ columns of nodes, with $f \leq \frac{\sqrt{n}-1}{2}$.*

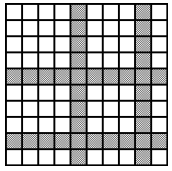


Figure 19.31: An example how to choose a quorum in the M-Grid with $f = 3$, i.e., 2 rows and 2 columns. The load is in $\Theta(\sqrt{f/n})$ when the access strategy is chosen to be uniform. Two quorums overlap with each row intersecting each other's column, i.e., $2\sqrt{f+1}^2 = 2f+2$ nodes.

Corollary 19.32. *The f-masking Grid quorum system and the M-Grid quorum system are f-masking quorum systems.*

Remarks:

- We achieved nearly the same load as without byzantine nodes! However, as mentioned earlier, what happens if we access a quorum that is not up-to-date, except for the intersection with an up-to-date quorum? Surely we can fix that as well without too much loss?
- This property will be handled in the last part of this chapter by *opaque* quorum systems. It will ensure that the number of correct up-to-date nodes accessed will be larger than the number of out-of-date nodes combined with the byzantine nodes in the quorum (cf. (19.33.1)).

Definition 19.33 (f-opaque quorum system)**.** *A quorum system $\mathcal{S}$ is f-opaque if the following two properties hold for any set of f byzantine nodes F and any two different quorums Q_1, Q_2:*

$$|(Q_1 \cap Q_2) \setminus F| > |(Q_2 \cap F) \cup (Q_2 \setminus Q_1)| \tag{19.33.1}$$

$$(F \cap Q) = \emptyset \text{ for some } Q \in \mathcal{S} \tag{19.33.2}$$

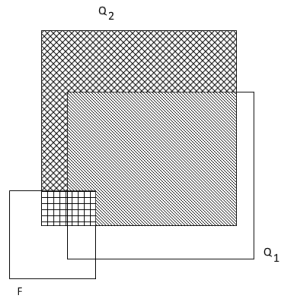


Figure 19.34: Intersection properties of an opaque quorum system. Equation (19.33.1) ensures that the set of non-byzantine nodes in the intersection of Q_1, Q_2 is larger than the set of out of date nodes, even if the byzantine nodes "team up" with those nodes. Thus, the correct up to date value can always be recognized by a majority voting.

Theorem 19.35. *Let $\mathcal{S}$ be a f-opaque quorum system. Then, $n > 5f$.*

Proof. Due to (19.33.2), there exists a quorum Q_1 with size at most $n-f$. With (19.33.1), $|Q_1| > f$ holds. Let F_1 be a set of f (byzantine) nodes $F_1 \subset Q_1$, and with (19.33.2), there exists a $Q_2 \subset V \setminus F_1$. Thus, $|Q_1 \cap Q_2| \leq n - 2f$. With (19.33.1), $|Q_1 \cap Q_2| > f$ holds. Thus, one could choose f (byzantine) nodes F_2 with $F_2 \subset (Q_1 \cap Q_2)$. Using (19.33.1) one can bound $n - 3f$ from below: $n - 3f > |(Q_2 \cap Q_1)| - |F_2| \geq |(Q_2 \cap Q_1) \cup (Q_1 \cap F_2)| \geq |F_1| + |F_2| = 2f$. □

Remarks:

- One can extend the Majority quorum system to be f-opaque by setting the size of each quorum to contain $\lceil (2n+2f)/3 \rceil$ nodes. Then its load is $1/n \lceil (2n+2f)/3 \rceil \approx 2/3 + 2f/3n \geq 2/3$.
- Can we do much better? Sadly, no...

Theorem 19.36. *Let $\mathcal{S}$ be a f-opaque quorum system. Then $L(\mathcal{S}) \geq 1/2$ holds.*

Proof. Equation (19.33.1) implies that for $Q_1, Q_2 \in \mathcal{S}$, the intersection of both Q_1, Q_2 is at least half their size, i.e., $|(Q_1 \cap Q_2)| \geq |Q_1|/2$. Let $\mathcal{S}$ consist of quorums $Q_1, Q_2, \ldots$. The load induced by an access strategy Z on Q_1 is:

$$\sum_{v \in Q_1} \sum_{v \in Q_i} L_Z(Q_i) = \sum_{Q_i} \sum_{v \in (Q_1 \cap Q_i)} L_Z(Q_i) \geq \sum_{Q_i} (|Q_1|/2)\ L_Z(Q_i) = |Q_1|/2 \ .$$

Using the pigeonhole principle, there must be at least one node in Q_1 with load of at least $1/2$. □

Chapter Notes

Historically, a quorum is the minimum number of members of a deliberative body necessary to conduct the business of that group. Their use has inspired the introduction of quorum systems in computer science since the late 1970s/early 1980s. Early work focused on Majority quorum systems [Lam78, Gif79, Tho79], with the notion of minimality introduced shortly after [GB85]. The Grid quorum system was first considered in [Mae85], with the B-Grid being introduced in [NW94]. The latter article and [PW95] also initiated the study of load and resilience.

The f-masking Grid quorum system and opaque quorum systems are from [MR98], and the M-Grid quorum system was introduced in [MRW97]. Both papers also mark the start of the formal study of Byzantine quorum systems. The f-masking and the M-Grid have asymptotic failure probabilities of 1, more complex systems with better values can be found in these papers as well.

Quorum systems have also been extended to cope with nodes dynamically leaving and joining, see, e.g., the dynamic paths quorum system in [NW05].

For a further overview on quorum systems, we refer to the book by Vukolić [Vuk12] and the article by Merideth and Reiter [MR10].

This chapter was written in collaboration with Klaus-Tycho Förster.

Bibliography

[GB85] Hector Garcia-Molina and Daniel Barbará. How to assign votes in a distributed system. *J. ACM*, 32(4):841–860, 1985.

[Gif79] David K. Gifford. Weighted voting for replicated data. In Michael D. Schroeder and Anita K. Jones, editors, *Proceedings of the Seventh Symposium on Operating System Principles, SOSP 1979, Asilomar Conference Grounds, Pacific Grove, California, USA, 10-12, December 1979*, pages 150–162. ACM, 1979.

[Lam78] Leslie Lamport. The implementation of reliable distributed multiprocess systems. *Computer Networks*, 2:95–114, 1978.

[Mae85] Mamoru Maekawa. A square root N algorithm for mutual exclusion in decentralized systems. *ACM Trans. Comput. Syst.*, 3(2):145–159, 1985.

[MR98] Dahlia Malkhi and Michael K. Reiter. Byzantine quorum systems. *Distributed Computing*, 11(4):203–213, 1998.

[MR10] Michael G. Merideth and Michael K. Reiter. Selected results from the latest decade of quorum systems research. In Bernadette Charron-Bost, Fernando Pedone, and André Schiper, editors, *Replication: Theory and Practice*, volume 5959 of *Lecture Notes in Computer Science*, pages 185–206. Springer, 2010.

[MRW97] Dahlia Malkhi, Michael K. Reiter, and Avishai Wool. The load and availability of byzantine quorum systems. In James E. Burns and Hagit Attiya, editors, *Proceedings of the Sixteenth Annual ACM Symposium on Principles of Distributed Computing, Santa Barbara, California, USA, August 21-24, 1997*, pages 249–257. ACM, 1997.

[NW94] Moni Naor and Avishai Wool. The load, capacity and availability of quorum systems. In *35th Annual Symposium on Foundations of Computer Science, Santa Fe, New Mexico, USA, 20-22 November 1994*, pages 214–225. IEEE Computer Society, 1994.

[NW05] Moni Naor and Udi Wieder. Scalable and dynamic quorum systems. *Distributed Computing*, 17(4):311–322, 2005.

[PW95] David Peleg and Avishai Wool. The availability of quorum systems. *Inf. Comput.*, 123(2):210–223, 1995.

[Tho79] Robert H. Thomas. A majority consensus approach to concurrency control for multiple copy databases. *ACM Trans. Database Syst.*, 4(2):180–209, 1979.

[Vuk12] Marko Vukolic. *Quorum Systems: With Applications to Storage and Consensus*. Synthesis Lectures on Distributed Computing Theory. Morgan & Claypool Publishers, 2012.

Chapter 20

Eventual Consistency & Bitcoin

How would you implement an ATM? Does the following implementation work satisfactorily?

Algorithm 20.1 Naïve ATM

```
1: ATM makes withdrawal request to bank
2: ATM waits for response from bank
3: if balance of customer sufficient then
4:     ATM dispenses cash
5: else
6:     ATM displays error
7: end if
```

Remarks:

- A connection problem between the bank and the ATM may block Algorithm 20.1 in Line 2.
- A *network partition* is a failure where a network splits into at least two parts that cannot communicate with each other. Intuitively any non-trivial distributed system cannot proceed during a partition *and* maintain consistency. In the following we introduce the tradeoff between consistency, availability and partition tolerance.
- There are numerous causes for partitions to occur, e.g., physical disconnections, software errors, or incompatible protocol versions. From the point of view of a node in the system, a partition is similar to a period of sustained message loss.

20.1 Consistency, Availability and Partitions

Definition 20.2 (Consistency)**.** *All nodes in the system agree on the current state of the system.*

Definition 20.3 (Availability)**.** *The system is operational and instantly processing incoming requests.*

Definition 20.4 (Partition Tolerance)**.** *Partition tolerance is the ability of a distributed system to continue operating correctly even in the presence of a network partition.*

Theorem 20.5 (CAP Theorem)**.** *It is impossible for a distributed system to simultaneously provide Consistency, Availability and Partition Tolerance. A distributed system can satisfy any two of these but not all three.*

Proof. Assume two nodes, sharing some state. The nodes are in different partitions, i.e., they cannot communicate. Assume a request wants to update the state and contacts a node. The node may either: 1) update its local state, resulting in inconsistent states, or 2) not update its local state, i.e., the system is no longer available for updates. □

Algorithm 20.6 Partition tolerant and available ATM

```
1: if bank reachable then
2:     Synchronize local view of balances between ATM and bank
3:     if balance of customer insufficient then
4:         ATM displays error and aborts user interaction
5:     end if
6: end if
7: ATM dispenses cash
8: ATM logs withdrawal for synchronization
```

Remarks:

- Algorithm 20.6 is partition tolerant and available since it continues to process requests even when the bank is not reachable.
- The ATM's local view of the balances may diverge from the balances as seen by the bank, therefore consistency is no longer guaranteed.
- The algorithm will synchronize any changes it made to the local balances back to the bank once connectivity is re-established. This is known as eventual consistency.

Definition 20.7 (Eventual Consistency)**.** *If no new updates to the shared state are issued, then eventually the system is in a quiescent state, i.e., no more messages need to be exchanged between nodes, and the shared state is consistent.*

Remarks:

- Eventual consistency is a form of *weak consistency*.
- Eventual consistency guarantees that the state is eventually agreed upon, but the nodes may disagree temporarily.
- During a partition, different updates may semantically conflict with each other. A *conflict resolution* mechanism is required to resolve the conflicts and allow the nodes to eventually agree on a common state.

- One example of eventual consistency is the Bitcoin cryptocurrency system.

20.2 Bitcoin

Definition 20.8 (Bitcoin Network)**.** *The Bitcoin network is a randomly connected overlay network of a few thousand* nodes*, controlled by a variety of owners. All nodes perform the same operations, i.e., it is a homogenous network and without central control.*

Remarks:

- The lack of structure is intentional: it ensures that an attacker cannot strategically position itself in the network and manipulate the information exchange. Information is exchanged via a simple broadcasting protocol.

Definition 20.9 (Address)**.** *Users may generate any number of private keys, from which a public key is then derived. An address is derived from a public key and may be used to identify the recipient of funds in Bitcoin. The private/public key pair is used to uniquely identify the owner of funds of an address.*

Remarks:

- The terms public key and address are often used interchangeably, since both are public information. The advantage of using an address is that its representation is shorter than the public key.
- It is hard to link addresses to the user that controls them, hence Bitcoin is often referred to as being *pseudonymous*.
- Not every user needs to run a fully validating node, and end-users will likely use a lightweight client that only temporarily connects to the network.
- The Bitcoin network collaboratively tracks the balance in bitcoins of each address.
- The address is composed of a network identifier byte, the hash of the public key and a checksum. It is commonly stored in base 58 encoding, a custom encoding similar to base 64 with some ambiguous symbols removed, e.g., lowercase letter "l" since it is similar to the number "1".
- The hashing algorithm produces addresses of size 20 bytes. This means that there are 2^{160} distinct addresses. It might be tempting to brute force a target address, however at one billion trials per second one still requires approximately 2^{45} years in expectation to find a matching private/public key pair. Due to the birthday paradox the odds improve if instead of brute forcing a single address we attempt to brute force any address. While the odds of a successful trial increase with the number of addresses, lookups become more costly.

Definition 20.10 (Output)**.** *An output is a tuple consisting of an amount of bitcoins and a spending condition. Most commonly the spending condition requires a valid signature associated with the private key of an address.*

Remarks:

- Spending conditions are scripts that offer a variety of options. Apart from a single signature, they may include conditions that require the result of a simple computation, or the solution to a cryptographic puzzle.
- Outputs exist in two states: unspent and spent. Any output can be spent at most once. The address balance is the sum of bitcoin amounts in unspent outputs that are associated with the address.
- The set of unspent transaction outputs (UTXO) and some additional global parameters is the shared state of Bitcoin. Every node in the Bitcoin network holds a complete replica of that state. Local replicas may temporarily diverge, but consistency is eventually re-established.

Definition 20.11 (Input)**.** *An input is a tuple consisting of a reference to a previously created output and arguments (signature) to the spending condition, proving that the transaction creator has the permission to spend the referenced output.*

Definition 20.12 (Transaction)**.** *A transaction is a datastructure that describes the transfer of bitcoins from spenders to recipients. The transaction consists of a number of inputs and new outputs. The inputs result in the referenced outputs spent (removed from the UTXO), and the new outputs being added to the UTXO.*

Remarks:

- Inputs reference the output that is being spent by a (h, i)-tuple, where h is the hash of the transaction that created the output, and i specifies the index of the output in that transaction.
- Transactions are broadcast in the Bitcoin network and processed by every node that receives them.

Remarks:

- Note that the effect of a transaction on the state is deterministic. In other words if all nodes receive the same set of transactions in the same order (Definition 15.8), then the state across nodes is consistent.
- The outputs of a transaction may assign less than the sum of inputs, in which case the difference is called the transaction's *fee*. The fee is used to incentivize other participants in the system (see Definition 20.19)
- Notice that so far we only described a local acceptance policy. Nothing prevents nodes to locally accept different transactions that spend the same output.

Algorithm 20.13 Node Receives Transaction

```
1: Receive transaction t
2: for each input (h, i) in t do
3:    if output (h, i) is not in local UTXO or signature invalid then
4:       Drop t and stop
5:    end if
6: end for
7: if sum of values of inputs < sum of values of new outputs then
8:    Drop t and stop
9: end if
10: for each input (h, i) in t do
11:    Remove (h, i) from local UTXO
12: end for
13: Append t to local history
14: Forward t to neighbors in the Bitcoin network
```

- Transactions are in one of two states: unconfirmed or confirmed. Incoming transactions from the broadcast are unconfirmed and added to a pool of transactions called the *memory pool*.

Definition 20.14 (Doublespend)**.** *A doublespend is a situation in which multiple transactions attempt to spend the same output. Only one transaction can be valid since outputs can only be spent once. When nodes accept different transactions in a doublespend, the shared state becomes inconsistent.*

Remarks:

- Doublespends may occur naturally, e.g., if outputs are co-owned by multiple users. However, often doublespends are intentional – we call these doublespend-attacks: In a transaction, an attacker pretends to transfer an output to a victim, only to doublespend the same output in another transaction back to itself.
- Doulespends can result in an inconsistent state since the validity of transactions depends on the order in which they arrive. If two conflicting transactions are seen by a node, the node considers the first to be valid, see Algorithm 20.13. The second transaction is invalid since it tries to spend an output that is already spent. The order in which transactions are seen, may not be the same for all nodes, hence the inconsistent state.
- If doublespends are not resolved, the shared state diverges. Therefore a conflict resolution mechanism is needed to decide which of the conflicting transactions is to be confirmed (accepted by everybody), to achieve eventual consistency.

Definition 20.15 (Proof-of-Work)**.** *Proof-of-Work (PoW) is a mechanism that allows a party to prove to another party that a certain amount of computational resources has been utilized for a period of time. A function $\mathcal{F}_d(c,x) \to \{true, false\}$, where difficulty d is a positive number, while challenge c and nonce x are usually bit-strings, is called a Proof-of-Work function if it has following properties:*

1. *$\mathcal{F}_d(c,x)$ is fast to compute if d, c, and x are given.*
2. *For fixed parameters d and c, finding x such that $\mathcal{F}_d(c,x) = true$ is computationally difficult but feasible. The difficulty d is used to adjust the time to find such an x.*

Definition 20.16 (Bitcoin PoW function)**.** *The Bitcoin PoW function is given by*

$$\mathcal{F}_d(c,x) \to \text{SHA256}(\text{SHA256}(c|x)) < \frac{2^{224}}{d}.$$

Remarks:

- This function concatenates the challenge c and nonce x, and hashes them twice using SHA256. The output of SHA256 is a cryptographic hash with a numeric value in $\{0, \ldots, 2^{256}-1\}$ which is compared to a target value $\frac{2^{224}}{d}$, which gets smaller with increasing difficulty.
- SHA256 is a cryptographic hash function with pseudorandom output. No better algorithm is known to find a nonce x such that the function $\mathcal{F}_d(c,x)$ returns true than simply iterating over possible inputs. This is by design to make it difficult to find such an input, but simple to verify the validity once it has been found.
- If the PoW functions of all nodes had the same challenge, the fastest node would always win. However, as we will see in Definition 20.19, each node attempts to find a valid nonce for a node-specific challenge.

Definition 20.17 (Block)**.** *A block is a datastructure used to communicate incremental changes to the local state of a node. A block consists of a list of transactions, a reference to a previous block and a nonce. A block lists some transactions the block creator ("miner") has accepted to its memory-pool since the previous block. A node finds and broadcasts a block when it finds a valid nonce for its PoW function.*

Algorithm 20.18 Node Finds Block

```
1: Nonce x = 0, challenge c, difficulty d, previous block b_{t-1}
2: repeat
3:    x = x + 1
4: until F_d(c, x) = true
5: Broadcast block b_t = (memory-pool, b_{t-1}, x)
```

Remarks:

- With their reference to a previous block, the blocks build a tree, rooted in the so called *genesis block*.
- The primary goal for using the PoW mechanism is to adjust the rate at which blocks are found in the network, giving the network time to synchronize on the latest block. Bitcoin sets the difficulty so that globally a block is created about every 10 minutes in expectation.

- Finding a block allows the finder to impose the transactions in its local memory pool to all other nodes. Upon receiving a block, all nodes roll back any local changes since the previous block and apply the new block's transactions.
- Transactions contained in a block are said to be *confirmed* by that block.

Definition 20.19 (Reward Transaction)**.** *The first transaction in a block is called the reward transaction. The block's miner is rewarded for confirming transactions by allowing it to mint new coins. The reward transaction has a dummy input, and the sum of outputs is determined by a fixed subsidy plus the sum of the fees of transactions confirmed in the block.*

Remarks:

- A reward transaction is the sole exception to the rule that the sum of inputs must be at least the sum of outputs.
- The number of bitcoins that are minted by the reward transaction and assigned to the miner is determined by a subsidy schedule that is part of the protocol. Initially the subsidy was 50 bitcoins for every block, and it is being halved every 210,000 blocks, or 4 years in expectation. Due to the halving of the block reward, the total amount of bitcoins in circulation never exceeds 21 million bitcoins.
- It is expected that the cost of performing the PoW to find a block, in terms of energy and infrastructure, is close to the value of the reward the miner receives from the reward transaction in the block.

Definition 20.20 (Blockchain)**.** *The longest path from the genesis block, i.e., root of the tree, to a leaf is called the blockchain. The blockchain acts as a consistent transaction history on which all nodes eventually agree.*

Remarks:

- The path length from the genesis block to block b is the height h_b.
- Only the longest path from the genesis block to a leaf is a valid transaction history, since branches may contradict each other because of doublespends.
- Since only transactions in the longest path are agreed upon, miners have an incentive to append their blocks to the longest chain, thus agreeing on the current state.
- The mining incentives quickly increased the difficulty of the PoW mechanism: initially miners used CPUs to mine blocks, but CPUs were quickly replaced by GPUs, FPGAs and even application specific integrated circuits (AS-ICs) as bitcoins appreciated. This results in an equilibrium today in which only the most cost efficient miners, in terms of hardware supply and electricity, make a profit in expectation.

- If multiple blocks are mined more or less concurrently, the system is said to have *forked*. Forks happen naturally because mining is a distributed random process and two new blocks may be found at roughly the same time.

Algorithm 20.21 Node Receives Block

```
1: Receive block b
2: For this node the current head is block b_max at height h_max
3: Connect block b in the tree as child of its parent p at height h_b = h_p + 1
4: if h_b > h_max then
5:     h_max = h_b
6:     b_max = b
7:     Compute UTXO for the path leading to b_max
8:     Cleanup memory pool
9: end if
```

Remarks:

- Algorithm 20.21 describes how a node updates its local state upon receiving a block. Notice that, like Algorithm 20.13, this describes the local policy and may also result in node states diverging, i.e., by accepting different blocks at the same height as current head.
- Unlike extending the current path, switching paths may result in confirmed transactions no longer being confirmed, because the blocks in the new path do not include them. Switching paths is referred to as a *reorg*.
- Cleaning up the memory pool involves 1) removing transactions that were confirmed in a block in the current path, 2) removing transactions that conflict with confirmed transactions, and 3) adding transactions that were confirmed in the previous path, but are no longer confirmed in the current path.
- In order to avoid having to recompute the entire UTXO at every new block being added to the blockchain, all current implementations use datastructures that store undo information about the operations applied by a block. This allows efficient switching of paths and updates of the head by moving along the path.

Theorem 20.22. *Forks are eventually resolved and all nodes eventually agree on which is the longest blockchain. The system therefore guarantees eventual consistency.*

Proof. In order for the fork to continue to exist, pairs of blocks need to be found in close succession, extending distinct branches, otherwise the nodes on the shorter branch would switch to the longer one. The probability of branches being extended almost simultaneously decreases exponentially with the length of the fork, hence there will eventually be a time when only one branch is being extended, becoming the longest branch. □

20.3 Smart Contracts

Definition 20.23 (Smart Contract)**.** *A smart contract is an agreement between two or more parties, encoded in such a way that the correct execution is guaranteed by the blockchain.*

Remarks:

- Contracts allow business logic to be encoded in Bitcoin transactions which mutually guarantee that an agreed upon action is performed. The blockchain acts as conflict mediator, should a party fail to honor an agreement.
- The use of scripts as spending conditions for outputs enables smart contracts. Scripts, together with some additional features such as timelocks, allow encoding complex conditions, specifying who may spend the funds associated with an output and when.

Definition 20.24 (Timelock)**.** *Bitcoin provides a mechanism to make transactions invalid until some time in the future:* timelocks*. A transaction may specify a locktime: the earliest time, expressed in either a Unix timestamp or a blockchain height, at which it may be included in a block and therefore be confirmed.*

Remarks:

- Transactions with a timelock are not released into the network until the timelock expires. It is the responsibility of the node receiving the transaction to store it locally until the timelock expires and then release it into the network.
- Transactions with future timelocks are invalid. Blocks may not include transactions with timelocks that have not yet expired, i.e., they are mined before their expiry timestamp or in a lower block than specified. If a block includes an unexpired transaction it is invalid. Upon receiving invalid transactions or blocks, nodes discard them immediately and do not forward them to their peers.
- Timelocks can be used to replace or supersede transactions: a timelocked transaction t_1 can be replaced by another transaction t_0, spending some of the same outputs, if the replacing transaction t_0 has an earlier timelock and can be broadcast in the network before the replaced transaction t_1 becomes valid.

Definition 20.25 (Singlesig and Multisig Outputs)**.** *When an output can be claimed by providing a single signature it is called a* singlesig output*. In contrast the script of* multisig outputs *specifies a set of m public keys and requires k-of-m (with $k \leq m$) valid signatures from distinct matching public keys from that set in order to be valid.*

Remarks:

- Most smart contracts begin with the creation of a 2-of-2 multisig output, requiring a signature from both parties. Once the transaction creating the multisig output is confirmed in the blockchain, both parties are guaranteed that the funds of that output cannot be spent unilaterally.

Algorithm 20.26 Parties A and B create a 2-of-2 multisig output o

1: B sends a list I_B of inputs with c_B coins to A
2: A selects its own inputs I_A with c_A coins
3: A creates transaction $t_s\{[I_A, I_B], [o = c_A + c_B \to (A, B)]\}$
4: A creates timelocked transaction $t_r\{[o], [c_A \to A, c_B \to B]\}$ and signs it
5: A sends t_s and t_r to B
6: B signs both t_s and t_r and sends them to A
7: A signs t_s and broadcasts it to the Bitcoin network

Remarks:

- t_s is called a *setup transaction* and is used to lock in funds into a shared account. If t_s is signed and broadcast immediately, one of the parties could not collaborate to spend the multisig output, and the funds become unspendable. To avoid a situation where the funds cannot be spent, the protocol also creates a timelocked *refund transaction* t_r which guarantees that, should the funds not be spent before the timelock expires, the funds are returned to the respective party. At no point in time one of the parties holds a fully signed setup transaction without the other party holding a fully signed refund transaction, guaranteeing that funds are eventually returned.
- Both transactions require the signature of both parties. In the case of the setup transaction because it has two inputs from A and B respectively which require individual signatures. In the case of the refund transaction the single input spending the multisig output requires both signatures being a 2-of-2 multisig output.

Algorithm 20.27 Simple Micropayment Channel from S to R with capacity c

1: $c_S = c$, $c_R = 0$
2: S and R use Algorithm 20.26 to set up output o with value c from S
3: Create settlement transaction $t_f\{[o], [c_S \to S, c_R \to R]\}$
4: **while** channel open **and** $c_R < c$ **do**
5: In exchange for good with value δ
6: $c_R = c_R + \delta$
7: $c_S = c_S - \delta$
8: Update t_f with outputs $[c_R \to R, c_S \to S]$
9: S signs and sends t_f to R
10: **end while**
11: R signs last t_f and broadcasts it

Remarks:

- Algorithm 20.27 implements a Simple Micropayment Channel, a smart contract that is used for rapidly adjusting micropayments from a spender to a recipient. Only two transactions are ever broadcast and inserted into the blockchain: the setup transaction t_s and the last settlement transaction t_f. There may have been any number of updates to the settlement transaction, transferring ever more of the shared output to the recipient.
- The number of bitcoins c used to fund the channel is also the maximum total that may be transferred over the simple micropayment channel.
- At any time the recipient R is guaranteed to eventually receive the bitcoins, since she holds a fully signed settlement transaction, while the spender only has partially signed ones.
- The simple micropayment channel is intrinsically unidirectional. Since the recipient may choose any of the settlement transactions in the protocol, she will use the one with maximum payout for her. If we were to transfer bitcoins back, we would be reducing the amount paid out to the recipient, hence she would choose not to broadcast that transaction.

20.4 Weak Consistency

Eventual consistency is only one form of weak consistency. A number of different tradeoffs between partition tolerance and consistency exist in literature.

Definition 20.28 (Monotonic Read Consistency)**.** *If a node u has seen a particular value of an object, any subsequent accesses of u will never return any older values.*

Remarks:

- Users are annoyed if they receive a notification about a comment on an online social network, but are unable to reply because the web interface does not show the same notification yet. In this case the notification acts as the first read operation, while looking up the comment on the web interface is the second read operation.

Definition 20.29 (Monotonic Write Consistency)**.** *A write operation by a node on a data item is completed before any successive write operation by the same node (i.e. system guarantees to serialize writes by the same node).*

Remarks:

- The ATM must replay all operations in order, otherwise it might happen that an earlier operation overwrites the result of a later operation, resulting in an inconsistent final state.

Definition 20.30 (Read-Your-Write Consistency)**.** *After a node u has updated a data item, any later reads from node u will never see an older value.*

Definition 20.31 (Causal Relation)**.** *The following pairs of operations are said to be causally related:*

- *Two writes by the same node to different variables.*
- *A read followed by a write of the same node.*
- *A read that returns the value of a write from any node.*
- *Two operations that are transitively related according to the above conditions.*

Remarks:

- The first rule ensures that writes by a single node are seen in the same order. For example if a node writes a value in one variable and then signals that it has written the value by writing in another variable. Another node could then read the signalling variable but still read the old value from the first variable, if the two writes were not causally related.

Definition 20.32 (Causal Consistency)**.** *A system provides causal consistency if operations that potentially are causally related are seen by every node of the system in the same order. Concurrent writes are not causally related, and may be seen in different orders by different nodes.*

Chapter Notes

The CAP theorem was first introduced by Fox and Brewer [FB99], although it is commonly attributed to a talk by Eric Brewer [Bre00]. It was later proven by Gilbert and Lynch [GL02] for the asynchronous model. Gilbert and Lynch also showed how to relax the consistency requirement in a partially synchronous system to achieve availability and partition tolerance.

Bitcoin was introduced in 2008 by Satoshi Nakamoto [Nak08]. Nakamoto is thought to be a pseudonym used by either a single person or a group of people; it is still unknown who invented Bitcoin, giving rise to speculation and conspiracy theories. Among the plausible theories are noted cryptographers Nick Szabo [Big13] and Hal Finney [Gre14]. The first Bitcoin client was published shortly after the paper and the first block was mined on January 3, 2009. The genesis block contained the headline of the release date's The Times issue "*The Times 03/Jan/2009 Chancellor on brink of second bailout for banks*", which serves as proof that the genesis block has been indeed mined on that date, and that no one had mined before that date. The quote in the genesis block is also thought to be an ideological hint: Bitcoin was created in a climate of financial crisis, induced by rampant manipulation by the banking sector, and Bitcoin quickly grew in popularity in anarchic and libertarian circles. The original client is nowadays maintained by a group of independent core developers and remains the most used client in the Bitcoin network.

Central to Bitcoin is the resolution of conflicts due to doublespends, which is solved by waiting for transactions to be included in the blockchain. This however introduces large delays for the confirmation of payments which are

undesirable in some scenarios in which an immediate confirmation is required. Karame et al. [KAC12] show that accepting unconfirmed transactions leads to a non-negligible probability of being defrauded as a result of a doublespending attack. This is facilitated by *information eclipsing* [DW13], i.e., that nodes do not forward conflicting transactions, hence the victim does not see both transactions of the doublespend. Bamert et al. [BDE+13] showed that the odds of detecting a doublespending attack in real-time can be improved by connecting to a large sample of nodes and tracing the propagation of transactions in the network.

Bitcoin does not scale very well due to its reliance on confirmations in the blockchain. A copy of the entire transaction history is stored on every node in order to bootstrap joining nodes, which have to reconstruct the transaction history from the genesis block. Simple micropayment channels were introduced by Hearn and Spilman [HS12] and may be used to bundle multiple transfers between two parties but they are limited to transferring the funds locked into the channel once. Recently Duplex Micropayment Channels [DW15] and the Lightning Network [PD15] have been proposed to build bidirectional micropayment channels in which the funds can be transferred back and forth an arbitrary number of times, greatly increasing the flexibility of Bitcoin transfers and enabling a number of features, such as micropayments and routing payments between any two endpoints.

This chapter was written in collaboration with Christian Decker.

Bibliography

[BDE+13] Tobias Bamert, Christian Decker, Lennart Elsen, Samuel Welten, and Roger Wattenhofer. Have a snack, pay with bitcoin. In *IEEE Internation Conference on Peer-to-Peer Computing (P2P), Trento, Italy*, 2013.

[Big13] John Biggs. Who is the real satoshi nakamoto? one researcher may have found the answer. http://on.tcrn.ch/l/R0vA, 2013.

[Bre00] Eric A. Brewer. Towards robust distributed systems. In *Symposium on Principles of Distributed Computing (PODC)*. ACM, 2000.

[DW13] Christian Decker and Roger Wattenhofer. Information propagation in the bitcoin network. In *IEEE International Conference on Peer-to-Peer Computing (P2P), Trento, Italy*, September 2013.

[DW15] Christian Decker and Roger Wattenhofer. A Fast and Scalable Payment Network with Bitcoin Duplex Micropayment Channels. In *Symposium on Stabilization, Safety, and Security of Distributed Systems (SSS)*, 2015.

[FB99] Armando Fox and Eric Brewer. Harvest, yield, and scalable tolerant systems. In *Hot Topics in Operating Systems*. IEEE, 1999.

[GL02] Seth Gilbert and Nancy Lynch. Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services. *SIGACT News*, 2002.

[Gre14] Andy Greenberg. Nakamoto's neighbor: My hunt for bitcoin's creator led to a paralyzed crypto genius. http://onforb.es/1rvyecq, 2014.

[HS12] Mike Hearn and Jeremy Spilman. Contract: Rapidly adjusting micro-payments. https://en.bitcoin.it/wiki/Contract, 2012. Last accessed on November 11, 2015.

[KAC12] G.O. Karame, E. Androulaki, and S. Capkun. Two Bitcoins at the Price of One? Double-Spending Attacks on Fast Payments in Bitcoin. In *Conference on Computer and Communication Security*, 2012.

[Nak08] Satoshi Nakamoto. Bitcoin: A peer-to-peer electronic cash system. https://bitcoin.org/bitcoin.pdf, 2008.

[PD15] Joseph Poon and Thaddeus Dryja. The bitcoin lightning network. 2015.

Chapter 21

Distributed Storage

How do you store 1M movies, each with a size of about 1GB, on 1M nodes, each equipped with a 1TB disk? Simply store the movies on the nodes, arbitrarily, and memorize (with a global index) which movie is stored on which node. What if the set of movies or nodes changes over time, and you do not want to change your global index too often?

21.1 Consistent Hashing

Several variants of hashing will do the job, e.g. consistent hashing:

Algorithm 21.1 Consistent Hashing

1: Hash the unique file name of each movie m with a known set of hash functions $h_i(m) \to [0,1)$, for $i = 1, \ldots, k$
2: Hash the unique name (e.g., IP address and port number) of each node with the same set of hash functions h_i, for $i = 1, \ldots, k$
3: Store a copy of a movie x on node u if $h_i(x) \approx h_i(u)$, for any i. More formally, store movie x on node u if

$$|h_i(x) - h_i(u)| = \min_m\{|h_i(m) - h_i(u)|\}, \text{ for any } i$$

Theorem 21.2 (Consistent Hashing)**.** *In expectation, Algorithm 21.1 stores each movie kn/m times.*

Proof. While it is possible that some movie does not hash closest to a node for any of its hash functions, this is highly unlikely: For each node (n) and each hash function (k), each movie has about the same probability ($1/m$) to be stored. By linearity of expectation, a movie is stored kn/m times, in expectation. □

Remarks:

- Let us do a back-of-the-envelope calculation. We have $m = 1$M movies, $n = 1$M nodes, each node has storage for 1TB/1GB = 1K movies, i.e., we use $k = 1$K hash functions. Theorem 21.2 shows that each movie is stored about 1K times. With a bit more math one can show that it is highly unlikely that a movie is stored much less often than its expected value.
- Instead of storing movies directly on nodes as in Algorithm 21.1, we can also store the movies on any nodes we like. The nodes of Algorithm 21.1 then simply store forward pointers to the actual movie locations.
- In this chapter we want to push unreliability to the extreme. What if the nodes are so unreliable that on average a node is only available for 1 hour? In other words, nodes exhibit a high *churn*, they constantly join and leave the distributed system.
- With such a high churn, hundreds or thousands of nodes will change every second. No single node can have an accurate picture of what other nodes are currently in the system. This is remarkably different to classic distributed systems, where a single unavailable node may already be a minor disaster: all the other nodes have to get a consistent view (Definition 18.4) of the system again. In high churn systems it is impossible to have a consistent view at any time.
- Instead, each node will just know about a small subset of 100 or less other nodes ("neighbors"). This way, nodes can withstand high churn situations.
- On the downside, nodes will not directly know which node is responsible for what movie. Instead, a node searching for a movie might have to ask a neighbor node, which in turn will recursively ask another neighbor node, until the correct node storing the movie (or a forward pointer to the movie) is found. The nodes of our distributed storage system form a virtual network, also called an *overlay network*.

21.2 Hypercubic Networks

In this section we present a few overlay topologies of general interest.

Definition 21.3 (Topology Properties)**.** *Our virtual network should have the following properties:*

- *The network should be (somewhat)* homogeneous*: no node should play a dominant role, no node should be a single point of failure.*
- *The nodes should have* IDs*, and the IDs should span the universe $[0,1)$, such that we can store data with hashing, as in Algorithm 21.1.*
- *Every node should have a small* degree*, if possible polylogarithmic in n, the number of nodes. This will allow every node to maintain a persistent connection with each neighbor, which will help us to deal with churn.*

- *The network should have a small* diameter*, and routing should be easy. If a node does not have the information about a data item, then it should know which neighbor to ask. Within a few (polylogarithmic in n) hops, one should find the node that has the correct information.*

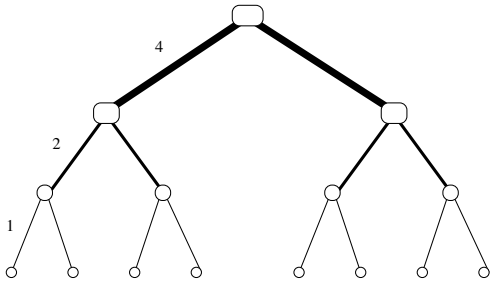


Figure 21.4: The structure of a fat tree.

Remarks:

- Some basic network topologies used in practice are trees, rings, grids or tori. Many other suggested networks are simply combinations or derivatives of these.
- The advantage of trees is that the routing is very easy: for every source-destination pair there is only one path. However, since the root of a tree is a bottleneck, trees are not homogeneous. Instead, so-called *fat trees* should be used. Fat trees have the property that every edge connecting a node v to its parent u has a capacity that is proportional to the number of leaves of the subtree rooted at v. See Figure 21.4 for a picture.
- Fat trees belong to a family of networks that require edges of non-uniform capacity to be efficient. Networks with edges of uniform capacity are easier to build. This is usually the case for grids and tori. Unless explicitly mentioned, we will treat all edges in the following to be of capacity 1.

Definition 21.5 (Torus, Mesh)**.** *Let $m, d \in \mathbb{N}$. The (m,d)-*mesh *$M(m,d)$ is a graph with node set $V = [m]^d$ and edge set*

$$E = \left\{ \{(a_1,\ldots,a_d),(b_1,\ldots,b_d)\} \mid a_i, b_i \in [m], \sum_{i=1}^{d} |a_i - b_i| = 1 \right\},$$

*where $[m]$ means the set $\{0,\ldots,m-1\}$. The (m,d)-*torus *$T(m,d)$ is a graph that consists of an (m,d)-mesh and additionally wrap-around edges from nodes $(a_1,\ldots,a_{i-1},m-1,a_{i+1},\ldots,a_d)$ to nodes $(a_1,\ldots,a_{i-1},0,a_{i+1},\ldots,a_d)$ for all*

$i \in \{1,\ldots,d\}$ and all $a_j \in [m]$ with $j \neq i$. In other words, we take the expression $a_i - b_i$ in the sum modulo m prior to computing the absolute value. $M(m,1)$ is also called a path*, $T(m,1)$ a* cycle*, and $M(2,d) = T(2,d)$ a* d-dimensional hypercube*. Figure 21.6 presents a linear array, a torus, and a hypercube.*

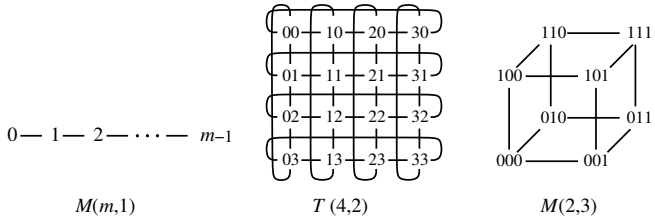


Figure 21.6: The structure of $M(m,1)$, $T(4,2)$, and $M(2,3)$.

Remarks:

- Routing on a mesh, torus, or hypercube is trivial. On a d-dimensional hypercube, to get from a source bitstring s to a target bitstring t one only needs to fix each "wrong" bit, one at a time; in other words, if the source and the target differ by k bits, there are $k!$ routes with k hops.
- If you put a dot in front of the d-bit ID of each node, the nodes exactly span the d-bit IDs $[0,1)$.
- The Chord architecture is a close relative of the hypercube, basically a less rigid hypercube. The hypercube connects every node with an ID in $[0,1)$ with every node in *exactly* distance 2^{-i}, $i = 1,2,\ldots,d$ in $[0,1)$. Chord instead connect nodes with *approximately* distance 2^{-i}.
- The hypercube has many derivatives, the so-called *hypercubic networks*. Among these are the butterfly, cube-connected-cycles, shuffle-exchange, and de Bruijn graph. We start with the butterfly, which is basically a "rolled out" hypercube.

Definition 21.7 (Butterfly)**.** *Let $d \in \mathbb{N}$. The d-*dimensional butterfly *$BF(d)$ is a graph with node set $V = [d+1] \times [2]^d$ and an edge set $E = E_1 \cup E_2$ with*

$$E_1 = \{\{(i,\alpha),(i+1,\alpha)\} \mid i \in [d],\ \alpha \in [2^d]\}$$

and

$$E_2 \quad = \quad \{\{(i,\alpha),(i+1,\beta)\} \mid i \in [d],\ \alpha,\beta \in [2^d], |\alpha - \beta| = 2^i\}.$$

A node set $\{(i,\alpha) \mid \alpha \in [2]^d\}$ is said to form level *i of the butterfly. The d-*dimensional wrap-around butterfly *W-$BF(d)$ is defined by taking the $BF(d)$ and having $(d,\alpha) = (0,\alpha)$ for all $\alpha \in [2]^d$.*

Remarks:

- Figure 21.8 shows the 3-dimensional butterfly $BF(3)$. The $BF(d)$ has $(d+1)2^d$ nodes, $2d \cdot 2^d$ edges and degree 4. It is not difficult to check that combining the node sets $\{(i,\alpha) \mid i \in [d]\}$ for all $\alpha \in [2]^d$ into a single node results in the hypercube.
- Butterflies have the advantage of a constant node degree over hypercubes, whereas hypercubes feature more fault-tolerant routing.
- You may have seen butterfly-like structures before, e.g. sorting networks, communication switches, data center networks, fast fourier transform (FFT). The Benes network (telecommunication) is nothing but two back-to-back butterflies. The Clos network (data centers) is a close relative to Butterflies too. Actually, merging the 2^i nodes on level i that share the first $d-i$ bits into a single node, the Butterfly becomes a fat tree. Every year there are new applications for which hypercubic networks are the perfect solution!
- Next we define the cube-connected-cycles network. It only has a degree of 3 and it results from the hypercube by replacing the corners by cycles.

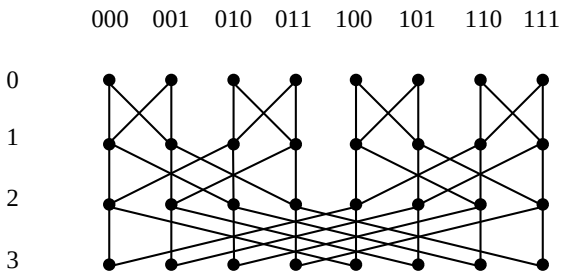


Figure 21.8: The structure of BF(3).

Definition 21.9 (Cube-Connected-Cycles)**.** *Let $d \in \mathbb{N}$. The* cube-connected-cycles *network CCC(d) is a graph with node set $V = \{(a,p) \mid a \in [2]^d, p \in [d]\}$ and edge set*

$$E = \big\{\{(a,p),(a,(p+1) \bmod d)\} \mid a \in [2]^d, p \in [d]\big\} \cup \big\{\{(a,p),(b,p)\} \mid a,b \in [2]^d, p \in [d], a = b \text{ except for } a_p\big\} \ .$$

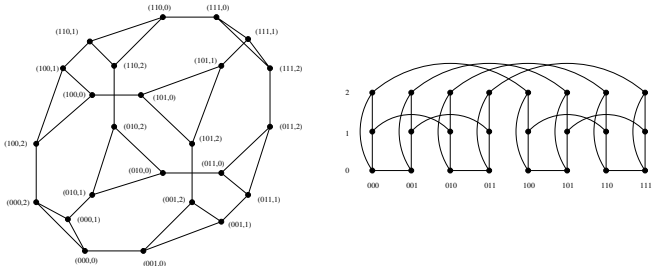


Figure 21.10: The structure of CCC(3).

Remarks:

- Two possible representations of a CCC can be found in Figure 21.10.
- The shuffle-exchange is yet another way of transforming the hypercubic interconnection structure into a constant degree network.

Definition 21.11 (Shuffle-Exchange)**.** *Let $d \in \mathbb{N}$. The d-*dimensional shuffle-exchange *$SE(d)$ is defined as an undirected graph with node set $V = [2]^d$ and an edge set $E = E_1 \cup E_2$ with*

$$E_1 = \{\{(a_1,\ldots,a_d),(a_1,\ldots,\bar{a}_d)\} \mid (a_1,\ldots,a_d) \in [2]^d,\ \bar{a}_d = 1 - a_d\}$$

and

$$E_2 = \{\{(a_1,\ldots,a_d),(a_d,a_1,\ldots,a_{d-1})\} \mid (a_1,\ldots,a_d) \in [2]^d\} \ .$$

Figure 21.12 shows the 3- and 4-dimensional shuffle-exchange graph.

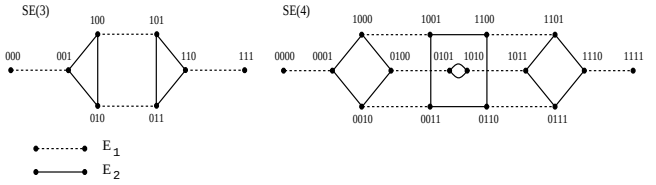


Figure 21.12: The structure of SE(3) and SE(4).

Definition 21.13 (DeBruijn)**.** *The b-*ary DeBruijn graph of dimension d *$DB(b,d)$ is an undirected graph $G = (V,E)$ with node set $V = \{v \in [b]^d\}$ and edge set E that contains all edges $\{v,w\}$ with the property that $w \in \{(x,v_1,\ldots,v_{d-1}) : x \in [b]\}$, where $v = (v_1,\ldots,v_d)$.*

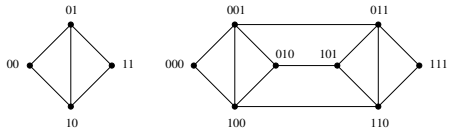


Figure 21.14: The structure of $DB(2,2)$ and $DB(2,3)$.

Remarks:

- Two examples of a DeBruijn graph can be found in Figure 21.14.
- There are some data structures which also qualify as hypercubic networks. An example of a hypercubic network is the skip list, the balanced binary search tree for the lazy programmer:

Definition 21.15 (Skip List)**.** *The skip list is an ordinary ordered linked list of objects, augmented with additional forward links. The ordinary linked list is the level 0 of the skip list. In addition, every object is promoted to level 1 with probability 1/2. As for level 0, all level 1 objects are connected by a linked list. In general, every object on level i is promoted to the next level with probability* $1/2$*. A special start-object points to the smallest/first object on each level.*

Remarks:

- Search, insert, and delete can be implemented in $\mathcal{O}(\log n)$ expected time in a skip list, simply by jumping from higher levels to lower ones when overshooting the searched position. Also, the amortized memory cost of each object is constant, as on average an object only has two forward links.
- The randomization can easily be discarded, by deterministically promoting a constant fraction of objects of level i to level $i+1$, for all i. When inserting or deleting, object o simply checks whether its left and right level i neighbors are being promoted to level $i+1$. If none of them is, promote object o itself. Essentially we establish a maximal independent set (MIS) on each level, hence at least every third and at most every second object is promoted.
- There are obvious variants of the skip list, e.g., the skip graph. Instead of promoting only half of the nodes to the next level, we always promote all the nodes, similarly to a balanced binary tree: All nodes are part of the root level of the binary tree. Half the nodes are promoted left, and half the nodes are promoted right, on each level. Hence on level i we have have 2^i lists (or, if we connect the last element again with the first: rings) of about $n/2^i$ objects. The skip graph features all the properties of Definition 21.3.
- More generally, how are degree and diameter of Definition 21.3 related? The following theorem gives a general lower bound.

Theorem 21.16. *Every graph of maximum degree $d > 2$ and size n must have a diameter of at least $\lceil (\log n)/(\log(d-1)) \rceil - 2$.*

Proof. Suppose we have a graph $G = (V, E)$ of maximum degree d and size n. Start from any node $v \in V$. In a first step at most d other nodes can be reached. In two steps at most $d \cdot (d-1)$ additional nodes can be reached. Thus, in general, in at most k steps at most

$$1 + \sum_{i=0}^{k-1} d \cdot (d-1)^i = 1 + d \cdot \frac{(d-1)^k - 1}{(d-1) - 1} \leq \frac{d \cdot (d-1)^k}{d-2}$$

nodes (including v) can be reached. This has to be at least n to ensure that v can reach all other nodes in V within k steps. Hence,

$$(d-1)^k \geq \frac{(d-2) \cdot n}{d} \quad \Leftrightarrow \quad k \geq \log_{d-1}((d-2) \cdot n/d) \; .$$

Since $\log_{d-1}((d-2)/d) > -2$ for all $d > 2$, this is true only if $k \geq \lceil (\log n)/(\log(d-1)) \rceil - 2$. □

Remarks:

- In other words, constant-degree hypercubic networks feature an asymptotically optimal diameter.
- Other hypercubic graphs manage to have a different tradeoff between node degree and diameter. The pancake graph, for instance, minimizes the maximum of these with $d = k = \Theta(\log n / \log \log n)$. The ID of a node u in the pancake graph of dimension d is an arbitrary permutation of the numbers $1, 2, \ldots, d$. Two nodes u, v are connected by an edge if one can get the ID of node v by taking the ID of node u, and reversing (flipping) the first i numbers of u's ID. For example, in dimension $d = 4$, nodes $u = 2314$ and $v = 1324$ are neighbors.
- There are a few other interesting graph classes which are not hypercubic networks, but nevertheless seem to relate to the properties of Definition 21.3. Small-world graphs (a popular representations for social networks) also have small diameter, however, in contrast to hypercubic networks, they are not homogeneous and feature nodes with large degrees.
- Expander graphs (an expander graph is a sparse graph which has good connectivity properties, that is, from every not too large subset of nodes you are connected to an even larger set of nodes) are homogeneous, have a low degree and small diameter. However, expanders are often not routable.

21.3 DHT & Churn

Definition 21.17 (Distributed Hash Table (DHT))**.** *A* distributed hash table *(DHT) is a distributed data structure that implements a distributed storage. A DHT should support at least (i) a search (for a key) and (ii) an insert (key, object) operation, possibly also (iii) a delete (key) operation.*

Remarks:

- A DHT has many applications beyond storing movies, e.g., the Internet domain name system (DNS) is essentially a DHT.
- A DHT can be implemented as a hypercubic overlay network with nodes having identifiers such that they span the ID space $[0, 1)$.
- A hypercube can directly be used for a DHT. Just use a globally known set of hash functions h_i, mapping movies to bit strings with d bits.
- Other hypercubic structures may be a bit more intricate when using it as a DHT: The butterfly network, for instance, may directly use the $d+1$ layers for replication, i.e., all the $d+1$ nodes are responsible for the same ID.
- Other hypercubic networks, e.g. the pancake graph, might need a bit of twisting to find appropriate IDs.
- We assume that a joining node knows a node which already belongs to the system. This is known as the bootstrap problem. Typical solutions are: If a node has been connected with the DHT previously, just try some of these previous nodes. Or the node may ask some authority for a list of IP addresses (and ports) of nodes that are regularly part of the DHT.
- Many DHTs in the literature are analyzed against an adversary that can crash a fraction of random nodes. After crashing a few nodes the system is given sufficient time to recover again. However, this seems unrealistic. The scheme sketched in this section significantly differs from this in two major aspects.
- First, we assume that joins and leaves occur in a worst-case manner. We think of an adversary that can remove and add a bounded number of nodes; the adversary can choose which nodes to crash and how nodes join.
- Second, the adversary does not have to wait until the system is recovered before it crashes the next batch of nodes. Instead, the adversary can constantly crash nodes, while the system is trying to stay alive. Indeed, the system is *never fully repaired* but *always fully functional*. In particular, the system is resilient against an adversary that continuously attacks the "weakest part" of the system. The adversary could for example insert a crawler into the DHT, learn the topology of the system, and then repeatedly crash selected nodes, in an attempt to partition the DHT. The system counters such an adversary by continuously moving the remaining or newly joining nodes towards the areas under attack.
- Clearly, we cannot allow the adversary to have unbounded capabilities. In particular, in any constant time interval, the adversary can at most add and/or remove $O(\log n)$ nodes, n being the total number of nodes currently in the system. This model covers an adversary which repeatedly takes down nodes by a distributed denial of service attack, however only a logarithmic number of nodes at each point in time. The algorithm relies on messages being delivered timely, in at most constant time between any pair of operational nodes, i.e., the synchronous model. Using the trivial synchronizer this is not a problem. We only need bounded message delays in order to have a notion of time which is needed for the adversarial model. The duration of a round is then proportional to the propagation delay of the slowest message.

Algorithm 21.18 DHT

1: Given: a globally known set of hash functions h_i, and a hypercube (or any other hypercubic network)
2: Each hypercube virtual node ("hypernode") consists of $\Theta(\log n)$ nodes.
3: Nodes have connections to all other nodes of their hypernode and to nodes of their neighboring hypernodes.
4: Because of churn, some of the nodes have to change to another hypernode such that up to constant factors, all hypernodes own the same number of nodes at all times.
5: If the total number of nodes n grows or shrinks above or below a certain threshold, the dimension of the hypercube is increased or decreased by one, respectively.

Remarks:

- Having a logarithmic number of hypercube neighbors, each with a logarithmic number of nodes, means that each node has $\Theta(\log^2 n)$ neighbors. However, with some additional bells and whistles one can achieve $\Theta(\log n)$ neighbor nodes.
- The balancing of nodes among the hypernodes can be seen as a dynamic token distribution problem on the hypercube. Each hypernode has a certain number of tokens, the goal is to distribute the tokens along the edges of the graph such that all hypernodes end up with the same or almost the same number of tokens. While tokens are moved around, an adversary constantly inserts and deletes tokens. See also Figure 21.19.
- In summary, the storage system builds on two basic components: (i) an algorithm which performs the described dynamic token distribution and (ii) an information aggregation algorithm which is used to estimate the number of nodes in the system and to adapt the dimension of the hypercube accordingly:

Theorem 21.20 (DHT with Churn)**.** *We have a fully scalable, efficient distributed storage system which tolerates* $\mathrm{O}(\log n)$ *worst-case joins and/or crashes per constant time interval. As in other storage systems, nodes have* $\mathrm{O}(\log n)$ *overlay neighbors, and the usual operations (e.g., search, insert) take time* $\mathrm{O}(\log n)$.

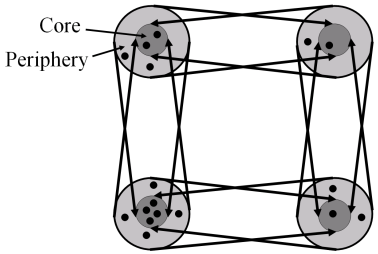


Figure 21.19: A simulated 2-dimensional hypercube with four hypernodes, each consisting of several nodes. Also, all the nodes are either in the core or in the periphery of a node. All nodes within the same hypernode are completely connected to each other, and additionally, all nodes of a hypernode are connected to the core nodes of the neighboring nodes. Only the core nodes store data items, while the peripheral nodes move between the nodes to balance biased adversarial churn.

Remarks:

- Indeed, handling churn is only a minimal requirement to make a distributed storage system work. Advanced studies proposed more elaborate architectures which can also handle other security issues, e.g., privacy or Byzantine attacks.

Chapter Notes

The ideas behind distributed storage were laid during the peer-to-peer (P2P) file sharing hype around the year 2000, so a lot of the seminal research in this area is labeled P2P. The paper of Plaxton, Rajaraman, and Richa [PRR97] laid out a blueprint for many so-called structured P2P architecture proposals, such as Chord [SMK+01], CAN [RFH+01], Pastry [RD01], Viceroy [MNR02], Kademlia [MM02], Koorde [KK03], SkipGraph [AS03], SkipNet [HJS+03], or Tapestry [ZHS+04]. Also the paper of Plaxton et. al. was standing on the shoulders of giants. Some of its eminent precursors are: linear and consistent hashing [KLL+97], locating shared objects [AP90, AP91], compact routing [SK85, PU88], and even earlier: hypercubic networks, e.g. [AJ75, Wit81, GS81, BA84].

Furthermore, the techniques in use for prefix-based overlay structures are related to a proposal called LAND, a locality-aware distributed hash table proposed by Abraham et al. [AMD04].

More recently, a lot of P2P research focussed on security aspects, describing for instance attacks [LMSW06, SENB07, Lar07], and provable countermeasures [KSW05, AS09, BSS09]. Another topic currently garnering interest is using P2P to help distribute live streams of video content on a large scale [LMSW07]. There are several recommendable introductory books on P2P computing, e.g. [SW05, SG05, MS07, KW08, BYL08].

Some of the figures in this chapter have been provided by Christian Scheideler.

Bibliography

[AJ75] George A. Anderson and E. Douglas Jensen. Computer Interconnection Structures: Taxonomy, Characteristics, and Examples. *ACM Comput. Surv.*, 7(4):197–213, December 1975.

[AMD04] Ittai Abraham, Dahlia Malkhi, and Oren Dobzinski. LAND: stretch (1 + epsilon) locality-aware networks for DHTs. In *Proceedings of the fifteenth annual ACM-SIAM symposium on Discrete algorithms*, SODA '04, pages 550–559, Philadelphia, PA, USA, 2004. Society for Industrial and Applied Mathematics.

[AP90] Baruch Awerbuch and David Peleg. Efficient Distributed Construction of Sparse Covers. Technical report, The Weizmann Institute of Science, 1990.

[AP91] Baruch Awerbuch and David Peleg. Concurrent Online Tracking of Mobile Users. In *SIGCOMM*, pages 221–233, 1991.

[AS03] James Aspnes and Gauri Shah. Skip Graphs. In *SODA*, pages 384–393. ACM/SIAM, 2003.

[AS09] Baruch Awerbuch and Christian Scheideler. Towards a Scalable and Robust DHT. *Theory Comput. Syst.*, 45(2):234–260, 2009.

[BA84] L. N. Bhuyan and D. P. Agrawal. Generalized Hypercube and Hyperbus Structures for a Computer Network. *IEEE Trans. Comput.*, 33(4):323–333, April 1984.

[BSS09] Matthias Baumgart, Christian Scheideler, and Stefan Schmid. A DoS-resilient information system for dynamic data management. In *Proceedings of the twenty-first annual symposium on Parallelism in algorithms and architectures*, SPAA '09, pages 300–309, New York, NY, USA, 2009. ACM.

[BYL08] John Buford, Heather Yu, and Eng Keong Lua. *P2P Networking and Applications*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2008.

[GS81] J.R. Goodman and C.H. Sequin. Hypertree: A Multiprocessor Interconnection Topology. *Computers, IEEE Transactions on*, C-30(12):923–933, dec. 1981.

[HJS+03] Nicholas J. A. Harvey, Michael B. Jones, Stefan Saroiu, Marvin Theimer, and Alec Wolman. SkipNet: a scalable overlay network with practical locality properties. In *Proceedings of the 4th conference on USENIX Symposium on Internet Technologies and Systems - Volume 4*, USITS'03, pages 9–9, Berkeley, CA, USA, 2003. USENIX Association.

[KK03] M. Frans Kaashoek and David R. Karger. Koorde: A Simple Degree-Optimal Distributed Hash Table. In M. Frans Kaashoek and Ion Stoica, editors, *IPTPS*, volume 2735 of *Lecture Notes in Computer Science*, pages 98–107. Springer, 2003.

[KLL+97] David R. Karger, Eric Lehman, Frank Thomson Leighton, Rina Panigrahy, Matthew S. Levine, and Daniel Lewin. Consistent Hashing and Random Trees: Distributed Caching Protocols for Relieving Hot Spots on the World Wide Web. In Frank Thomson Leighton and Peter W. Shor, editors, *STOC*, pages 654–663. ACM, 1997.

[KSW05] Fabian Kuhn, Stefan Schmid, and Roger Wattenhofer. A Self-Repairing Peer-to-Peer System Resilient to Dynamic Adversarial Churn. In *4th International Workshop on Peer-To-Peer Systems (IPTPS), Cornell University, Ithaca, New York, USA, Springer LNCS 3640*, February 2005.

[KW08] Javed I. Khan and Adam Wierzbicki. Introduction: Guest editors' introduction: Foundation of peer-to-peer computing. *Comput. Commun.*, 31(2):187–189, February 2008.

[Lar07] Erik Larkin. Storm Worm's virulence may change tactics. http://www.networkworld.com/news/2007/080207-black-hat-storm-worms-virulence.html, Agust 2007. Last accessed on June 11, 2012.

[LMSW06] Thomas Locher, Patrick Moor, Stefan Schmid, and Roger Wattenhofer. Free Riding in BitTorrent is Cheap. In *5th Workshop on Hot Topics in Networks (HotNets), Irvine, California, USA*, November 2006.

[LMSW07] Thomas Locher, Remo Meier, Stefan Schmid, and Roger Wattenhofer. Push-to-Pull Peer-to-Peer Live Streaming. In *21st International Symposium on Distributed Computing (DISC), Lemesos, Cyprus*, September 2007.

[MM02] Petar Maymounkov and David Mazières. Kademlia: A Peer-to-Peer Information System Based on the XOR Metric. In *Revised Papers from the First International Workshop on Peer-to-Peer Systems*, IPTPS '01, pages 53–65, London, UK, UK, 2002. Springer-Verlag.

[MNR02] Dahlia Malkhi, Moni Naor, and David Ratajczak. Viceroy: a scalable and dynamic emulation of the butterfly. In *Proceedings of the twenty-first annual symposium on Principles of distributed computing*, PODC '02, pages 183–192, New York, NY, USA, 2002. ACM.

[MS07] Peter Mahlmann and Christian Schindelhauer. *Peer-to-Peer Networks*. Springer, 2007.

[PRR97] C. Greg Plaxton, Rajmohan Rajaraman, and Andréa W. Richa. Accessing Nearby Copies of Replicated Objects in a Distributed Environment. In *SPAA*, pages 311–320, 1997.

[PU88] David Peleg and Eli Upfal. A tradeoff between space and efficiency for routing tables. In *Proceedings of the twentieth annual ACM symposium on Theory of computing*, STOC '88, pages 43–52, New York, NY, USA, 1988. ACM.

[RD01] Antony Rowstron and Peter Druschel. Pastry: Scalable, decentralized object location and routing for large-scale peer-to-peer systems. In *IFIP/ACM International Conference on Distributed Systems Platforms (Middleware)*, pages 329–350, November 2001.

[RFH+01] Sylvia Ratnasamy, Paul Francis, Mark Handley, Richard Karp, and Scott Shenker. A scalable content-addressable network. *SIGCOMM Comput. Commun. Rev.*, 31(4):161–172, August 2001.

[SENB07] Moritz Steiner, Taoufik En-Najjary, and Ernst W. Biersack. Exploiting KAD: possible uses and misuses. *SIGCOMM Comput. Commun. Rev.*, 37(5):65–70, October 2007.

[SG05] Ramesh Subramanian and Brian D. Goodman. *Peer to Peer Computing: The Evolution of a Disruptive Technology*. IGI Publishing, Hershey, PA, USA, 2005.

[SK85] Nicola Santoro and Ramez Khatib. Labelling and Implicit Routing in Networks. *Comput. J.*, 28(1):5–8, 1985.

[SMK+01] Ion Stoica, Robert Morris, David Karger, M. Frans Kaashoek, and Hari Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. *SIGCOMM Comput. Commun. Rev.*, 31(4):149–160, August 2001.

[SW05] Ralf Steinmetz and Klaus Wehrle, editors. *Peer-to-Peer Systems and Applications*, volume 3485 of *Lecture Notes in Computer Science*. Springer, 2005.

[Wit81] L. D. Wittie. Communication Structures for Large Networks of Microcomputers. *IEEE Trans. Comput.*, 30(4):264–273, April 1981.

[ZHS+04] Ben Y. Zhao, Ling Huang, Jeremy Stribling, Sean C. Rhea, Anthony D. Joseph, and John Kubiatowicz. Tapestry: a resilient global-scale overlay for service deployment. *IEEE Journal on Selected Areas in Communications*, 22(1):41–53, 2004.

Chapter 22

Game Theory

"Game theory is a sort of umbrella or 'unified field' theory for the rational side of social science, where 'social' is interpreted broadly, to include human as well as non-human players (computers, animals, plants)."

– Robert Aumann, 1987

22.1 Introduction

In this chapter we look at a distributed system from a different perspective. Nodes no longer have a common goal, but are *selfish*. The nodes are not byzantine (actively malicious), instead they try to benefit from a distributed system – possibly without contributing.

Game theory attempts to mathematically capture behavior in strategic situations, in which an individual's success depends on the choices of others.

Remarks:

- Examples of potentially selfish behavior are file sharing or TCP. If a packet is dropped, then most TCP implementations interpret this as a congested network and alleviate the problem by reducing the speed at which packets are sent. What if a selfish TCP implementation will not reduce its speed, but instead transmit each packet twice?
- We start with one of the most famous games to introduce some definitions and concepts of game theory.

22.2 Prisoner's Dilemma

A team of two prisoners (players u and v) are being questioned by the police. They are both held in solitary confinement and cannot talk to each other. The prosecutors offer a bargain to each prisoner: snitch on the other prisoner to reduce your prison sentence.

v \ u		Player u: Cooperate	Player u: Defect
Player v	Cooperate	1 (for v), 1 (for u)	0 (for v), 3 (for u)
	Defect	3 (for v), 0 (for u)	2 (for v), 2 (for u)

Table 22.1: The prisoner's dilemma game as a matrix.

- If both of them stay silent (*cooperate*), both will be sentenced to one year of prison on a lesser charge.
- If both of them testify against their fellow prisoner (*defect*), the police has a stronger case and they will be sentenced to two years each.
- If player u defects and the player v cooperates, then player u will go free (snitching pays off) and player v will have to go to jail for three years; and vice versa.
- This two player game can be represented as a matrix, see Table 22.1.

Definition 22.2 (game)**.** *A game requires at least two rational players, and each player can choose from at least two options (*strategies*). In every possible outcome (*strategy profile*) each player gets a certain payoff (or cost). The payoff of a player depends on the strategies of the other players.*

Definition 22.3 (social optimum)**.** *A strategy profile is called social optimum (SO) if and only if it minimizes the sum of all costs (or maximizes payoff).*

Remarks:

- The social optimum for the prisoner's dilemma is when both players cooperate – the corresponding cost sum is 2.

Definition 22.4 (dominant)**.** *A strategy is dominant if a player is never worse off by playing this strategy. A dominant strategy profile is a strategy profile in which each player plays a dominant strategy.*

Remarks:

- The dominant strategy profile in the prisoner's dilemma is when both players defect – the corresponding cost sum is 4.

Definition 22.5 (Nash Equilibrium)**.** *A Nash Equilibrium (NE) is a strategy profile in which no player can improve by unilaterally (the strategies of the other players do not change) changing its strategy.*

Remarks:

- A game can have multiple Nash Equilibria.
- In the prisoner's dilemma both players defecting is the only Nash Equilibrium.
- If every player plays a dominant strategy, then this is by definition a Nash Equilibrium.
- Nash Equilibria and dominant strategy profiles are so called solution concepts. They are used to analyze a game. There are more solution concepts, e.g. correlated equilibria or best response.
- The best response is the best strategy given a belief about the strategy of the other players. In this game the best response to both strategies of the other player is to defect. If one strategy is the best response to any strategy of the other players, it is a dominant strategy.
- If two players play the prisoner's dilemma repeatedly, it is called iterated prisoner's dilemma. It is a dominant strategy to always defect. To see this, consider the final game. Defecting is a dominant strategy. Thus, it is fixed what both players do in the last game. Now the penultimate game is the last game and by induction always defecting is a dominant strategy.
- Game theorists were invited to come up with a strategy for 200 iterations of the prisoner's dilemma to compete in a tournament. Each strategy had to play against every other strategy and accumulated points throughout the tournament. The simple Tit4Tat strategy (cooperate in the first game, then copy whatever the other player did in the previous game) won. One year later, after analyzing each strategy, another tournament (with new strategies) was held. Tit4Tat won again.
- We now look at a distributed system game.

22.3 Selfish Caching

Computers in a network want to access a file regularly. Each node $v \in V$, with V being the set of nodes and $n = |V|$, has a demand d_v for the file and wants to minimize the cost for accessing it. In order to access the file, node v can either cache the file locally which costs 1 or request the file from another node u which costs $c_{v \leftarrow u}$. If a node does not cache the file, the cost it incurs is the minimal cost to access the file remotely. Note that if no node caches the file, then every node incurs cost ∞. There is an example in Figure 22.6.

Remarks:

- We will sometimes depict this game as a graph. The cost $c_{v \leftarrow u}$ for node v to access the file from node u is equivalent to the length of the shortest path times the demand d_v.

- Note that in undirected graphs $c_{u \leftarrow v} > c_{v \leftarrow u}$ if and only if $d_u > d_v$. We assume that the graphs are undirected for the rest of the chapter.

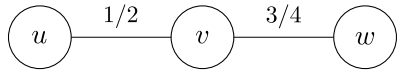


Figure 22.6: In this example we assume $d_u = d_v = d_w = 1$. Either the nodes u and w cache the file. Then neither of the three nodes has an incentive to change its behavior. The costs are 1, 1/2, and 1 for the nodes u, v, w, respectively. Alternatively, only node v caches the file. Again, neither of the three nodes has an incentive to change its behavior. The costs are 1/2, 1, and 3/4 for the nodes u, v, w, respectively.

Algorithm 22.7 Nash Equilibrium for Selfish Caching

```
1: S = {}                                   //set of nodes that cache the file
2: repeat
3:    Let v be a node with maximum demand d_v in set V
4:    S = S ∪ {v}, V = V \ {v}
5:    Remove every node u from V with c_{u←v} < 1
6: until V = {}
```

Theorem 22.8. *Algorithm 22.7 computes a Nash Equilibrium for Selfish Caching.*

Proof. Let u be a node that is not caching the file. Then there exists a node v for which $c_{u \leftarrow v} \leq 1$. Hence, node u has no incentive to cache.

Let u be a node that is caching the file. We now consider any other node v that is also caching the file. First, we consider the case where v cached the file before u did. Then it holds that $c_{u \leftarrow v} > 1$ by construction.

It could also be that v started caching the file after u did. Then it holds that $d_u \geq d_v$ and therefore $c_{u \leftarrow v} \geq c_{v \leftarrow u}$. Furthermore, we have $c_{v \leftarrow u} > 1$ by construction. Combining these implies that $c_{u \leftarrow v} \geq c_{v \leftarrow u} > 1$.

In either case, node u has no incentive to stop caching. □

Definition 22.9 (Price of Anarchy)**.** *Let NE_- denote the Nash Equilibrium with the highest cost (smallest payoff). The* Price of Anarchy *(PoA) is defined as*

$$PoA = \frac{\text{cost}(NE_-)}{\text{cost}(SO)}.$$

Definition 22.10 (Optimistic Price of Anarchy)**.** *Let NE_+ denote the Nash Equilibrium with the smallest cost (highest payoff). The* Optimistic Price of Anarchy *(OPoA) is defined as*

$$OPoA = \frac{\text{cost}(NE_+)}{\text{cost}(SO)}.$$

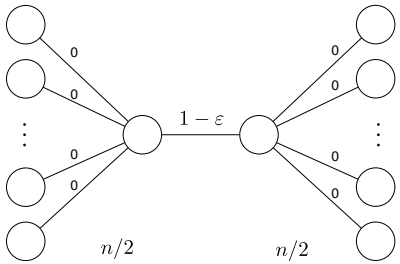


Figure 22.12: A network with a Price of Anarchy of $\Theta(n)$.

Remarks:

- The Price of Anarchy measures how much a distributed system degrades because of selfish nodes.
- We have $PoA \geq OPoA \geq 1$.

Theorem 22.11. *The (Optimistic) Price of Anarchy of Selfish Caching can be* $\Theta(n)$.

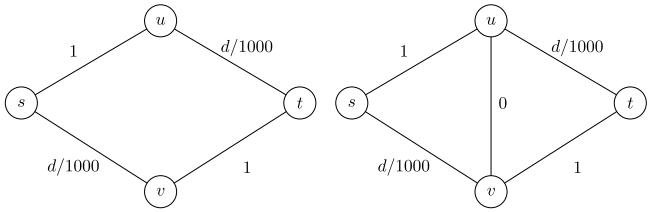
Proof. Consider a network as depicted in Figure 22.12. Every node v has demand $d_v = 1$. Note that if any node caches the file, no other node has an incentive to cache the file as well since the cost to access the file is at most $1-\varepsilon$. Wlog let us assume that a node v on the left caches the file, then it is cheaper for every node on the right to access the file remotely. Hence, the total cost of this solution is $1 + \frac{n}{2} \cdot (1-\varepsilon)$. In the social optimum one node from the left and one node from the right cache the file. This reduces the cost to 2. Hence, the Price of Anarchy is $\frac{1+\frac{n}{2}\cdot(1-\varepsilon)}{2} \underset{\varepsilon\to 0}{=} \frac{1}{2} + \frac{n}{4} = \Theta(n)$. □

22.4 Braess' Paradox

Consider the graph in Figure 22.13, it models a road network. Let us assume that there are 1000 drivers (each in their own car) that want to travel from node s to node t. Traveling along the road from s to u (or v to t) always takes 1 hour. The travel time from s to v (or u to t) depends on the traffic and increases by 1/1000 of an hour per car, i.e., when there are 500 cars driving, it takes 30 minutes to use this road.

Lemma 22.14. *Adding a super fast road (delay is 0) between* u *and* v *can increase the travel time from* s *to* t.

Proof. Since the drivers act rationally, they want to minimize the travel time. In the Nash Equilibrium, 500 drivers first drive to node u and then to t and 500 drivers first to node v and then to t. The travel time for each driver is 1 + 500 / 1000 = 1.5.



(a) The road network without the shortcut (b) The road network with the shortcut

Figure 22.13: Braess' Paradox, where d denotes the number of drivers using an edge.

To reduce congestion, a super fast road (delay is 0) is built between nodes u and v. This results in the following Nash Equilibrium: every driver now drives from s to v to u to t. The total cost is now $2 > 1.5$. □

Remarks:

- There are physical systems which exhibit similar properties. Some famous ones employ a spring. YouTube has some fascinating videos about this. Simply search for "Braess Paradox Spring".
- We will now look at another famous game that will allow us to deepen our understanding of game theory.

22.5 Rock-Paper-Scissors

There are two players, u and v. Each player simultaneously chooses one of three options: rock, paper, or scissors. The rules are simple: paper beats rock, rock beats scissors, and scissors beat paper. A matrix representation of this game is in Table 22.15.

	u	Player u		
v		Rock	Paper	Scissors
Player v	Rock	0 0	1 -1	-1 1
	Paper	-1 1	0 0	1 -1
	Scissors	1 -1	-1 1	0 0

Table 22.15: Rock-Paper-Scissors as a matrix.

Remarks:

- None of the three strategies is a Nash Equilibrium. Whatever player u chooses, player v can always switch her strategy such that she wins.
- This is highlighted in the best response concept. The best response to e.g. scissors is to play rock. The other player switches to paper. And so on.
- Is this a game without a Nash Equilibrium? John Nash answered this question in 1950. By choosing each strategy with a certain probability, we can obtain a so called mixed Nash Equilibrium. Indeed:

Theorem 22.16. *Every game has a mixed Nash Equilibrium.*

Remarks:

- The Nash Equilibrium of this game is if both players choose each strategy with probability 1/3. The expected payoff is 0.
- Any strategy (or mix of them) is a best response to a player choosing each strategy with probability 1/3.
- In a pure Nash Equilibrium, the strategies are chosen deterministically. Rock-Paper-Scissors does not have a pure Nash Equilibrium.
- Unfortunately, game theory does not always model problems accurately. Many real world problems are too complex to be captured by a game. And as you may know, humans (not only politicians) are often not rational.
- In distributed systems, players can be servers, routers, etc. Game theory can tell us whether systems and protocols are prone to selfish behavior.

22.6 Mechanism Design

Whereas game theory analyzes existing systems, there is a related area that focuses on designing games – mechanism design. The task is to create a game where nodes have an incentive to behave "nicely".

Definition 22.17 (auction)**.** *One good is sold to a group of bidders in an auction. Each bidder v_i has a secret value z_i for the good and tells his bid b_i to the auctioneer. The auctioneer sells the good to one bidder for a price p.*

Remarks:

- For simplicity, we assume that no two bids are the same, and that $b_1 > b_2 > b_3 > \dots$

Definition 22.19 (truthful)**.** *An auction is truthful if no player v_i can gain anything by not stating the truth, i.e., $b_i = z_i$.*

Algorithm 22.18 First Price Auction

1: every bidder v_i submits his bid b_i
2: the good is allocated to the highest bidder v_1 for the price $p = b_1$

Theorem 22.20. *A First Price Auction (Algorithm 22.18) is not truthful.*

Proof. Consider an auction with two bidders, with bids b_1 and b_2. By not stating the truth and decreasing his bid to $b_1 - \varepsilon > b_2$, player one could pay less and thus gain more. Thus, the first price auction is not truthful. □

Algorithm 22.21 Second Price Auction

1: every bidder v_i submits his bid b_i
2: the good is allocated to the highest bidder v_1 for $p = b_2$

Theorem 22.22. *Truthful bidding is a dominant strategy in a Second Price Auction.*

Proof. Let z_i be the truthful value of node v_i and b_i his bid. Let $b_{\max} = \max_{j \neq i} b_j$ is the largest bid from other nodes but v_i. The payoff for node v_i is $z_i - b_{\max}$ if $b_i > b_{\max}$ and 0 else. Let us consider overbidding first, i.e., $b_i > z_i$:

- If $b_{\max} < z_i < b_i$, then both strategies win and yield the same payoff $(z_i - b_{\max})$.
- If $z_i < b_i < b_{\max}$, then both strategies lose and yield a payoff of 0.
- If $z_i < b_{\max} < b_i$, then overbidding wins the auction, but the payoff $(z_i - b_{\max})$ is negative. Truthful bidding loses and yields a payoff of 0.

Likewise underbidding, i.e. $b_i < z_i$:

- If $b_{\max} < b_i < z_i$, then both strategies win and yield the same payoff $(z_i - b_{\max})$.
- If $b_i < z_i < b_{\max}$, then both strategies lose and yield a payoff of 0.
- If $b_i < b_{\max} < z_i$, then truthful bidding wins and yields a positive payoff $(z_i - b_{\max})$. Underbidding loses and yields a payoff of 0.

Hence, truthful bidding is a dominant strategy for each node v_i. □

Remarks:

- Let us use this for Selfish Caching. We need to choose a node that is the first to cache the file. But how? By holding an auction. Every node says for which price it is willing to cache the file. We pay the node with the lowest offer and pay it the second lowest offer to ensure truthful offers.
- Since a mechanism designer can manipulate incentives, she can implement a strategy profile by making all the strategies in this profile dominant.

Theorem 22.23. *Any Nash Equilibrium of Selfish Caching can be implemented for free.*

Proof. If the mechanism designer wants the nodes from the caching set S of the Nash Equilibrium to cache, then she can offer the following deal to every node not in S: If not every node from set S caches the file, then I will ensure a positive payoff for you. Thus, all nodes not in S prefer not to cache since this is a dominant strategy for them. Consider now a node $v \in S$. Since S is a Nash Equilibrium, node v incurs cost of at least 1 if it does not cache the file. For nodes that incur cost of exactly 1, the mechanism designer can even issue a penatly if the node does not cache the file. Thus, every node $v \in S$ caches the file. □

Remarks:

- Mechanism design assumes that the players act rationally and want to maximize their payoff. In real-world distributed systems some players may be not selfish, but actively malicious (byzantine).
- What about P2P file sharing? To increase the overall experience, BitTorrent suggests that peers offer better upload speed to peers who upload more. This idea can be exploited. By always claiming to have nothing to trade yet, the BitThief client downloads without uploading. In addition to that, it connects to more peers than the standard client to increase its download speed.
- Many techniques have been proposed to limit such free riding behavior, e.g., tit-for-tat trading: I will only share something with you if you share something with me. To solve the bootstrap problem ("I don't have anything yet"), nodes receive files or pieces of files whose hash match their own hash for free. One can also imagine indirect trading. Peer u uploads to peer v, who uploads to peer w, who uploads to peer u. Finally, one could imagine using virtual currencies or a reputation system (a history of who uploaded what). Reputation systems suffer from collusion and Sybil attacks. If one node pretends to be many nodes who rate each other well, it will have a good reputation.

Chapter Notes

Game theory was started by a proof for mixed-strategy equilibria in two-person zero-sum games by John von Neumann [Neu28]. Later, von Neumann and Morgenstern introduced game theory to a wider audience [NM44]. In 1950 John Nash proved that every game has a mixed Nash Equilibrium [Nas50]. The Prisoner's Dilemma was first formalized by Flood and Dresher [Flo52]. The iterated prisoner's dilemma tournament was organized by Robert Axelrod [AH81]. The Price of Anarchy definition is from Koutsoupias and Papadimitriou [KP99]. This allowed the creation of the Selfish Caching Game [CCW+04], which we used as a running example in this chapter. Braess' paradox was discovered by Dietrich Braess in 1968 [Bra68]. A generalized version of the second-price auction is the VCG auction, named after three successive papers from first Vickrey, then Clarke, and finally Groves [Vic61, Cla71, Gro73]. One popular example of selfishness in practice is BitThief – a BitTorrent client that successfully downloads without uploading [LMSW06]. Using game theory economists try to understand markets and predict crashes. Apart from John Nash, the Sveriges Riksbank Prize (Nobel Prize) in Economics has been awarded many times to game theorists. For example in 2007 Hurwicz, Maskin, and Myerson received the prize for "for having laid the foundations of mechanism design theory".

This chapter was written in collaboration with Philipp Brandes.

Bibliography

[AH81] Robert Axelrod and William Donald Hamilton. The evolution of cooperation. *Science*, 211(4489):1390–1396, 1981.

[Bra68] Dietrich Braess. Über ein paradoxon aus der verkehrsplanung. *Unternehmensforschung*, 12(1):258–268, 1968.

[CCW+04] Byung-Gon Chun, Kamalika Chaudhuri, Hoeteck Wee, Marco Barreno, Christos H Papadimitriou, and John Kubiatowicz. Selfish caching in distributed systems: a game-theoretic analysis. In *Proceedings of the twenty-third annual ACM symposium on Principles of distributed computing*, pages 21–30. ACM, 2004.

[Cla71] Edward H Clarke. Multipart pricing of public goods. *Public choice*, 11(1):17–33, 1971.

[Flo52] Merrill M Flood. Some experimental games. *Management Science*, 5(1):5–26, 1952.

[Gro73] Theodore Groves. Incentives in teams. *Econometrica: Journal of the Econometric Society*, pages 617–631, 1973.

[KP99] Elias Koutsoupias and Christos Papadimitriou. Worst-case equilibria. In *STACS 99*, pages 404–413. Springer, 1999.

[LMSW06] Thomas Locher, Patrick Moor, Stefan Schmid, and Roger Wattenhofer. Free Riding in BitTorrent is Cheap. In *5th Workshop on Hot Topics in Networks (HotNets), Irvine, California, USA*, November 2006.

[Nas50] John F. Nash. Equilibrium points in n-person games. *Proc. Nat. Acad. Sci. USA*, 36(1):48–49, 1950.

[Neu28] John von Neumann. Zur Theorie der Gesellschaftsspiele. *Mathematische Annalen*, 100(1):295–320, 1928.

[NM44] John von Neumann and Oskar Morgenstern. *Theory of games and economic behavior*. Princeton university press, 1944.

[Vic61] William Vickrey. Counterspeculation, auctions, and competitive sealed tenders. *The Journal of finance*, 16(1):8–37, 1961.

Chapter 23

Dynamic Networks

Many large-scale distributed systems and networks are dynamic. In some networks, e.g., peer-to-peer, nodes participate only for a short period of time, and the topology can change at a high rate. In wireless ad-hoc networks, nodes are mobile and move around. In this chapter, we will study how to solve some basic tasks if the network is dynamic. Under what conditions is it possible to compute an accurate estimate of the size or some other property of the system? How efficiently can information be disseminated reliably in the network? To what extent does stability in the communication graph help solve these problems?

There are various reasons why networks can change over time and as a consequence, there also is a wide range of possible models for dynamic networks. Nodes might join or leave a distributed system. Some components or communication links may fail in different ways. Especially if the network devices are mobile, the connectivity between them can change. Dynamic changes can occur constantly or they might be infrequent enough so that the system can adapt to each change individually.

We will look at a synchronous dynamic network model in which the graph can change from round to round in a worst-case manner. To simplify things (and to make the problems we study well-defined), we assume that the set of nodes in the network is fixed and does not change. However, we will make almost no assumptions how the set of edges changes over time. We require some guarantees about the connectivity, apart from this, in each round, the communication graph is chosen in a worst-case manner by an adversary.

23.1 Synchronous Edge-Dynamic Networks

We model a synchronous dynamic network by a dynamic graph $G = (V, E)$, where V is a static set of nodes, and $E : \mathbb{N}_0 \to \binom{V}{2}$ is a function mapping a round number $r \in \mathbb{N}_0$ to a set of undirected edges $E(r)$. Here $\binom{V}{2} := \{\{u, v\} \mid u, v \in V\}$ is the set of all possible undirected edges over V.

Definition 23.1 (T-Interval Connectivity)**.** *A dynamic graph $G = (V, E)$ is said to be* T-interval connected *for $T \in \mathbb{N}$ if for all $r \in \mathbb{N}$, the static graph $G_{r,T} := \left(V, \bigcap_{i=r}^{r+T-1} E(i)\right)$ is connected. If G is 1-interval connected we say that G is* always connected.

For simplicity, we restrict to deterministic algorithms. Nodes communicate with each other using *anonymous broadcast*. At the beginning of round r, each node u decides what message to broadcast based on its internal state; at the same time (and independently), the adversary chooses a set $E(r)$ of edges for the round. As in standard synchronous message passing, all nodes v for which $\{u, v\} \in E(r)$ receive the message broadcast by node u in round r and each node can perform arbitrary local computations upon receiving the messages from its neighbors. We assume that all nodes in the network have a unique identifier (ID). In most cases, we will assume that messages are restricted to $\mathcal{O}(\log n)$ bits. In these cases, we assume that node IDs can be represented using $\mathcal{O}(\log n)$ bits, so that a constant number of node IDs and some additional information can be transmitted in a single message. We refer to the special case where all nodes are woken up at once as *synchronous start* and to the general case as *asynchronous start*.

We assume that each node in the network starts an execution of the protocol in an initial state which contains its own ID and its input. Additionally, nodes know nothing about the network, and initially cannot distinguish it from any other network.

23.2 Problem Definitions

In the context of this chapter, we study the following problems.

Counting. An algorithm is said to solve the counting problem if whenever it is executed in a dynamic graph comprising n nodes, all nodes eventually terminate and output n.

***k*-verification.** Closely related to counting, the k-verification problem requires nodes to determine whether or not $n \leq k$. All nodes begin with k as their input, and must eventually terminate and output "yes" or "no". Nodes must output "yes" if and only if there are at most k nodes in the network.

***k*-token dissemination.** An instance of k-token dissemination is a pair (V, I), where $I : V \to \mathcal{P}(\mathcal{T})$ assigns a set of tokens from some domain $\mathcal{T}$ to each node, and $|\bigcup_{u \in V} I(v)| = k$. An algorithm solves k-token dissemination if for all instances (V, I), when the algorithm is executed in any dynamic graph $G = (V, E)$, all nodes eventually terminate and output $\bigcup_{u \in V} I(u)$. We assume that each token in the nodes' input is represented using $\mathcal{O}(\log n)$ bits. Nodes may or may not know k, depending on the context. Of particular interest is *all-to-all token dissemination*, a special case where $k = n$ and each node initially knows exactly one token, i.e., $|I(u)| = 1$ for all nodes u.

***k*-committee election.** As an useful step towards solving counting and token dissemination, we consider a problem called k-committee election. In this problem, nodes must partition themselves into sets, called *committees*, such that

a) the size of each committee is at most k and

b) if $k \geq n$, then there is just one committee containing all nodes.

Each committee has a unique committee ID, and the goal is for all nodes to eventually terminate and output a committee ID such that the two conditions are satisfied.

23.3 Basic Information Dissemination

To start, let us study how a single piece of information is propagated through a dynamic network. We assume that we have a dynamic network graph G with n nodes such that G is always connected (G is 1-interval connected as defined in Definition 23.1). Further assume that there is a single piece of information (token), which is initially known by a single node.

Theorem 23.2. *Assume that there is a single token in the network. Further assume that at time* 0 *at least one node knows the token and that once they know the token, all nodes broadcast it in every round. In a* 1*-interval connected graph* $G = (V, E)$ *with* n *nodes, after* $r \leq n-1$ *rounds, at least* $r+1$ *nodes know the token. Hence, in particular after* $n-1$ *rounds, all nodes know the token.*

Proof. We can proof the theorem by induction on r. Let $T(r)$ be the set of nodes that know the token after r rounds. We need to show that for all $r \geq 0$, $|T(r)| \geq \min\{r+1, n\}$. Because we assume that at time 0 at least one node knows the token, clearly, $|T(0)| \geq 1$. For the induction step, assume that after r rounds, $|T(r)| \geq \min\{r+1, n\}$. If $T(r) = V$, we have $|T(r+1)| \geq |T(r)| = n$ and we are done. Otherwise, we have $V \setminus T(r) \neq \emptyset$. Therefore, by the 1-interval connectivity assumption, there must be two nodes $u \in T(r)$ and $v \in V \setminus T(r)$ such that $\{u, v\} \in E(r+1)$. Hence, in round $r+1$, node v gets the token an therefore $|T(r+1)| \geq |T(r)| + 1 \geq \min\{r+2, n\}$. □

Remarks:

- Note that Theorem 23.2 only shows that after $n-1$ rounds all nodes know the token. If the nodes do not know n or an upper bound on n, they do not know if all nodes know the token.
- We can apply the above techniques also if there is more than one token in the network, provided that tokens form a totally-ordered set and nodes forward the smallest (or biggest) token they know. It is then guaranteed that the smallest (resp. biggest) token in the network will be known by all nodes after at most $n-1$ rounds. Note, however, that in this case nodes do not *know* when they know the smallest or biggest token.

The next theorem shows that essentially, for the general asynchronous start case, 1-interval connectivity does not suffice to obtain anything better than what is stated by the above theorem. If nodes do not know n or an upper bound on n initially, they cannot find n.

Theorem 23.3. *Counting is impossible in 1-interval connected graphs with asynchronous start.*

Proof. Suppose by way of contradiction that $\mathcal{A}$ is a protocol for counting which requires at most $t(n)$ rounds in 1-interval connected graphs of size n. Let $n' = \max\{t(n)+1, n+1\}$. We will show that the protocol cannot distinguish a static line of length n from a dynamically changing line of length n'.

Given a sequence $A = a_1 \circ \ldots \circ a_m$, let $\text{shift}(A, r)$ denote the cyclic left-shift of A in which the first r symbols ($r \geq 0$) are removed from the beginning of the sequence and appended to the end. Consider an execution in a dynamic line of length n', where the line in round r is composed of two adjacent sections $A \circ B_r$, where $A = 0 \circ \ldots \circ (n-1)$ remains static throughout the execution, and $B(r) = \text{shift}(n \circ \ldots \circ (n'-1), r)$ is left-shifted by one in every round. The computation is initiated by node 0 and all other nodes are initially asleep. We claim that the execution of the protocol in the dynamic graph $G = A \circ B(r)$ is indistinguishable in the eyes of nodes $0, \ldots, n-1$ from an execution of the protocol in the static line of length n (that is, the network comprising section A alone). This is proven by induction on the round number, using the fact that throughout rounds $0, \ldots, t(n)-1$ none of the nodes in section A ever receives a message from a node in section B: although one node in section B is awakened in every round, this node is immediately removed and attached at the end of section B, where it cannot communicate with the nodes in section A. Thus, the protocol cannot distinguish the dynamic graph A from the dynamic graph $A \circ B(r)$, and it produces the wrong output in one of the two graphs. □

Remark:

- The above impossibility result extends to all problems introduced in Section 23.2 as long as we do not assume that the nodes know n or an upper bound on n.

In light of the impossibility result of Theorem 23.3, let us now first consider the synchronous start case where all nodes start the protocol at time 0 (with round 1). We first look at the case where there is no bound on the message size and describe a simple linear-time protocol for counting (and token dissemination). The protocol is extremely simple, but it demonstrates some of the ideas used in some of the later algorithms, where we eliminate the large messages using a stability assumption (T-interval connectivity) which allows nodes to communicate with at least one of their neighbors for at least T rounds.

In the simple protocol, all nodes maintain a set A containing all the IDs they have collected so far. In every round, each node broadcasts A and adds any IDs it receives. Nodes terminate when they first reach a round r in which $|A| \leq r$.

```
A ← {self};
for r = 1, 2, ... do
    broadcast A;
    receive B_1, ..., B_s from neighbors;
    A ← A ∪ B_1 ∪ ... ∪ B_s;
    if |A| ≤ r then terminate and output |A|;
    ;
end
```

Algorithm 1: Counting in linear time using large messages

Before analyzing Algorithm 1, let us fix some notation that will help to argue about the algorithms we will study. If x is a variable of an algorithm, let $x_u(r)$

be the value of the variable x at node u after r rounds (immediately before the broadcast operation of round $r+1$). For instance in Algorithm 1, $A_u(r)$ denotes the set of IDs of node u at the end of the r^{th} iteration of the for-loop.

Lemma 23.4. *Assume that we are given an 1-interval connected graph $G = (V, E)$ and that all nodes in V execute Algorithm 1. If all nodes together start at time 0, we have $|A_u(r)| \geq r + 1$ for all $u \in V$ and $r < n$.*

Proof. We prove the lemma by induction on r. We clearly have $|A_u(0)| = 1$ for all u because initially each node includes its own ID in A. Hence, the lemma is true for $r = 0$.

For the induction step, assume that the claim of the lemma is true for some given $r < n-1$ for all dynamic graphs G. Let $A'_u(r+1)$ be the set of identifiers known by node u if all nodes start the protocol at time 1 (instead of 0) and run it for r rounds. By the induction hypothesis, we have $|A'_u(r+1)| \geq r+1$. If the algorithm is started at time 0 instead of time 1, the set of identifiers in $A_u(r+1)$ is exactly the union of all the identifiers known by the nodes in $A'_u(r+1)$ after the first round (at time 1). This includes all the nodes in $A'_u(r+1)$ as well as their neighbors in the first round. If $|A'_u(r+1)| \geq r+2$, we also have $|A_u(r+1)| \geq r+2$ and we are done. Otherwise, by 1-interval connectivity, there must at least be one node $v \in V \setminus A'_u(r+1)$ for which there is an edge to a node in $A'_u(r+1)$ in round 1. We therefore have $|A_u(r+1)| \geq |A'_u(r+1)| + 1 \geq r+2$. □

Theorem 23.5. *In an 1-interval connected graph G, Algorithm 1 terminates at all nodes after n rounds and output n.*

Proof. Follows directly from Lemma 23.4. For all nodes u, $|A_u(r)| \geq r+1 > r$ for all $r < n$ and $|A_u(n)| = |A_u(n-1)| = n$. □

Lemma 23.6. *Assume that we are given a 2-interval connected graph $G = (V, E)$ and that all nodes in V execute Algorithm 1. If node u is waken up and starts the algorithm at time t, it holds that have $|A_u(t + 2r)| \geq r + 1$ for all $0 \leq r < n$.*

Proof. The proof follows along the same lines as the proof of Lemma 23.4 (see exercises). □

Remarks:

- Because we did not bound the maximal message size and because every node receives information (an identifier) from each other node, Algorithm 1 can be used to solve all the problems defined in Section 23.2. For the token dissemination problem, the nodes also need to attach a list of all known tokens to all messages

- As a consequence of Theorem 23.3, 1-interval connectivity does not suffice to compute the number of nodes n in a dynamic network if nodes start asynchronously. It turns out that in this case, we need a slightly stronger connectivity assumption. If the network is 2-interval connected instead of 1-interval connected, up to a constant factor in the time complexity, the above results can also be obtained in the asynchronous start case (see exercises).

- For the remainder of the chapter, we will only consider the simpler synchronous start case. For $T \geq 2$, all discussed results that hold for T-interval connected networks with synchronous start also hold for asynchronous start with the same asymptotic bounds.

23.4 Small Messages

We now switch to the more interesting (and more realistic) case where in each round, each node can only broadcast a message of $\mathcal{O}(\log n)$ bits. We will first show how to use k-committee election to solve counting. We first describe how to obtain a good upper bound on n. We will then see that the same algorithm can also be used to find n exactly and to solve token dissemination.

23.4.1 k-Verification

The counting algorithm works by successive doubling: at each point the nodes have a guess k for the size of the network, and attempt to verify whether or not $k \geq n$. If it is discovered that $k < n$, the nodes double k and repeat; if $k \geq n$, the nodes halt and output the count.

Suppose that nodes start out in a state that represents a solution to k-committee election: each node has a committee ID, such that no more than k nodes have the same ID, and if $k \geq n$ then all nodes have the same committee ID. The problem of checking whether $k \geq n$ is then equivalent to checking whether there is more than one committee: if $k \geq n$ there must be one committee only, and if $k < n$ there must be more than one. Nodes can therefore check if $k \geq n$ by executing a simple k-round protocol that checks if there is more than one committee in the graph.

The k-verification protocol Each node has a local variable x, which is initially set to 1. While $x_u = 1$, node u broadcasts its committee ID. If it hears from some neighbor a different committee ID from its own, or the special value $\perp$, it sets $x_u \leftarrow 0$ and broadcasts $\perp$ in all subsequent rounds. After k rounds, all nodes output the value of their x variable.

Lemma 23.7. *If the initial state of the execution represents a solution to k-committee election, at the end of the k-verification protocol each node outputs 1 iff $k \geq n$.*

Proof. First suppose that $k \geq n$. In this case there is only one committee in the graph; no node ever hears a committee ID different from its own. After k rounds all nodes still have $x = 1$, and all output 1.

Next, suppose $k < n$. We can show that after the ith round of the protocol, at least i nodes in each committee have $x = 0$. In any round of the protocol, consider a cut between the nodes that belong to a particular committee and still have $x = 1$, and the rest of the nodes, which either belong to a different committee or have $x = 0$. From 1-interval connectivity, there is an edge in the cut, and some node u in the committee that still has $x_u = 1$ hears either a different committee ID or $\perp$. Node u then sets $x_u \leftarrow 0$, and the number of nodes in the committee that still have $x = 1$ decreases by at least one. Since

each committee initially contains at most k nodes, after k rounds all nodes in all committees have $x = 0$, and all output 0. □

23.4.2 k-Committee Election

We can solve k-committee in $\mathcal{O}(k^2)$ rounds as follows. Each node u stores two local variables, $committee_u$ and $leader_u$. A node that has not yet joined a committee is called *active*, and a node that has joined a committee is *inactive*. Once nodes have joined a committee they do not change their choice.

Initially all nodes consider themselves leaders, but throughout the protocol, any node that hears an ID smaller than its own adopts that ID as its leader. The protocol proceeds in k cycles, each consisting of two phases, *polling* and *selection*.

1. Polling phase: for $k-1$ rounds, all nodes propagate the ID of the smallest active node of which they are aware.

2. Selection phase: in this phase, each node that considers itself a leader selects the smallest ID it heard in the previous phase and invites that node to join its committee. An invitation is represented as a pair (x, y), where x is the ID of the leader that issued the invitation, and y is the ID of the invited node. All nodes propagate the smallest invitation of which they are aware for $k-1$ (invitations are sorted in lexicographic order, so the invitations issued by the smallest node in the network will win out over other invitations. It turns out, though, that this is not necessary for correctness; it is sufficient for each node to forward an arbitrary invitation from among those it received).

   At the end of the selection phase, a node that receives an invitation to join its leader's committee does so and becomes inactive. (Invitations issued by nodes that are not the current leader can be accepted or ignored; this, again, does not affect correctness.)

At the end of the k cycles, any node u that has not been invited to join a committee outputs $committee_u = u$. The details are given in Algorithm 2.

Lemma 23.8. *Algorithm 2 solves the k-committee problem in $\mathcal{O}(k^2)$ rounds in 1-interval connected networks.*

Proof. The time complexity is immediate. To prove correctness, we show that after the protocol ends, the values of the local $committee_u$ variables constitute a valid solution to k-committee.

1. In each cycle, each node invites at most one node to join its committee. After k cycles at most k nodes have joined any committee. Note that the first node invited by a leader u to join u's committee is always u itself. Thus, if after k cycles node u has not been invited to join a committee, it follows that u did not invite any other node to join its committee; when it forms its own committee in the last line of the algorithm, the committee's size is 1.

2. Suppose that $k \geq n$, and let u be the node with the smallest ID in the network. Following the polling phase of the first cycle, all nodes v have


```
leader ← self;
committee ← ⊥;
for i = 0, ..., k do
    // Polling phase
    if committee = ⊥ then
        min_active ← self ; // The node nominates itself for selection
    else
        min_active ← ⊥;
    end
    for j = 0, ..., k − 1 do
        broadcast min_active;
        receive x_1, ..., x_s from neighbors;
        min_active ← min {min_active, x_1, ..., x_s};
    end
    // Update leader
    leader ← min {leader, min_active};
    // Selection phase
    if leader = self then
        // Leaders invite the smallest ID they heard
        invitation ← (self, min_active);
    else
        // Non-leaders do not invite anybody
        invitation ← ⊥
    end
    for j = 0, ..., k − 1 do
        broadcast invitation;
        receive y_1, ..., y_s from neighbors;
        invitation ← min {invitation, y_1, ..., y_s} ; // (in lexicographic
        order)
    end
    // Join the leader's committee, if invited
    if invitation = (leader, self) then
        committee = leader;
    end
end
if committee = ⊥ then
    committee ← self;
end
```

Algorithm 2: k-committee in always-connected graphs

$leader_v = u$ for the remainder of the protocol. Thus, throughout the execution, only node u issues invitations, and all nodes propagate u's invitations. Since $k \geq n$ rounds are sufficient for u to hear the ID of the minimal active node in the network, in every cycle node u successfully identifies this node and invites it to join u's committee. After k cycles, all nodes will have joined.

□

Remark:

- The protocol can be modified easily to solve all-to-all token dissemination if $k \geq n$. Let t_u be the token node u received in its input (or $\perp$ if node u did not receive a token). Nodes attach their tokens to their IDs, and send pairs of the form (u, t_u) instead of just u. Likewise, invitations now contain the token of the invited node, and have the structure $(leader, (u, t_u))$. The min operation disregards the token and applies only to the ID. At the end of each selection phase, nodes extract the token of the invited node, and add it to their collection. By the end of the protocol every node has been invited to join the committee, and thus all nodes have seen all tokens.

23.5 More Stable Graphs

```
S ← ∅;
for i = 0, ..., ⌈k/T⌉ − 1 do
    for r = 0, ..., 2T − 1 do
        if S ≠ A then
            t ← min(A \ S);
            broadcast t;
            S ← S ∪ {t}
        end
        receive t_1, ..., t_s from neighbors;
        A ← A ∪ {t_1, ..., t_s}
    end
    S ← ∅
end
return A
```

Procedure disseminate(A, T, k)

In this section we show that in T-interval connected graphs the computation can be sped up by a factor of T. To do this we employ a neat pipelining effect, using the temporarily stable subgraphs that T-interval connectivity guarantees; this allows us to disseminate information more quickly. Basically, because we are guaranteed that some edges and paths persist for T rounds, it suffices to send a particular ID or token only once in T rounds to guarantee progress. Other rounds can then be used for different tokens. For convenience we assume that the graph is $2T$-interval connected for some $T \geq 1$.

Procedure `disseminate` gives an algorithm for exchanging at least T pieces of information in n rounds when the dynamic graph is $2T$-interval connected. The procedure takes three arguments: a set of tokens A, the parameter T, and a guess k for the size of the graph. If $k \geq n$, each node is guaranteed to learn the T smallest tokens that appeared in the input to all the nodes.

The execution of procedure `disseminate` is divided into $\lceil k/T \rceil$ phases, each consisting of $2T$ rounds. During each phase, each node maintains the set A of tokens it has already learned and a set S of tokens it has already broadcast in the current phase (initially empty). In each round of the phase, the node broadcasts the smallest token it has not yet broadcast in the current phase, then adds that token to S.

We refer to each iteration of the inner loop as a *phase*. Since a phase lasts $2T$ rounds and the graph is $2T$-interval connected, there is some connected subgraph that exists throughout the phase. Let G'_i be a connected subgraph that exists throughout phase i, for $i = 0, \ldots, \lceil k/T \rceil - 1$. We use $\text{dist}_i(u, v)$ to denote the distance between nodes $u, v \in V$ in G'_i.

Let $K_t(r)$ denote the set of nodes that know token t by the beginning of round r, that is, $K_t(r) = \{u \in V \mid t \in A_u(r)\}$. In addition, let I be the set of T smallest tokens in $\bigcup_{u \in V} A_u(0)$. Our goal is to show that when the protocol terminates we have $K_t(r) = V$ for all $t \in I$.

For a node $u \in V$, a token $t \in P$, and a phase i, we define $\text{tdist}_i(u, t)$ to be the distance of u from the nearest node in G'_i that knows t at the beginning of phase i:

$$\text{tdist}(u, t) := \min \{\text{dist}_i(u, v) \mid v \in K_t(2T \cdot i)\}.$$

Here and in the sequel, we use the convention that $\min \emptyset := \infty$. For convenience, we use $S^i_u(r) := S_u(2T \cdot i + r)$ to denote the value of S_u in round r of phase i. Similarly we denote $A^i_u(r) := A_u(2T \cdot i + r)$ and $K^i_t(r) := K_t(2T \cdot i + r)$. Correctness hinges on the following property.

Lemma 23.9. *For any node $u \in V$, token $t \in \bigcup_{v \in V} A_v(0)$, and round r such that $\text{tdist}_i(u, t) \leq r \leq 2T$, either $t \in S^i_u(r + 1)$ or $S_u(r + 1)$ includes at least $(r - \text{tdist}_i(u, t))$ tokens that are smaller than t.*

Proof. By induction on r. For $r = 0$ the claim is immediate.

Suppose the claim holds for round $r - 1$ of phase i, and consider round $r \geq \text{tdist}_i(u, t)$. If $r = \text{tdist}_i(u, t)$, then $r - \text{tdist}_i(u, t) = 0$ and the claim holds trivially. Thus, suppose that $r > \text{tdist}_i(u, t)$. Hence, $r - 1 \geq \text{tdist}_i(u, t)$, and the induction hypothesis applies: either $t \in S^i_u(r)$ or $S^i_u(r)$ includes at least $(r - 1 - \text{tdist}_i(u, t))$ tokens that are smaller than t. In the first case we are done, since $S^i_u(r) \subseteq S^i_u(r+1)$; thus, assume that $t \notin S^i_u(r)$, and $S^i_u(r)$ includes at least $(r - 1 - \text{tdist}_i(u, t))$ tokens smaller than t. However, if $S^i_u(r)$ includes at least $(r - \text{tdist}_i(u, t))$ tokens smaller than t, then so does $S^i_u(r + 1)$, and the claim is again satisfied; thus we assume that $S^i_u(r)$ includes *exactly* $(r - 1 - \text{tdist}_i(u, t))$ tokens smaller than t.

It is sufficient to prove that $\min\left(A^i_u(r) \setminus S^i_u(r)\right) \leq t$: if this holds, then in round r node u broadcasts $\min\left(A^i_u(r) \setminus S^i_u(r)\right)$, which is either t or a token smaller than t; thus, either $t \in S^i_u(r + 1)$ or $S^i_u(r + 1)$ includes at least $(r - \text{tdist}_i(u, t))$ tokens smaller than t, and the claim holds.

First we handle the case where $\text{tdist}_i(u,t) = 0$. In this case, $t \in A_u^i(0) \subseteq A_u^i(r)$. Since we assumed that $t \notin S_u^i(r)$ we have $t \in A_u^i(r) \setminus S_u^i(r)$, which implies that $\min\left(A_u^i(r) \setminus S_u^i(r)\right) \leq t$.

Next suppose that $\text{tdist}_i(u,t) > 0$. Let $x \in K_t^i(0)$ be a node such that $\text{dist}_i(u,x) = \text{tdist}(u,t)$ (such a node must exist from the definition of $\text{tdist}_i(u,t)$), and let v be a neighbor of u along the path from u to x in G_i, such that $\text{dist}_i(v,x) = \text{dist}_i(u,x) - 1 < r$. From the induction hypothesis, either $t \in S_v^i(r)$ or $S_v^i(r)$ includes at least $(r - 1 - \text{tdist}_i(v,t)) = (r - \text{tdist}_i(u,t))$ tokens that are smaller than t. Since the edge between u and v exists throughout phase i, node u receives everything v sends in phase i, and hence $S_v^i(r) \subseteq A_u^i(r)$. Finally, because we assumed that $S_u^i(r)$ contains exactly $(r - 1 - \text{tdist}_i(u,t))$ tokens smaller than t, and does not include t itself, we have $\min\left(A_u^i(r) \setminus S_u^i(r)\right) \leq t$, as desired. □

Using Lemma 23.9 we can show: correct.

Lemma 23.10. *If $k \geq n$, at the end of procedure* `disseminate` *the set A_u of each node u contains the T smallest tokens.*

Proof. Let $N_i^d(t) := \{u \in V \mid \text{tdist}_i(u,t) \leq d\}$ denote the set of nodes at distance at most d from some node that knows t at the beginning of phase i, and let t be one of the T smallest tokens.

From Lemma 23.9, for each node $u \in N_i^T(t)$, either $t \in S_u^i(2T+1)$ or $S_u^i(2T+1)$ contains at least $2T - T = T$ tokens that are smaller than t. But t is one of the T smallest tokens, so the second case is impossible. Therefore all nodes in $\mathbb{N}_i^T(t)$ know token t at the end of phase i. Because G_i is connected we have $|N_i^T(t)| \geq \min\{n - |K_i(t)|, T\}$; that is, in each phase T new nodes learn t, until all the nodes know t. Since there are no more than k nodes and we have $\lceil k/T \rceil$ phases, at the end of the last phase all nodes know t. □

To solve counting and token dissemination with up to n tokens, we use Procedure `disseminate` to speed up the k-committee election protocol from Algorithm 2. Instead of inviting one node in each cycle, we can use `disseminate` to have the leader learn the IDs of the T smallest nodes in the polling phase, and use procedure `disseminate` again to extend invitations to all T smallest nodes in the selection phase. Thus, in $\mathcal{O}(k+T)$ rounds we can increase the size of the committee by T.

Theorem 23.11. *It is possible to solve k-committee election in $\mathcal{O}(k + k^2/T)$ rounds in T-interval connected graphs. When used in conjunction with the k-verification protocol, this approach yields $\mathcal{O}(n+n^2/T)$-round protocols for counting all-to-all token dissemination.*

Remarks:

- The same result can also be achieved for the asynchronous start case, as long as $T \geq 2$.
- The described algorithm is based on the assumptions that all nodes know T (or that they have a common lower bound on T). At the cost of a log-factor, it is possible to drop this assumption and adapt to the actual interval-connectivity T.

- It is not known whether the bound of Theorem 23.11 is tight. It can be shown that it is tight for a restricted class of protocols (see exercises).
- If we make additional assumptions about the stable subgraphs that are guaranteed for intervals of length T, the bound in Theorem 23.11 can be improved. E.g., if intervals of length T induce a stable k-vertex connected subgraph, the complexity can be improved to $\mathcal{O}(n + n^2/(kT))$.

Chapter Notes

See [Sch10, BW05].

Bibliography

[BW05] Regina ODell Bischoff and Roger Wattenhofer. Information Dissemination in Highly Dynamic Graphs. In *3rd ACM Joint Workshop on Foundations of Mobile Computing (DIALM-POMC), Cologne, Germany*, September 2005.

[Sch10] Leonard J. Schulman, editor. *Proceedings of the 42nd ACM Symposium on Theory of Computing, STOC 2010, Cambridge, Massachusetts, USA, 5-8 June 2010*. ACM, 2010.

Chapter 24

All-to-All Communication

In the previous chapters, we have mostly considered communication on a particular graph $G = (V, E)$, where any two nodes u and v can only communicate directly if $\{u, v\} \in E$. This is however not always the best way to model a network. In the Internet, for example, every machine (node) is able to "directly" communicate with every other machine via a series of routers. If every node in a network can communicate directly with all other nodes, many problems can be solved easily. For example, assume we have n servers, each hosting an arbitrary number of (numeric) elements. If all servers are interested in obtaining the maximum of all elements, all servers can simultaneously, i.e., in one communication round, send their local maximum element to all other servers. Once these maxima are received, each server knows the global maximum.

Note that we can again use graph theory to model this *all-to-all* communication scenario: The communication graph is simply the complete graph $\mathcal{K}_n := (V, \binom{V}{2})$. If each node can send its entire local state in a single message, then all problems could be solved in 1 communication round in this model! Since allowing unbounded messages is not realistic in most practical scenarios, we restrict the message size: Assuming that all node identifiers and all other variables in the system (such as the numeric elements in the example above) can be described using $\mathcal{O}(\log n)$ bits, each node can only send a message of size $\mathcal{O}(\log n)$ bits to all other nodes (messages to different neighbors can be different). In other words, only a constant number of identifiers (and elements) can be packed into a single message. Thus, in this model, the limiting factor is the amount of information that can be transmitted in a fixed amount of time. This is fundamentally different from the model we studied before where nodes are restricted to local information about the network graph.

In this chapter, we study one particular problem in this model, the computation of a minimum spanning tree (MST), i.e., we will again look at the construction of a basic network structure. Let us first review the definition of a minimum spanning tree from Chapter 2. We assume that each edge e is assigned a weight ω_e.

Definition 24.1 (MST)**.** *Given a weighted graph $G = (V, E, \omega)$. The MST of G is a spanning tree T minimizing $\omega(T)$, where $\omega(H) = \sum_{e \in H} \omega_e$ for any subgraph $H \subseteq G$.*

Remarks:

- Since we have a complete communication graph, the graph has $\binom{n}{2}$ edges in the beginning.
- As in Chapter 2, we assume that no two edges of the graph have the same weight. Recall that this assumption ensures that the MST is unique. Recall also that this simplification is not essential as one can always break ties by using the IDs of adjacent vertices.

For simplicity, we assume that we have a synchronous model (as we are only interested in the time complexity, our algorithm can be made asynchronous using synchronizer α at no additional cost (cf. Chapter 10). As usual, in every round, every node can send a (potentially different) message to each of its neighbors. In particular, note that the message delay is 1 for every edge e independent of the weight ω_e. As mentioned before, every message can contain a constant number of node IDs and edge weights (and $\mathcal{O}(\log n)$ additional bits).

Remarks:

- Note that for graphs of arbitrary diameter D, if there are no bounds on the number of messages sent, on the message size, and on the amount of local computations, there is a straightforward generic algorithm to compute an MST in time D: In every round, every node sends its complete state to all its neighbors. After D rounds, every node knows the whole graph and can compute any graph structure locally without any further communication.
- In general, the diameter D is also an obvious lower bound for the time needed to compute an MST. In a weighted ring, e.g., it takes time D to find the heaviest edge. In fact, on the ring, time D is required to compute any spanning tree.

In this chapter, we are not concerned with lower bounds, we want to give an algorithm that computes the MST as quickly as possible instead! We again use the following lemma that is proven in Chapter 2.

Lemma 24.2. *For a given graph G let T be an MST, and let $T' \subseteq T$ be a subgraph (also known as a fragment) of the MST. Edge $e = (u, v)$ is an outgoing edge of T' if $u \in T'$ and $v \notin T'$ (or vice versa). Let the minimum weight outgoing edge of the fragment T' be the so-called blue edge $b(T')$. Then $T' \cup b(T') \subseteq T$.*

Lemma 24.2 leads to a straightforward distributed MST algorithm. We start with an empty graph, i.e., every node is a fragment of the MST. The algorithm consists of phases. In every phase, we add the blue edge $b(T')$ of every existing fragment T' to the MST. Algorithm 24.3 shows how the described simple MST construction can be carried out in a network of diameter 1.

Theorem 24.4. *On a complete graph, Algorithm 24.3 computes an MST in time $\mathcal{O}(\log n)$.*

Proof. The algorithm is correct because of Lemma 24.2. Every node only needs to send a single message to all its neighbors in every phase (line 4). All other computations can be done locally without sending other messages. In particular,

Algorithm 24.3 Simple MST Construction (at node v)

```
1: // all nodes always know all current MST edges and thus all MST fragments
2: while v has neighbor u in different fragment do
3:    find lowest-weight edge e between v and a node u in a different fragment
4:    send e to all nodes
5:    determine blue edges of all fragments
6:    add blue edges of all fragments to MST, update fragments
7: end while
```

the blue edge of a given fragment is the lightest edge sent by any node of that fragment. Because every node always knows the current MST (and all current fragments), lines 5 and 6 can be performed locally.

In every phase, every fragment connects to at least one other fragment. The minimum fragment size therefore at least doubles in every phase. Thus, the number of phases is at most $\log_2 n$. □

Remarks:

- Algorithm 24.3 does essentially the same thing as the GHS algorithm (Algorithm 2.18) discussed in Chapter 2. Because we now have a complete graph and thus every node can communicate with every other node, things get simpler (and also much faster).
- Algorithm 24.3 does not make use of the fact that a node can send different messages to different nodes. Making use of this possibility will allow us to significantly reduce the running time of the algorithm.

Our goal is now to improve Algorithm 24.3. We assume that every node has a unique identifier. By sending its own identifier to all other nodes, every node knows the identifiers of all other nodes after one round. Let $\ell(F)$ be the node with the smallest identifier in fragment F. We call $\ell(F)$ the leader of fragment F. In order to improve the running time of Algorithm 24.3, we need to be able to connect every fragment to more than one other fragment in a single phase. Algorithm 24.5 shows how the nodes can learn about the $k = |F|$ lightest outgoing edges of each fragment F (in constant time!).

Given this set E' of edges, each node can locally decide which edges can safely be added to the constructed tree by calling the subroutine AddEdges (Algorithm 24.6). Note that the set of received edges E' in line 14 is the same for all nodes. Since all nodes know all current fragments, all nodes add the same set of edges!

Algorithm 24.6 uses the lightest outgoing edge that connects two fragments (to a larger super-fragment) as long as it is safe to add this edge, i.e., as long as it is clear that this edge is a blue edge. A (super-)fragment that has outgoing edges in E' that are surely blue edges is called *safe*. As we will see, a super-fragment $\mathcal{F}$ is safe if all the original fragments that make up $\mathcal{F}$ are still incident to at least one edge in E' that has not yet been considered. In order to determine whether all lightest outgoing edges in E' that are incident to a certain fragment F have been processed, a counter $c(F)$ is maintained (see line 2). If an edge incident to two (distinct) fragments F_i and F_j is processed, both $c(F_i)$ and $c(F_j)$ are decremented by 1 (see Line 8).

Algorithm 24.5 Fast MST construction (at node v)

```
1: // all nodes always know all current MST edges and thus all MST fragments
2: repeat
3:    F := fragment of v;
4:    ∀F' ≠ F, compute min-weight edge e_F' connecting v to F'
5:    ∀F' ≠ F, send e_F' to ℓ(F')
6:    if v = ℓ(F) then
7:       ∀F' ≠ F, determine min-weight edge e_F,F' between F and F'
8:       k := |F|
9:       E(F) := k lightest edges among e_F,F' for F' ≠ F
10:      send each edge in E(F) to a different node in F
                    // for simplicity assume that v also sends an edge to itself
11:   end if
12:   send edge received from ℓ(F) to all nodes
13:   // the following operations are performed locally by each node
14:   E' := edges received by other nodes
15:   AddEdges(E')
16: until all nodes are in the same fragment
```

An edge connecting two distinct super-fragments $\mathcal{F}'$ and $\mathcal{F}''$ is added if at least one of the two super-fragments is safe. In this case, the two super-fragments are merged into one (new) super-fragment. The new super-fragment is safe if and only if both original super-fragements are safe and the processed edge e is not the last edge in E' incident to any of the two fragments F_i and F_j that are incident to e, i.e., both counters $c(F_i)$ and $c(F_j)$ are still positive (see line 12).

The considered edge e may not be added for one of two reasons. It is possible that both $\mathcal{F}'$ and $\mathcal{F}''$ are not safe. Since a super-fragment cannot become safe again, nothing has to be done in this case. The second reason is that $\mathcal{F}' = \mathcal{F}''$. In this case, this single fragment may become unsafe if e reduced either $c(F_i)$ or $c(F_j)$ to zero (see line 18).

Lemma 24.7. *The algorithm only adds MST edges.*

Proof. We have to prove that at the time we add an edge e in line 9 of Algorithm 24.6, e is the blue edge of some (super-)fragment. By definition, e is the lightest edge that has not been considered and that connects two distinct super-fragments $\mathcal{F}'$ and $\mathcal{F}''$. Since e is added, we know that either $safe(\mathcal{F}')$ or $safe(\mathcal{F}'')$ is true. Without loss of generality, assume that $\mathcal{F}'$ is safe. According to the definition of *safe*, this means that from each fragment F in the super-fragment $\mathcal{F}'$ we know at least the lightest outgoing edge, which implies that we also know the lightest outgoing edge, i.e., the blue edge, of $\mathcal{F}'$. Since e is the lightest edge that connects *any* two super-fragments, it must hold that e is exactly the blue edge of $\mathcal{F}'$. Thus, whenever an edge is added, it is an MST edge. □

Theorem 24.8. *Algorithm 24.5 computes an MST in time $\mathcal{O}(\log \log n)$.*

Proof. Let β_k denote the size of the smallest fragment after phase k of Algorithm 24.5. We first show that every fragment merges with at least β_k other fragments in each phase. Since the size of each fragment after phase k is at least

Algorithm 24.6 AddEdges(E'): Given the set of edges E', determine which edges are added to the MST

```
1: Let F_1, ..., F_r be the initial fragments
2: ∀F_i ∈ {F_1, ..., F_r}, c(F_i) := # incident edges in E'
3: Let F_1 := F_1, ..., F_r := F_r be the initial super-fragments
4: ∀F_i ∈ {F_1, ..., F_r}, safe(F_i) := true
5: while E' ≠ ∅ do
6:    e := lightest edge in E' between the original fragments F_i and F_j
7:    E' := E' \ {e}
8:    c(F_i) := c(F_i) − 1, c(F_j) := c(F_j) − 1
9:    if e connects super-fragments F' ≠ F'' and (safe(F') or safe(F'')) then
10:      add e to MST
11:      merge F' and F'' into one super-fragment F_new
12:      if safe(F') and safe(F'') and c(F_i) > 0 and c(F_j) > 0 then
13:         safe(F_new) := true
14:      else
15:         safe(F_new) := false
16:      end if
17:   else if F' = F'' and (c(F_i) = 0 or c(F_j) = 0) then
18:      safe(F') := false
19:   end if
20: end while
```

β_k by definition, we get that the size of each fragment after phase $k+1$ is at least $\beta_k(\beta_k+1)$. Assume that a fragment F, consisting of at least β_k nodes, does not merge with β_k other fragments in phase $k+1$ for any $k \geq 0$. Note that F cannot be safe because being safe implies that there is at least one edge in E' that has not been considered yet and that is the blue edge of F. Hence, the phase cannot be completed in this case. On the other hand, if F is not safe, then at least one of its sub-fragments has used up all its β_k edges to other fragments. However, such an edge is either used to merge two fragments or it must have been dropped because the two fragments already belong to the same fragment because another edge connected them (in the same phase). In either case, we get that any fragment, and in particular F, must merge with at least β_k other fragments.

Given that the minimum fragment size grows (quickly) in each phase and that only edges belonging to the MST are added according to Lemma 24.7, we conclude that the algorithm correctly computes the MST. The fact that

$$\beta_{k+1} \geq \beta_k(\beta_k+1)$$

implies that $\beta_k \geq 2^{2^{k-1}}$ for any $k \geq 1$. Therefore after $1+\log_2 \log_2 n$ phases, the minimum fragment size is n and thus all nodes are in the same fragment. □

Chapter Notes

There is a considerable amount of work on distributed MST construction. Table 24.9 lists the most important results for various network diameters D. In the above text we focus only on $D = 1$.

Upper Bounds

Graph Class	Time Complexity	Authors
General Graphs	$\mathcal{O}(D+\sqrt{n}\cdot\log^* n)$	Kutten, Peleg [KP95]
Diameter 2	$\mathcal{O}(\log n)$	Lotker, Patt-Shamir, Peleg [LPSP06]
Diameter 1	$\mathcal{O}(\log\log n)$	Lotker, Patt-Shamir, Pavlov, Peleg [LPPSP03]

Lower Bounds

Graph Class	Time Complexity	Authors
Diameter $\Omega(\log n)$	$\Omega(D+\sqrt{n}/\log n)$	Das Sarma, Holzer, Kor, Korman, Nanongkai, Pandurangan, Peleg, Wattenhofer [SHK+12]
Diameter 4	$\Omega\left((n/\log n)^{1/3}\right)$	Das Sarma, Holzer, Kor, Korman, Nanongkai, Pandurangan, Peleg, Wattenhofer [SHK+12]
Diameter 3	$\Omega\left((n/\log n)^{1/4}\right)$	Das Sarma, Holzer, Kor, Korman, Nanongkai, Pandurangan, Peleg, Wattenhofer [SHK+12]

Table 24.9: Time complexity of distributed MST construction

We want to remark that the above lower bounds remain true for randomized algorithms. We can even not hope for a better randomized approximation algorithm for the MST as long as the approximation factor is bounded polynomially in n. On the other hand it is not known whether the $\mathcal{O}(\log\log n)$ time complexity of Algorithm 24.5 is optimal. In fact, no lower bounds are known for the MST construction on graphs of diameter 1 and 2. Algorithm 24.5 makes use of the fact that it is possible to send different messages to different nodes. If we assume that every node always has to send the same message to all other nodes, Algorithm 24.3 is the best that is known. Also for this simpler case, no lower bound is known.

Bibliography

[KP95] Shay Kutten and David Peleg. Fast distributed construction of k-dominating sets and applications. In *Proceedings of the fourteenth annual ACM symposium on Principles of distributed computing*, pages 238–251. ACM, 1995.

[LPPSP03] Zvi Lotker, Elan Pavlov, Boaz Patt-Shamir, and David Peleg. Mst

construction in o (log log n) communication rounds. In *Proceedings of the fifteenth annual ACM symposium on Parallel algorithms and architectures*, pages 94–100. ACM, 2003.

[LPSP06] Zvi Lotker, Boaz Patt-Shamir, and David Peleg. Distributed mst for constant diameter graphs. *Distributed Computing*, 18(6):453–460, 2006.

[SHK+12] Atish Das Sarma, Stephan Holzer, Liah Kor, Amos Korman, Danupon Nanongkai, Gopal Pandurangan, David Peleg, and Roger Wattenhofer. Distributed verification and hardness of distributed approximation. *SIAM Journal on Computing*, 41(5):1235–1265, 2012.

Chapter 25

Multi-Core Computing

This chapter is based on the article"Distributed Computing and the Multicore Revolution" by Maurice Herlihy and Victor Luchangco. Thanks!

25.1 Introduction

In the near future, nearly all computers, ranging from supercomputers to cell phones, will be multiprocessors. It is harder and harder to increase processor clock speed (the chips overheat), but easier and easier to cram more processor cores onto a chip (thanks to Moore's Law). As a result, uniprocessors are giving way to dual-cores, dual-cores to quad-cores, and so on.

However, there is a problem: Except for "embarrassingly parallel" applications, no one really knows how to exploit lots of cores.

25.1.1 The Current State of Concurrent Programming

In today's programming practice, programmers typically rely on combinations of locks and conditions, such as monitors, to prevent concurrent access by different threads to the same shared data. While this approach allows programmers to treat sections of code as "atomic", and thus simplifies reasoning about interactions, it suffers from a number of severe shortcomings.

- Programmers must decide between *coarse-grained* locking, in which a large data structure is protected by a single lock (usually implemented using operations such as test-and-set or compare and swap(CAS)), and *fine-grained* locking, in which a lock is associated with each component of the data structure. Coarse-grained locking is simple, but permits little or no concurrency, thereby preventing the program from exploiting multiple processing cores. By contrast, fine-grained locking is substantially more complicated because of the need to ensure that threads acquire all necessary locks (and only those, for good performance), and because of the need to avoid deadlocks, when acquiring multiple locks. The decision is further complicated by the fact that the best engineering solution may be

Algorithm Move(Element e, Table from, Table to)

```
1: if from.find(e) then
2:    to.insert(e)
3:    from.delete(e)
4: end if
```

platform-dependent, varying with different machine sizes, workloads, and so on, making it difficult to write code that is both scalable and portable.

- Conventional locking provides poor support for code composition and reuse. For example, consider a lock-based hash table that provides atomic `insert` and `delete` methods. Ideally, it should be easy to *move* an element atomically from one table to another, but this kind of composition simply does not work. If the table methods synchronize internally, then there is no way to acquire and hold both locks simultaneously. If the tables export their locks, then modularity and safety are compromised. For a concrete example, assume we have two hash tables T_1 and T_2 storing integers and using internal locks only. Every number is only inserted into a table, if it is not already present, i.e., multiple occurrences are not permitted. We want to atomically move elements using two threads between the tables using Algorithm Move. If we have external locks, we must pay attention to avoid deadlocks etc.

	Table T1 is contains 1 and T2 is empty	
Time	Thread 1	Thread 2
	Move(1,T1,T2)	Move(1,T2,T1)
1	T1.find(1)	delayed
2	T2.insert(1)	
3	delayed	T2.find(1)
4		T1.insert(1)
5	T1.delete(1)	T2.delete(1)
	both T1 and T2 are empty	

- Such basic issues as the mapping from locks to data, that is, which locks protect which data, and the order in which locks must be acquired and released, are all based on convention, and violations are notoriously difficult to detect and debug. For these and other reasons, today's software practices make lock-based concurrent programs (too) difficult to develop, debug, understand, and maintain.

  The research community has addressed this issue for more than fifteen years by developing nonblocking algorithms for stacks, queues and other data structures. These algorithms are subtle and difficult. For example, the pseudo code of a delete operation for a (non-blocking) linked list, recently presented at a conference, contains more than 30 lines of code, whereas a delete procedure for a (non-concurrent, used only by one thread) linked list can be written with 5 lines of code.

25.2 Transactional Memory

Recently the *transactional memory* programming paradigm has gained momentum as an alternative to locks in concurrent programming. Rather than using locks to give the illusion of atomicity by preventing concurrent access to shared data with transactional memory, programmers designate regions of code as transactions, and the system guarantees that such code appears to execute atomically. A transaction that cannot complete is aborted—its effects are discarded—and may be retried. Transactions have been used to build large, complex and reliable database systems for over thirty years; with transactional memory, researchers hope to translate that success to multiprocessor systems. The underlying system may use locks or nonblocking algorithms to implement transactions, but the complexity is hidden from the application programmer. Proposals exist for implementing transactional memory in hardware, in software, and in schemes that mix hardware and software. This area is growing at a fast pace.

More formally, a transaction is defined as follows:

Definition 25.1. *A transaction in transactional memory is characterized by three properties (ACI):*

- *Atomicity: Either a transaction finishes all its operations or no operation has an effect on the system.*
- *Consistency: All objects are in a valid state before and after the transaction.*
- *Isolation: Other transactions cannot access or see data in an intermediate (possibly invalid) state of any parallel running transaction.*

Remarks:

- For database transactions there exists a fourth property called durability: If a transaction has completed, its changes are permanent, i.e., even if the system crashes, the changes can be recovered. In principle, it would be feasible to demand the same thing for transactional memory, however this would mean that we had to use slow hard discs instead of fast DRAM chips...
- Although transactional memory is a promising approach for concurrent programming, it is not a panacea, and in any case, transactional programs will need to interact with other (legacy) code, which may use locks or other means to control concurrency.
- One major challenge for the adoption of transactional memory is that it has no universally accepted specification. It is not clear yet how to interact with I/O and system calls should be dealt with. For instance, imagine you print a news article. The printer job is part of a transaction. After printing half the page, the transaction gets aborted. Thus the work (printing) is lost. Clearly, this behavior is not acceptable.
- From a theory perspective we also face a number of open problems. For example:

  - System model: An abstract model for a (shared-memory) multiprocessor is needed that properly accounts for performance. In the 80s, the PRAM model became a standard model for parallel computation, and the research community developed many elegant parallel algorithms for this model. Unfortunately, PRAM assume that processors are synchronous, and that memory can be accessed only by read and write operations. Modern computer architectures are asynchronous and they provide additional operations such as test-and-set. Also, PRAM did not model the effects of contention nor the performance implications of multilevel caching, assuming instead a flat memory with uniform-cost access. More realistic models have been proposed to account for the costs of interprocess communication, but these models still assume synchronous processors with only read and write access to memory.
  - How to resolve conflicts? Many transactional memory implementations "optimistically" execute transactions in parallel. Conflicts between two transactions intending to modify the same memory at the same time are resolved by a contention manager. A contention manager decides whether a transaction continues, waits or is aborted. The contention management policy of a transactional memory implementation can have a profound effect on its performance, and even its progress guarantees.

25.3 Contention Management

After the previous introduction of transactional memory, we look at different aspects of contention management from a theoretical perspective. We start with a description of the model.

We are given a set of *transactions* $S := \{T_1, ..., T_n\}$ sharing up to s *resources* (such as memory cells) that are executed on n *threads*. Each thread runs on a separate processor/core $P_1, ..., P_n$. For simplicity, each transaction T consists of a sequence of t_T operations. An operation requires one time unit and can be a write access of a resource R or some arbitrary computation.[1] To perform a write, the written resource must be acquired exclusively (i.e., locked) before the access. Additionally, a transaction must store the original value of a written resource. Only one transaction can lock a resource at a time. If a transaction A attempts to acquire a resource, locked by B, then A and B face a conflict. If multiple transactions concurrently attempt to acquire an unlocked resource, an arbitrary transaction A will get the resource and the others face a conflict with A. A *contention manager* decides how to resolve a conflict. Contention managers operate in a distributed fashion, that is to say, a separate instance of a contention manager is available for every thread and they operate independently. Contention managers can make a transaction wait (arbitrarily long) or abort. An aborted transaction undoes all its changes to resources and frees all locks before restarting. Freeing locks and undoing the changes can be done with one operation. A successful transaction finishes with a commit and simply frees

[1] Reads are of course also possible, but are not critical because they do not attempt to modify data.

all locks. A contention manager is unaware of (potential) future conflicts of a transaction. The required resources might also change at any time.

The quality of a contention manager is characterized by different properties:

- Throughput: How long does it take until all transactions have committed? How good is our algorithm compared to an optimal?

  Definition 25.2. *The makespan of the set S of transactions is the time interval from the start of the first transaction until all transactions have committed.*

  Definition 25.3. *The* competitive ratio *is the ratio of the makespans of the algorithm to analyze and an optimal algorithm.*

- Progress guarantees: Is the system deadlock-free? Does every transaction commit in finite time?

  Definition 25.4. *We look at three levels of progress guarantees:*

  - *wait freedom (strongest guarantee): all threads make progress in a finite number of steps*
  - *lock freedom: one thread makes progress in a finite number of steps*
  - *obstruction freedom (weakest): one thread makes progress in a finite number of steps in absence of contention (no other threads compete for the same resources)*

Remarks:

- For the analysis we assume an *oblivious* adversary. It knows the algorithm to analyze and chooses/modifies the operations of transactions arbitrarily. However, the adversary does not know the random choices (of a randomized algorithm). The optimal algorithm knows all decisions of the adversary, i.e. first the adversary must say how transactions look like and then the optimal algorithm, having full knowledge of all transaction, computes an (optimal) schedule.
- Wait freedom implies lock freedom. Lock freedom implies obstruction freedom.
- Here is an example to illustrate how needed resources change over time: Consider a dynamic data structure such as a balanced tree. If a transaction attempts to insert an element, it must modify a (parent) node and maybe it also has to do some rotations to rebalance the tree. Depending on the elements of the tree, which change over time, it might modify different objects. For a concrete example, assume that the root node of a binary tree has value 4 and the root has a (left) child of value 2. If a transaction A inserts value 5, it must modify the pointer to the right child of the root node with value 4. Thus it locks the root node. If A gets aborted by a transaction B, which deletes the node with value 4 and commits, it will attempt to lock the new root node with value 2 after its restart.

- There are also systems, where resources are not locked exclusively. All we need is a correct serialization (analogous to transactions in database systems). Thus a transaction might speculatively use the current value of a resource, modified by an uncommitted transaction. However, these systems must track dependencies to ensure the ACI properties of a transaction (see Definition 25.1). For instance, assume a transaction T_1 increments variable x from 1 to 2. Then transaction T_2 might access x and assume its correct value is 2. If T_1 commits everything is fine and the ACI properties are ensured, but if T_1 aborts, T_2 must abort too, since otherwise the atomicity property was violated.
- In practice, the number of concurrent transactions might be much larger than the number of processors. However, performance may decrease with an increasing number of threads since there is time wasted to switch between threads. Thus, in practice, load adaption schemes have been suggested to limit the number of concurrent transactions close to (or even below) the number of cores.
- In the analysis, we will assume that the number of operations is fixed for each transaction. However, the execution time of a transaction (in the absence of contention) might also change, e.g., if data structures shrink, less elements have to be considered. Nevertheless, often the changes are not substantial, i.e., only involve a constant factor. Furthermore, if an adversary can modify the duration of a transaction arbitrarily during the execution of a transaction, then any algorithm must make the exact same choices as an optimal algorithm: Assume two transactions T_0 and T_1 face a conflict and an algorithm *Alg* decides to let T_0 wait (or abort). The adversary could make the opposite decision and let T_0 proceed such that it commits at time t_0. Then it sets the execution time T_0 to infinity, i.e., $t_{T_0} = \infty$ after t_0. Thus, the makespan of the schedule for algorithm *Alg* is unbounded though there exists a schedule with bounded makespan. Thus the competitive ratio is unbounded.

Problem complexity

In graph theory, coloring a graph with as few colors as possible is known to be hard problem. A (vertex) coloring assigns a color to each vertex of a graph such that no two adjacent vertices share the same color. It was shown that computing an optimal coloring given complete knowledge of the graph is NP-hard. Even worse, computing an approximation within a factor of $\chi(G)^{\log \chi(G)/25}$, where $\chi(G)$ is the minimal number of colors needed to color the graph, is NP-hard as well.

To keep things simple, we assume for the following theorem that resource acquisition takes no time, i.e., as long as there are no conflicts, transactions get all locks they wish for at once. In this case, there is an immediate connection to graph coloring, showing that even an *offline* version of contention management, where all potential conflicts are known and do not change over time, is extremely hard to solve.

Theorem 25.5. *If the optimal schedule has makespan k and resource acquisition takes zero time, it is NP-hard to compute a schedule of makespan less than $k^{\log k/25}$, even if all conflicts are known and transactions do not change their resource requirements.*

Proof. We will prove the claim by showing that any algorithm finding a schedule taking $k' < k^{(\log k)/25}$ can be utilized to approximate the chromatic number of any graph better than $\chi(G)^{\frac{\log \chi(G)}{25}}$.

Given the graph $G = (V, E)$, define that V is the set of transactions and E is the set of resources. Each transaction (node) $v \in V$ needs to acquire a lock on all its resources (edges) $\{v, w\} \in E$, and then computes something for exactly one round. Obviously, this "translation" of a graph into our scheduling problem does not require any computation at all.

Now, if we knew a $\chi(G)$-coloring of G, we could simply use the fact that the nodes sharing one color form an independent set and execute all transactions of a single color in parallel and the colors sequentially. Since no two neighbors are in an independent set and resources are edges, all conflicts are resolved. Consequently, the makespan k is at most $\chi(G)$.

On the other hand, the makespan k must be at least $\chi(G)$: Since each transaction (i.e., node) locks all required resources (i.e., adjacent edges) for at least one round, no schedule could do better than serve a (maximum) independent set in parallel while all other transactions wait. However, by definition of the chromatic number $\chi(G)$, V cannot be split into less than $\chi(G)$ independent sets, meaning that $k \geq \chi(G)$. Therefore $k = \chi(G)$.

In other words, if we could compute a schedule using $k' < k^{(\log k)/25}$ rounds in polynomial time, we knew that

$$\chi(G) = k \leq k' < k^{(\log k)/25} = \chi(G)^{(\log \chi(G))/25}.$$

□

Remarks:

- The theorem holds for a central contention manager, knowing all transactions and all potential conflicts. Clearly, the *online* problem, where conflicts remain unknown until they occur, is even harder. Furthermore, the distributed nature of contention managers also makes the problem even more difficult.
- If resource acquisition does not take zero time, the connection between the problems is not a direct equivalence. However, the same proof technique shows that it is NP-hard to compute a polynomial approximation, i.e., $k' \leq k^c$ for some constant $c \geq 1$.

Deterministic contention managers

Theorem 25.5 showed that even if all conflicts are known, one cannot produce schedules which makespan get close to the optimal without a lot of computation. However, we target to construct contention managers that make their decisions quickly without knowing conflicts in advance. Let us look at a couple of contention managers and investigate their throughput and progress guarantees.

- A first naive contention manger: Be aggressive! Always abort the transaction having locked the resource. Analysis: The throughput might be zero, since a livelock is possible. But the system is still obstruction free. Consider two transactions consisting of three operations. The first operation of both is a write to the same resource R. If they start concurrently, they will abort each other infinitely often.
- A smarter contention manager: Approximate the work done. Assume before a start (also before a restart after an abort) a transaction gets a unique timestamp. The older transaction, which is believed to have already performed more work, should win the conflict.

  Analysis: Clearly, the oldest transaction will always run until commit without interruption. Thus we have lock-freedom, since at least one transaction makes progress at any time. In other words, at least one processor is always busy executing a transaction until its commit. Thus, the bound says that all transactions are executed sequentially. How about the competitive ratio? We have s resources and n transactions starting at the same time. For simplicity, assume every transaction T_i needs to lock at least one resource for a constant fraction c of its execution time t_{T_i}. Thus, at most s transactions can run concurrently from start until commit without (possibly) facing a conflict (if $s+1$ transactions run at the same time, at least two of them lock the same resource). Thus, the makespan of an optimal contention manager is at least: $\sum_{i=0}^{n} \frac{c \cdot t_{T_i}}{s}$. The makespan of our timestamping algorithm is at most the duration of a sequential execution, i.e. the sum of the lengths of all transactions: $\sum_{i=0}^{n} t_{T_i}$. The competitive ratio is:

  $$\frac{\sum_{i=0}^{n} t_{T_i}}{\sum_{i=0}^{n} \frac{c \cdot t_{T_i}}{s}} = \frac{s}{c} = O(s).$$

  Remarks:

  – Unfortunately, in most relevant cases the number of resources is larger than the number of cores, i.e., $s > n$. Thus, our timestamping algorithm only guarantees sequential execution, whereas the optimal might execute all transactions in parallel.

Are there contention managers that guarantee more than sequential execution, if a lot of parallelism is possible? If we have a powerful adversary, that can change the required resources after an abort, the analysis is tight. Though we restrict to deterministic algorithms here, the theorem also holds for randomized contention managers.

Theorem 25.6. *Suppose n transactions start at the same time and the adversary is allowed to alter the resource requirement of any transaction (only) after an abort, then the competitive ratio of any deterministic contention manager is $\Omega(n)$.*

Proof. Assume we have n resources. Suppose all transactions consist of two operations, such that conflicts arise, which force the contention manager to

abort one of the two transactions T_{2i-1}, T_{2i} for every $i < n/2$. More precisely, transaction T_{2i-1} writes to resource R_{2i-1} and to R_{2i} afterwards. Transaction T_{2i} writes to resource R_{2i} and to R_{2i-1} afterwards. Clearly, any contention manager has to abort $n/2$ transactions. Now the adversary tells each transaction which did not finish to adjust its resource requirements and write to resource R_0 as their first operation. Thus, for any deterministic contention manager the $n/2$ aborted transactions must execute sequentially and the makespan of the algorithm becomes $\Omega(n)$.

The optimal strategy first schedules all transactions that were aborted and in turn aborts the others. Since the now aborted transactions do not change their resource requirements, they can be scheduled in parallel. Hence the optimal makespan is 4, yielding a competitive ratio of $\Omega(n)$. □

Remarks:

- The prove can be generalized to show that the ratio is $\Omega(s)$ if s resources are present, matching the previous upper bound.

- But what if the adversary is not so powerful, i.e., a transaction has a fixed set of needed resources?

  The analysis of algorithm timestamp is still tight. Consider the dining philosophers problem: Suppose all transactions have length n and transaction i requires its first resource R_i at time 1 and its second R_{i+1} (except T_n, which only needs R_n) at time $n - i$. Thus, each transaction T_i potentially conflicts with transaction T_{i-1} and transaction T_{i+1}. Let transaction i have the i^{th} oldest timestamp. At time $n-i$ transaction $i+1$ with $i \geq 1$ will get aborted by transaction i and only transaction 1 will commit at time n. After every abort transaction i restarts 1 time unit before transaction $i-1$. Since transaction $i-1$ acquires its second resource $i-1$ time units before its termination, transaction $i-1$ will abort transaction i at least $i-1$ times. After $i-1$ aborts transaction i may commit. The total time until the algorithm is done is bounded by the time transaction n stays in the system, i.e., at least $\sum_{i=1}^{n}(n-i) = \Omega(n^2)$. An optimal schedule requires only $\mathcal{O}(n)$ time: First schedule all transactions with even indices, then the ones with odd indices.

- Let us try to approximate the work done differently. The transaction, which has performed more work should win the conflict. A transaction counts the number of accessed resources, starting from 0 after every restart. The transaction which has acquired more resources, wins the conflict. In case both have accessed the same number of resources, the transaction having locked the resource may proceed and the other has to wait.

  Analysis: Deadlock possible: Transaction A and B start concurrently. Transaction A writes to R_1 as its first operation and to R_2 as its second operation. Transaction B writes to the resources in opposite order.

Randomized contention managers

Though the lower bound of the previous section (Theorem 25.6) is valid for both deterministic and randomized schemes, let us look at a randomized approach:

Each transaction chooses a random priority in $[1, n]$. In case of a conflict, the transaction with lower priority gets aborted. (If both conflicting transactions have the same priority, both abort.)

Additionally, if a transaction A was aborted by transaction B, it waits until transaction B committed or aborted, then transaction A restarts and draws a new priority.

Analysis: Assume the adversary cannot change the resource requirements, otherwise we cannot show more than a competitive ratio of n, see Theorem 25.6. Assume if two transactions A and B (potentially) conflict (i.e., write to the same resource), then they require the resource for at least a fraction c of their running time. We assume a transaction T potentially conflicts with d_T other transactions. Therefore, if a transaction has highest priority among these d_T transactions, it will abort all others and commit successfully. The chance that for a transaction T a conflicting transaction chooses the same random number is $(1 - 1/n)^{d_T} > (1 - 1/n)^n \approx 1/e$. The chance that a transaction chooses the largest random number and no other transaction chose this number is thus at least $1/d_T \cdot 1/e$. Thus, for any constant $c \geq 1$, after choosing $e \cdot d_T \cdot c \cdot \ln n$ random numbers the chance that transaction T has commited successfully is

$$1 - \left(1 - \frac{1}{e \cdot d_T}\right)^{e \cdot d_T \cdot c \cdot \ln n} \approx 1 - e^{-c \ln n} = 1 - \frac{1}{n^c}.$$

Assuming that the longest transaction takes time t_{max}, within that time a transaction either commits or aborts and chooses a new random number. The time to choose $e \cdot t_{max} \cdot c \cdot \ln n$ numbers is thus at most $e \cdot t_{max} \cdot d_T \cdot c \cdot \ln n = O(t_{max} \cdot d_T \cdot \ln n)$. Therefore, with high probability each transaction makes progress within a finite amount of time, i.e., our algorithm ensures wait freedom. Furthermore, the competitive ratio of our randomized contention manger for the previously considered dining philosophers problem is w.h.p. only $\mathcal{O}(\ln n)$, since any transaction only conflicts with two other transactions.

Chapter Notes

See [GWK09, Att08, SW09, HSW10].

Bibliography

[Att08] Hagit Attiya. Needed: foundations for transactional memory. *SIGACT News*, 39(1):59–61, 2008.

[GWK09] Vijay K. Garg, Roger Wattenhofer, and Kishore Kothapalli, editors. *Distributed Computing and Networking, 10th International Conference, ICDCN 2009, Hyderabad, India, January 3-6, 2009. Proceedings*, volume 5408 of *Lecture Notes in Computer Science*. Springer, 2009.

[HSW10] David Hasenfratz, Johannes Schneider, and Roger Wattenhofer. Transactional Memory: How to Perform Load Adaption in a Simple And Distributed Manner. In *The 2010 International Conference on High Performance Computing & Simulation (HPCS), Caen, France*, June 2010.

[SW09] Johannes Schneider and Roger Wattenhofer. Bounds On Contention Management Algorithms. In *20th International Symposium on Algorithms and Computation (ISAAC), Honolulu, USA*, 2009.

Chapter 26

Dominating Set

In this chapter we present another randomized algorithm that demonstrates the power of randomization to break symmetries. We study the problem of finding a small dominating set of the network graph. As it is the case for the MIS, an efficient dominating set algorithm can be used as a basic building block to solve a number of problems in distributed computing. For example, whenever we need to partition the network into a small number of local clusters, the computation of a small dominating set usually occurs in some way. A particularly important application of dominating sets is the construction of an efficient backbone for routing.

Definition 26.1 (Dominating Set)**.** *Given an undirected graph $G = (V, E)$, a* dominating set *is a subset $S \subseteq V$ of its nodes such that for all nodes $v \in V$, either $v \in S$ or a neighbor u of v is in S.*

Remarks:

- It is well-known that computing a dominating set of minimal size is NP-hard. We therefore look for approximation algorithms, that is, algorithms which produce solutions which are optimal up to a certain factor.
- Note that every MIS (cf. Chapter 7) is a dominating set. In general, the size of every MIS can however be larger than the size of an optimal minimum dominating set by a factor of $\Omega(n)$. As an example, connect the centers of two stars by an edge. Every MIS contains all the leaves of at least one of the two stars whereas there is a dominating set of size 2.

All the dominating set algorithms that we study throughout this chapter operate in the following way. We start with $S = \emptyset$ and add nodes to S until S is a dominating set. To simplify presentation, we color nodes according to their state during the execution of an algorithm. We call nodes in S *black*, nodes which are covered (neighbors of nodes in S) *gray*, and all uncovered nodes *white*. By $W(v)$, we denote the set of white nodes among the direct neighbors of v, including v itself. We call $w(v) = |W(v)|$ the *span* of v.

26.1 Sequential Greedy Algorithm

Intuitively, to end up with a small dominating set S, nodes in S need to cover as many neighbors as possible. It is therefore natural to add nodes v with a large span $w(v)$ to S. This idea leads to a simple greedy algorithm:

Algorithm 26.2 Greedy Algorithm

```
1: S := ∅;
2: while there are white nodes do
3:     v := {v | w(v) = max_u{w(u)}};
4:     S := S ∪ v;
5: end while
```

Theorem 26.3. *The* Greedy Algorithm *computes a $\ln \Delta$-approximation, that is, for the computed dominating set S and an optimal dominating set S^*, we have*

$$\frac{|S|}{|S^*|} \leq \ln \Delta.$$

Proof. Each time, we choose a new node of the dominating set (each greedy step), we have cost 1. Instead of letting this node pay the whole cost, we distribute the cost equally among all newly covered nodes. Assume that node v, chosen in line 3 of the algorithm, is white itself and that its white neighbors are v_1, v_2, v_3, and v_4. In this case each of the 5 nodes v and $v_1, \ldots, v_4$ get charged $1/5$. If v is chosen as a gray node, only the nodes $v_1, \ldots, v_4$ get charged (they all get $1/4$).

Now, assume that we know an optimal dominating set S^*. By the definition of dominating sets, to each node which is not in S^*, we can assign a neighbor from S^*. By assigning each node to exactly one neighboring node of S^*, the graph is decomposed into stars, each having a dominator (node in S^*) as center and non-dominators as leaves. Clearly, the cost of an optimal dominating set is 1 for each such star. In the following, we show that the amortized cost (distributed costs) of the greedy algorithm is at most $\ln \Delta + 2$ for each star. This suffices to prove the theorem.

Consider a single star with center $v^* \in S^*$ before choosing a new node u in the greedy algorithm. The number of nodes that become dominated when adding u to the dominating set is $w(u)$. Thus, if some white node v in the star of v^* becomes gray or black, it gets charged $1/w(u)$. By the greedy condition, u is a node with maximal span and therefore $w(u) \geq w(v^*)$. Thus, v is charged at most $1/w(v^*)$. After becoming gray, nodes do not get charged any more. Therefore first node that is covered in the star of v^* gets charged at most $1/(d(v^*)+1)$. Because $w(v^*) \geq d(v^*)$ when the second node is covered, the second node gets charged at most $1/d(v^*)$. In general, the i^{th} node that is covered in the star of v^* gets charged at most $1/(d(v^*)+i-2)$. Thus, the total amortized cost in the star of v^* is at most

$$\frac{1}{d(v^*)+1} + \frac{1}{d(v^*)} + \cdots + \frac{1}{2} + \frac{1}{1} = \mathrm{H}(d(v^*)+1) \leq \mathrm{H}(\Delta+1) < \ln(\Delta) + 2$$

where Δ is the maximal degree of G and where $H(n) = \sum_{i-1}^{n} 1/i$ is the n^{th} number. □

Remarks:

- One can show that unless NP $\subseteq$ DTIME($n^{O(\log\log n)}$), no polynomial-time algorithm can approximate the minimum dominating set problem better than $(1-o(1))\cdot\ln\Delta$. Thus, unless P $\approx$ NP, the approximation ratio of the simple greedy algorithm is optimal (up to lower order terms).

26.2 Distributed Greedy Algorithm

For a distributed algorithm, we use the following observation. The span of a node can only be reduced if any of the nodes at distance at most 2 is included in the dominating set. Therefore, if the span of node v is greater than the span of any other node at distance at most 2 from v, the greedy algorithm chooses v before any of the nodes at distance at most 2. This leads to a very simple distributed version of the greedy algorithm. Every node v executes the following algorithm.

Algorithm 26.4 Distributed Greedy Algorithm (at node v):

```
1: while v has white neighbors do
2:    compute span w(v);
3:    send w(v) to nodes at distance at most 2;
4:    if w(v) largest within distance 2 (ties are broken by IDs) then
5:       join dominating set
6:    end if
7: end while
```

Theorem 26.5. *Algorithm 26.4 computes a dominating set of size at most* $\ln\Delta+2$ *times the size of an optimal dominating set in* $\mathcal{O}(n)$ *rounds.*

Proof. The approximation quality follows directly from the above observation and the analysis of the *greedy algorithm*. The time complexity is at most linear because in every iteration of the while loop, at least one node is added to the dominating set and because one iteration of the while loop can be implemented in a constant number of rounds. □

The approximation ratio of the above distributed algorithm is best possible (unless P $\approx$ NP or unless we allow local computations to be exponential). However, the time complexity is very bad. In fact, there really are graphs on which in each iteration of the while loop, only one node is added to the dominating set. As an example, consider a graph as in Figure 26.6. An optimal dominating set consists of all nodes on the center axis. The *distributed greedy algorithm* computes an optimal dominating set, however, the nodes are chosen sequentially from left to right. Hence, the running time of the algorithm on the graph of Figure 26.6 is $\Omega(\sqrt{n})$. Below, we will see that there are graphs on which Algorithm 26.4 even needs $\Omega(n)$ rounds.

The problem of the graph of Figure 26.6 is that there is a long path of descending degrees (spans). Every node has to wait for the neighbor to the left. Therefore, we want to change the algorithm in such a way that there

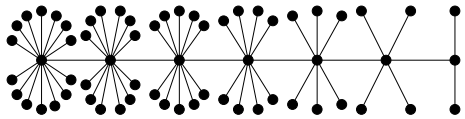


Figure 26.6: Distributed greedy algorithm: Bad example

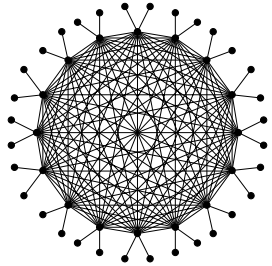


Figure 26.7: Distributed greedy algorithm with rounded spans: Bad example

are no long paths of descending spans. Allowing for an additional factor 2 in the approximation ratio, we can round all spans to the next power of 2 and let the greedy algorithm take a node with a maximal rounded span. In this case, a path of strictly descending rounded spans has at most length $\log n$. For the distributed version, this means that nodes whose rounded span is maximal within distance 2 are added to the dominating set. Ties are again broken by unique node IDs. If node IDs are chosen at random, the time complexity for the graph of Figure 26.6 is reduced from $\Omega(\sqrt{n})$ to $\mathcal{O}(\log n)$.

Unfortunately, there still is a problem remaining. To see this, we consider Figure 26.7. The graph of Figure 26.7 consists of a clique with $n/3$ nodes and two leaves per node of the clique. An optimal dominating set consists of all the $n/3$ nodes of the clique. Because they all have distance 1 from each other, the described distributed algorithm only selects one in each while iteration (the one with the largest ID). Note that as soon as one of the nodes is in the dominating set, the span of all remaining nodes of the clique is 2. They do not have common neighbors and therefore there is no reason not to choose all of them in parallel. However, the time complexity of the simple algorithm is $\Omega(n)$. In order to improve this example, we need an algorithm that can choose many nodes simultaneously as long as these nodes do not interfere too much, even if they are neighbors. In Algorithm 26.8, $N(v)$ denotes the set of neighbors of v (including v itself) and $N_2(v)=\bigcup_{u\in N(V)}N(u)$ are the nodes at distance at most 2 of v. As before, $W(v)=\{u\in N(v): u \text{ is white}\}$ and $w(v)=|W(v)|$. It is clear that if Algorithm 26.8 terminates, it computes a valid dominating set. We will now show that the computed dominating set is small and that the algorithm terminates quickly.

Theorem 26.9. *Algorithm 26.8 computes a dominating set of size at most*

Algorithm 26.8 Fast Distributed Dominating Set Algorithm (at node v):

1: $W(v) := N(v)$; $w(v) := |W(v)|$;
2: **while** $W(v) \neq \emptyset$ **do**
3: $\quad \tilde{w}(v) := 2^{\lfloor \log_2 w(v) \rfloor}$; // round down to next power of 2
4: $\quad \hat{w}(v) := \max_{u \in N_2(v)} \tilde{w}(u)$;
5: $\quad$ **if** $\tilde{w}(v) = \hat{w}(v)$ **then** $v.active := \textbf{true}$ **else** $v.active := \textbf{false}$ **end if**;
6: $\quad$ compute support $s(v) := |\{u \in N(v) : u.active = \textbf{true}\}|$;
7: $\quad \hat{s}(v) := \max_{u \in W(v)} s(u)$;
8: $\quad v.candidate := \textbf{false}$;
9: $\quad$ **if** $v.active$ **then**
10: $\quad\quad v.candidate := \textbf{true}$ with probability $1/\hat{s}(v)$
11: $\quad$ **end if**;
12: $\quad$ compute $c(v) := |\{u \in W(v) : u.candidate = \textbf{true}\}|$;
13: $\quad$ **if** $v.candidate$ **and** $\sum_{u \in W(v)} c(u) \leq 3w(v)$ **then**
14: $\quad\quad$ node v joins dominating set
15: $\quad$ **end if**
16: $\quad W(v) := \{u \in N(v) : u \text{ is white}\}$; $w(v) := |W(v)|$;
17: **end while**

$(6 \cdot \ln \Delta + 12) \cdot |S^*|$, where S^* is an optimal dominating set.

Proof. The proof is a bit more involved but analogous to the analysis of the approximation ratio of the greedy algorithm. Every time, we add a new node v to the dominating set, we distribute the cost among v (if it is still white) and its white neighbors. Consider an optimal dominating set S^*. As in the analysis of the greedy algorithm, we partition the graph into stars by assigning every node u not in S^* to a neighbor v^* in S^*. We want to show that the total distributed cost in the star of every $v^* \in S^*$ is at most $6H(\Delta + 1)$.

Consider a node v that is added to the dominating set by Algorithm 26.8. Let $W(v)$ be the set of white nodes in $N(v)$ when v becomes a dominator. For a node $u \in W(v)$ let $c(u)$ be the number of candidate nodes in $N(u)$. We define $C(v) = \sum_{u \in W(v)} c(u)$. Observe that $C(v) \leq 3w(v)$ because otherwise v would not join the dominating set in line 15. When adding v to the dominating set, every newly covered node $u \in W(v)$ is charged $3/(c(u)w(v))$. This compensates the cost 1 for adding v to the dominating set because

$$\sum_{u \in W(v)} \frac{3}{c(u)w(v)} \;\geq\; w(v) \cdot \frac{3}{w(v) \cdot \sum_{u \in W(v)} c(u)/w(v)} \;=\; \frac{3}{C(v)/w(v)} \;\geq\; 1.$$

The first inequality follows because it can be shown that for $\alpha_i > 0$, $\sum_{i=1}^{k} 1/\alpha_i \geq k/\bar{\alpha}$ where $\bar{\alpha} = \sum_{i=1}^{k} \alpha_i/k$.

Now consider a node $v^* \in S^*$ and assume that a white node $u \in W(v^*)$ turns gray or black in iteration t of the while loop. We have seen that u is charged $3/(c(u)w(v))$ for every node $v \in N(u)$ that joins the dominating set in iteration t. Since a node can only join the dominating set if its span is largest up to a factor of two within two hops, we have $w(v) \geq w(v^*)/2$ for every node $v \in N(u)$ that joins the dominating set in iteration t. Because there are at most $c(u)$ such nodes, the charge of u is at most $6/w(v^*)$. Analogously to the sequential greedy algorithm, we now get that the total cost in the star of a node $v^* \in S^*$ is at most

$$\sum_{i=1}^{|N(v^*)|} \frac{6}{i} \;\leq\; 6 \cdot H(|N(v^*)|) \;\leq\; 6 \cdot H(\Delta + 1) \;=\; 6 \cdot \ln \Delta + 12.$$

□

To bound the time complexity of the algorithm, we first need to prove the following lemma.

Lemma 26.10. *Consider an iteration of the while loop. Assume that a node u is white and that $2s(u) \geq \max_{v \in \mathcal{C}(u)} \hat{s}(v)$ where $\mathcal{C}(u) = \{v \in N(u) : v.candidate = \textbf{true}\}$. Then, the probability that u becomes dominated (turns gray or black) in the considered while loop iteration is larger than $1/9$.*

Proof. Let $D(u)$ be the event that u becomes dominated in the considered while loop iteration, i.e., $D(u)$ is the event that u changes its color from white to gray or black. Thus, we need to prove that $\Pr\big[D(u)\big] > 1/9$. We can write this probability as

$$\Pr\big[D(u)\big] = \Pr\big[c(u) > 0\big] \cdot \Pr\big[D(u) | c(u) > 0\big] + \Pr\big[c(u) = 0\big] \cdot \underbrace{\Pr\big[D(u) | c(u) = 0\big]}_{=0}.$$

It is therefore sufficient to lower bound the probabilities $\Pr\big[c(u) > 0\big]$ and $\Pr\big[D(u) | c(u) > 0\big]$. We have $2s(u) \geq \max_{v \in \mathcal{C}(u)} \hat{s}(v)$. Therefore, in line 10, each of the $s(u)$ active nodes $v \in N(u)$ becomes a candidate node with probability $1/\hat{s}(v) \geq 1/(2s(u))$. The probability that at least one of the $s(u)$ active nodes in $N(u)$ becomes a candidate therefore is

$$\Pr[c(u) > 0] \;>\; 1 - \left(1 - \frac{1}{2s(u)}\right)^{s(u)} \;>\; 1 - \frac{1}{\sqrt{e}} > \frac{1}{3}.$$

We used that for $x \geq 1$, $(1 - 1/x)^x < 1/e$. We next want to bound the probability $\Pr\big[D(u) | c(u) > 0\big]$ that at least one of the $c(u)$ candidates in $N(u)$ joins the dominating set. We have

$$\Pr\big[D(u) | c(u) > 0\big] \;\geq\; \min_{v \in N(u)} \Pr\big[v \text{ joins dominating set} | v.candidate = \textbf{true}\big].$$

Consider some node v and let $C(v) = \sum_{v' \in W(v)} c(v')$. If v is a candidate, it joins the dominating set if $C(v) \leq 3w(v)$. We are thus interested in the probability $\Pr\big[C(v) \leq 3w(v) \big| v.candidate = \textbf{true}\big]$. Assume that v is a candidate. Let $c'(v') = c(v') - 1$ be the number of candidates in $N(v') \setminus \{v\}$. For a node $v' \in W(v)$, $c'(v')$ is upper bounded by a binomial random variable $\mathrm{Bin}\big(s(v') - 1, 1/s(v')\big)$ with expectation $(s(v') - 1)/s(v')$. We therefore have

$$\mathbb{E}\big[c(v') | v.candidate = \textbf{true}\big] \;=\; 1 + \mathbb{E}\big[c'(v')\big] = 1 + \frac{s(v') - 1}{s(v')} < 2.$$

By linearity of expectation, we hence obtain

$$\begin{aligned} \mathbb{E}\big[C(v) | v.candidate = \textbf{true}\big] \;&=\; \sum_{v' \in W(v)} \mathbb{E}\big[c(v') | v.candidate = \textbf{true}\big] \\ &<\; 2w(v). \end{aligned}$$

We can now use Markov's inequality to bound the probability that $C(v)$ becomes too large:

$$\Pr\big[C(v) > 3w(v)\big|v.candidate = \textbf{true}\big] < \frac{2}{3}.$$

Combining everything, we get

$$\begin{aligned}\Pr\big[&v \text{ joins dom. set}|v.candidate = \textbf{true}\big]\\ &= \Pr\big[C(v) \leq 3w(v)\big|v.candidate = \textbf{true}\big] \;>\; \frac{1}{3}\end{aligned}$$

and hence

$$\Pr\big[D(u)\big] = \Pr\big[c(u) > 0\big] \cdot \Pr\big[D(u)|c(u) > 0\big] > \frac{1}{3} \cdot \frac{1}{3} = \frac{1}{9}.$$

□

Theorem 26.11. *In expectation, Algorithm 26.8 terminates in $\mathcal{O}(\log^2\Delta \cdot \log n)$ rounds.*

Proof. First observe that every iteration of the while loop can be executed in a constant number of rounds. Consider the state after t iterations of the while loop. Let $\tilde{w}_{\max}(t) = \max_{v\in V} \tilde{w}(v)$ be the maximal span rounded down to the next power of 2 after t iterations. Further, let $s_{\max}(t)$ be the maximal support $s(v)$ of any node v for which there is a node $u \in N(v)$ with $w(u) \geq \tilde{w}_{\max}(t)$ after t while loop iterations. Observe that all nodes v with $w(v) \geq \tilde{w}_{\max}(t)$ are active in iteration $t+1$ and that as long as the maximal rounded span $\tilde{w}_{\max}(t)$ does not change, $s_{\max}(t)$ can only get smaller with increasing t. Consider the pair $(\tilde{w}_{\max}, s_{\max})$ and define a relation $\prec$ such that $(w', s') \prec (w, s)$ iff $w' < w$ or $w = w'$ and $s' \leq s/2$. From the above observations, it follows that

$$(\tilde{w}_{\max}(t), s_{\max}(t)) \prec (\tilde{w}_{\max}(t'), s_{\max}(t')) \implies t > t'. \tag{26.11.1}$$

For a given time t, let $T(t)$ be the first time for which

$$(\tilde{w}_{\max}(T(t)), s_{\max}(T(t))) \prec (\tilde{w}_{\max}(t), s_{\max}(t)).$$

We first want to show that for all t,

$$\mathbb{E}[T(t) - t] \;=\; O(\log n). \tag{26.11.2}$$

Let us look at the state after t while loop iterations. By Lemma 26.10, every white node u with support $s(u) \geq s_{\max}(t)/2$ will be dominated after the following while loop iteration with probability larger than $1/9$. Consider a node u that satisfies the following three conditions:

(1) u is white

(2) there is a node $v \in N(u) : w(v) \geq \tilde{w}_{\max}(t)$

(3) $s(u) \geq s_{\max}(t)/2$.

As long as u satisfies all three conditions, the probability that u becomes dominated is larger than $1/9$ in every while loop iteration. Hence, after $t+\tau$ iterations (from the beginning), u is dominated or does not satisfy (2) or (3) with probability larger than $(8/9)^\tau$. Choosing $\tau = \log_{9/8}(2n)$, this probability becomes $1/(2n)$. There are at most n nodes u satisfying Conditions $(1)-(3)$. Therefore, applying union bound, we obtain that with probability more than $1/2$, there is no white node u satisfying Conditions $(1)-(3)$ at time $t + \log_{9/8}(2n)$. Equivalently, with probability more than $1/2$, $T(t) \leq t + \log_{9/8}(2n)$. Analogously, we obtain that with probability more than $1/2^k$, $T(t) \leq t + k\log_{9/8}(2n)$. We then have

$$\begin{aligned}\mathbb{E}[T(t) - t] \;&=\; \sum_{\tau=1}^{\infty} \Pr[T(t) - t = \tau] \cdot \tau\\ &\leq\; \sum_{k=1}^{\infty} \left(\frac{1}{2^k} - \frac{1}{2^{k+1}}\right) \cdot k\log_{9/8}(2n) \;=\; \log_{9/8}(2n)\end{aligned}$$

and thus Equation (26.11.2) holds.

Let $t_0 = 0$ and $t_i = T(t_{i-1})$ for $i = 1, \ldots, k$. where $t_k = \min_t \tilde{w}_{\max}(t) = 0$. Because $\tilde{w}_{\max}(t) = 0$ implies that $w(v) = 0$ for all $v \in V$ and that we therefore have computed a dominating set, by Equations (26.11.1) and (26.11.2) (and linearity of expectation), the expected number of rounds until Algorithm 26.8 terminates is $\mathcal{O}(k \cdot \log n)$. Since $\tilde{w}_{\max}(t)$ can only have $\lfloor \log \Delta \rfloor$ different values and because for a fixed value of $\tilde{w}_{\max}(t)$, the number of times $s_{\max}(t)$ can be decreased by a factor of 2 is at most $\log \Delta$ times, we have $k \leq \log^2\Delta$. □

Remarks:

- It is not hard to show that Algorithm 26.8 even terminates in $\mathcal{O}(\log^2\Delta \cdot \log n)$ rounds with probability $1 - 1/n^c$ for an arbitrary constant c.
- Using the median of the supports of the neighbors instead of the maximum in line 8 results in an algorithm with time complexity $\mathcal{O}(\log\Delta \cdot \log n)$. With another algorithm, this can even be slightly improved to $\mathcal{O}(\log^2\Delta)$.
- One can show that $\Omega(\log\Delta/\log\log\Delta)$ rounds are necessary to obtain an $\mathcal{O}(\log\Delta)$-approximation.
- It is not known whether there is a fast deterministic approximation algorithm. This is an interesting and important open problem. The best deterministic algorithm known to achieve an $\mathcal{O}(\log\Delta)$-approximation has time complexity $2^{O(\sqrt{\log n})}$.

Chapter Notes

See [JRS02, KW05].

Bibliography

[JRS02] Lujun Jia, Rajmohan Rajaraman, and Torsten Suel. An efficient distributed algorithm for constructing small dominating sets. *Distributed Computing*, 15(4):193–205, 2002.

[KW05] Fabian Kuhn and Roger Wattenhofer. Constant-Time Distributed Dominating Set Approximation. In *Springer Journal for Distributed Computing, Volume 17, Number 4*, May 2005.

Chapter 27

Routing

27.1 Array

(Routing is important for any distributed system. This chapter is only an introduction into routing; we will see other facets of routing in a next chapter.)

Definition 27.1 (Routing)**.** *We are given a graph and a set of routing requests, each defined by a source and a destination node.*

Definition 27.2 (One-to-one, Permutation)**.** *In a one-to-one routing problem, each node is the source of at most one packet and each node is the destination of at most one packet. In a permutation routing problem, each node is the source of exactly one packet and each node is the destination of exactly one packet.*

Remarks:

- Permutation routing is a special case of one-to-one routing.

Definition 27.3 (Store and Forward Routing)**.** *The network is synchronous. In each step, at most two packets (one in each direction) can be sent over each link.*

Remarks:

- If two packets want to follow the same link, then one is queued (stored) at the sending node. This is known as contention.

Algorithm 27.4 Greedy on Array

An array is a linked list of n nodes; that is, node i is connected with nodes $i-1$ and $i+1$, for $i = 2, \ldots, n-1$. With the greedy algorithm, each node injects its packet at time 0. At each step, each packet that still needs to move rightward or leftward does so.

Theorem 27.5 (Analysis)**.** *The greedy algorithm terminates in $n-1$ steps.*

Proof. By induction two packets will never contend for the same link. Then each packet arrives at its destination in d steps, where d is the distance between source and destination. □

Remarks:

- Unfortunately, only the array (or the ring) allows such a simple contention-free analysis. Already in a tree (with nodes of degree 3 or more) there might be two packets arriving at the same step at the same node, both want to leave on the same link, and one needs to be queued. In a "Mercedes-Benz" graph $\Omega(n)$ packets might need to be queued. We will study this problem in the next section.
- There are many strategies for scheduling packets contending for the same edge (e.g. "farthest goes first"); these queuing strategies have a substantial impact on the performance of the algorithm.

27.2 Mesh

Algorithm 27.6 Greedy on Mesh

A mesh (a.k.a. grid, matrix) is a two-dimensional array with m columns and m rows ($n = m^2$). Packets are routed to their correct column (on the row in greedy array style), and then to their correct row. The farthest packet will be given priority.

Theorem 27.7 (Analysis)**.** *In one-to-one routing, the greedy algorithm terminates in $2m-2$ steps.*

Proof. First note that packets in the first phase of the algorithm do not interfere with packets in the second phase of the algorithm. With Theorem 27.5 each packet arrives at its correct column in $m-1$ steps. (Some packets may arrive at their turning node earlier, and already start the second phase; we will not need this in the analysis.) We need the following Lemma for the second phase of the algorithm.

Lemma 27.8 (Many-to-One on Array, Lemma 1.5 in Leighton Section 1.7)**.** *We are given an array with n nodes. Each node is a destination for at most one packet (but may be the source of many). If edge contention is resolved by farthest-to-go (FTG), the algorithm terminates in $n-1$ steps.*

Leighton Section 1.7 Lemma 1.5. Leftward moving packets and rightward moving packets never interfere; so we can restrict ourselves to rightward moving packets. We name the packets with their destination node. Since the queuing strategy is FTG, packet i can only be stopped by packets $j > i$. Note that a packet i may be contending with the same packet j several times. However, packet i will either find its destination "among" the higher packets, or directly after the last of the higher packets. More formally, after k steps, packets $j, j+1, \ldots, n$ do not need links $1, \ldots, l$ anymore, with $k = n - j + l$. Proof by induction: Packet n has the highest priority: After k steps it has escaped

the first k links. Packet $n-1$ can therefore use link l in step $l+1$, and so on. Packet i not needing link i in step $k=n$ means that packet i has arrived at its destination node i in step $n-1$ or earlier. □

Lemma 27.8 completes the proof. □

Remarks:

- A $2m-2$ time bound is the best we can hope for, since the distance between the two farthest nodes in the mesh is exactly $2m-2$.
- One thing still bugs us: The greedy algorithm might need queues in the order of m. And queues are expensive! In the next section, we try to bring the queue size down!

27.3 Routing in the Mesh with Small Queues

(First we look at a slightly simpler problem.)

Definition 27.9 (Random Destination Routing)**.** *In a random destination routing problem, each node is the source of at most one packet with destination chosen uniformly at random.*

Remarks:

- Random destination routing is not one-to-one routing. In the worst case, a node can be destination for all n packets, but this case is very unlikely (with probability $1/n^{n-1}$)
- We study algorithm 27.6, but this time in the random destination model. Studying the random destination model will give us a deeper understanding of routing... and distributed computing in general!

Theorem 27.10 (Random destination analysis of algorithm 27.6)**.** *If destinations are chosen at random the maximum queue size is $\mathcal{O}(\log n/\log\log n)$ with high probability. (With high probability means with probability at least $1-O(1/n)$.)*

Proof. We can restrict ourselves to column edges because there will not be any contention at row edges. Let us consider the queue for a north-bound column edge. In each step, there might be three packets arriving (from south, east, west). Since each arriving south packet will be forwarded north (or consumed when the node is the destination), the queue size can only grow from east or west packets – packets that are "turning" at the node. Hence the queue size of a node is always bounded by the number of packets turning at the node. A packet only turns at a node u when it is originated at a node in the same row as u (there are only m nodes in the row). Packets have random destinations, so the probability to turn for each of these packets is $1/m$ only. Thus the probability P that r or more packets turn in some particular node u is at most

$$P \leq \binom{m}{r}\left(\frac{1}{m}\right)^r$$

(The factor $(1-1/m)^{m-r}$ is not present because the event "exactly r" includes the event "more than r" already.) Using

$$\binom{x}{y} < \left(\frac{xe}{y}\right)^y, \text{ for } 0 < y < x$$

we directly get

$$P < \left(\frac{me}{r}\right)^r\left(\frac{1}{m}\right)^r = \left(\frac{e}{r}\right)^r$$

Hence most queues do not grow larger than $\mathcal{O}(1)$. Also, when we choose $r := \frac{e\log n}{\log\log n}$ we can show $P = o(1/n^2)$. The probability that any of the $4n$ queues ever exceeds r is less than $1-(1-P)^{4n} = o(1/n)$. □

Remarks:

- OK. We got a bound on the queue size. Now what about time complexity?!? The same analysis as for one-to-one routing applies. The probability that a node sees "many" packets in phase 2 is small... it can be shown that the algorithm terminates in $\mathcal{O}(m)$ time with high probability.
- In fact, maximum queue sizes are likely to be a lot less than logarithmic. The reason is the following: Though $\Theta(\log n/\log\log n)$ packets might turn at some node, these turning packets are likely to be spread in time. Early arriving packets might use gaps and do not conflict with late arriving packets. With a much more elaborate method (using the so-called "wide-channel" model) one can show that there will never be more than four(!) packets in any queue (with high probability only, of course).
- Unfortunately, the above analysis only works for random destination problems. Question: Can we devise an algorithm that uses small queues only but for any one-to-one routing problem? Answer: Yes, we can! In the simplest form we can use a clever trick invented by Leslie Valiant: Instead of routing the packets directly on their row-column path, we route each packet to a randomly chosen intermediate node (on the row-column path), and from there to the destination (again on the row-column path). Valiant's trick routes all packets in $\mathcal{O}(m)$ time (with high probability) and only needs queues of size $\mathcal{O}(\log n)$. Instead of choosing a random intermediate node one can choose a random node that is more or less in the direction of the destination, solving any one-to-one routing problem in $2m+O(\log n)$ time with only constant-size queues. You don't wanna know the details...
- What about no queues at all?!?

27.4 Hot-Potato Routing

Definition 27.11 (Hot-Potato Routing)**.** *Like the store-and-forward model the hot-potato model is synchronous and at most two packets (one in each direction)*

can be sent over a link. However, contending packets cannot be stored; instead all but one contending packet must be sent over a "wrong link" (known as deflection) immediately, since the hot-potato model does not allow queuing.

Remarks:

- Don't burn your fingers with "hot-potato" packets. If you get one you better forward it directly!
- A node with degree δ receives up to δ packets at the beginning of each step – since the node has δ links, it can forward all of them, but unfortunately not all in the right direction.
- Hot-potato routing is easier to implement, especially on light-based networks, where you don't want to convert photons into electrons and then back again. There are a couple of parallel machines that use the hot-potato paradigm to simplify and speed up routing.
- How bad does hot-potato routing get (in the random or the one-to-one model)? How bad can greedy hot-potato routing (greedy: whenever there is no contention you must send a packet into the right direction) get in a worst case?

Algorithm 27.12 Greedy Hot-Potato Routing on a Mesh

Packets move greedy towards their destination (any good link is fine if there is more than one). If a packet gets deflected, it gets excited with probability p (we set $p = \Theta(1/m)$). An excited packet has higher priority. When being excited, a packet tries to reach the destination on the row-column path. If two excited packets contend, then the one that wants to exit the opposite link is given priority. If an excited packet fails to take its desired link it becomes normal again.

Theorem 27.13 (Analysis)**.** *A packet will reach its destination in $\mathcal{O}(m)$ expected time.*

Sketch, full proof in Busch et al., SODA 2000. An excited packet can only be deflected at its start node (after becoming excited), and when trying to turn. In both cases, the probability to fail is only constant since other excited packets need to be at the same node at exactly the right instant. Thus the probability that an excited packets finds to its destination is constant, and therefore a packet needs to "try" (to become excited) only constantly often. Since a packet tries every p'th time it gets deflected, in only gets deflected $\mathcal{O}(1/p) = \mathcal{O}(m)$ times in expectation. Since each time it does not get deflected, it gets closer to its destination, it will arrive at the destination in $\mathcal{O}(m)$ expected time. □

Remarks:

- It seems that at least in expectation having no memory at all does not harm the time bounds much.
- It is conjectured that one-to-one routing can be shown to have time complexity $\mathcal{O}(m)$ for this greedy hot-potato routing algorithm. However, the best known bound needs an additional logarithmic factor.

27.5 More Models

Routing comes in many flavors. We mention some of them in this section for the sake of completeness.

Store-and-forward and hot-potato routing are variants of packet-switching. In the circuit-switching model, the entire path from source to destination must be locked such that a stream of packets can be transmitted.

A packet-switching variant where more than one packet needs to be sent from source to destination in a stream is known as wormhole routing.

Static routing is when all the packets to be routed are injected at time 0. Instead, in dynamic routing, nodes may inject new packets constantly (at a certain rate). Not much is known for dynamic routing.

Instead of having a single source and a single destination for each packet as in one-to-one routing, researchers have studied many-to-one routing, where a node may be destination for many sources. The problem of many-to-one routing is that there might be congested areas in the network (areas with nodes that are destinations of many packets). Packets that can be routed around such a congested area should do that, or they increase the congestion even more. Such an algorithm was studied by Busch et al. at STOC 2000.

Also one-to-many routing (multicasting) was considered, where a source needs to send the same packet to many destinations. In one-to-many routing, packets can be duplicated whenever needed.

Nobody knows the topology of the Internet (and it is certainly not an array or a mesh!). The problem is to find short paths without storing huge routing tables at each node. There are several forms of routing (e.g. compact routing, interval routing) that study the trade-off between routing table size and quality of routing.

Also, researchers started studying the effects of mixing various queuing strategies in one network. This area of research is known as adversarial queuing theory.

And last not least there are several special networks. A mobile ad-hoc network, for example, consists of mobile nodes equipped with a wireless communication device. In such a networks nodes can only communicate when they are within transmission range. Since the network is mobile (dynamic), and since the nodes are considered to be simple, a variety of new problems arise.

Chapter Notes

See [BHW00a, BHW00b, RT92, Lei90, VB81].

Bibliography

[BHW00a] Costas Busch, Maurice Herlihy, and Roger Wattenhofer. Hard-Potato Routing. In *32nd Annual ACM Symposium on Theory of Computing (STOC), Portland, Oregon*, May 2000.

[BHW00b] Costas Busch, Maurice Herlihy, and Roger Wattenhofer. Randomized Greedy Hot-Potato Routing. In *11th Annual ACM-SIAM Sym-*

posium on Discrete Algorithms (SODA), pp. 458-466, San Francisco, California, USA, January 2000.

[Lei90] Frank Thomson Leighton. Average Case Analysis of Greedy Routing algorithms on Arrays. In *SPAA*, pages 2–10, 1990.

[RT92] Sanguthevar Rajasekaran and Thanasis Tsantilas. Optimal Routing Algorithms for Mesh-Connected Processor Arrays. *Algorithmica*, 8(1):21–38, 1992.

[VB81] Leslie G. Valiant and Gordon J. Brebner. Universal Schemes for Parallel Communication. In *STOC*, pages 263–277. ACM, 1981.

Chapter 28

Routing Strikes Back

28.1 Butterfly

Let's first assume that all the sources are on level 0, all destinations are on level d of a d-dimensional butterfly.

Algorithm 28.1 Greedy Butterfly Routing

The unique path from a source on level 0 to a destination on level d with d hops is the greedy path. In the greedy butterfly routing algorithm each packet is constrained to follow its greedy path.

Remarks:

- In the bit-reversal permutation routing problem, the destination of a packet is the bit-reversed address of the source. With $d = 3$ you can see that both source $(000, 0)$ and source $(001, 0)$ route through edge $(000, 1..2)$. Will the contention grow with higher dimension? Yes! Choose an odd d, then all the sources $(0 \dots 0 b_{(d+1)/2} \dots b_{d-1}, 0)$ will route through edge $(00..0, (d-1)/2...(d+1)/2)$. You can choose the bits b_i arbitrarily. There are $2^{(d+1)/2}$ bit combinations, which is $\sqrt{n/2}$ for $n = 2^d$ sources.
- On the good side, this contention is also a guaranteed time bound, as the following theorem shows.

Theorem 28.2 (Analysis)**.** *The greedy butterfly algorithm terminates in $\mathcal{O}(\sqrt{n})$ steps.*

Proof. For simplicity we assume that d is odd. An edge on level l (from a node on level l to a node on level $l+1$) has at most 2^l sources, and at most 2^{d-l-1} destinations. Therefore the number of paths through an edge on level l is bounded by $n_l = 2^{\min(l, d-l-1)}$. A packet can therefore be delayed at most $n_l - 1$ times on level l. Summing up over all levels, a packet is delayed at most

$$\sum_{l=0}^{d-1} n_l = \sum_{l=0}^{(d-1)/2} n_l + \sum_{l=(d+1)/2}^{d-1} n_l = \sum_{l=0}^{(d-1)/2} 2^l + \sum_{l=0}^{(d-3)/2} 2^l < 3 \cdot 2^{(d-1)/2} = O(\sqrt{n}).$$

steps. □

Remarks:

- The bit-reversed routing is therefore asymptotically a worst-case example.
- However, one that requires square-root queues. When being limited to constant queue sizes the greedy algorithm can be forced to use $\Theta(n)$ steps for some permutations.
- A routing problem where all the sources are on level 0 and all the destinations are on level d is called an end-to-end routing problem. Surprisingly, solving an arbitrary routing problem on a butterfly (or any hypercubic network) is often not harder.
- In the next section we show that there is general square-root lower bound for "greedy" algorithms for any constant-degree graph. (In other words, our optimal greedy mesh routing algorithm of Chapter 4 was only possible because the mesh has such a bad diameter...)

28.2 Oblivious Routing

Definition 28.3 (Oblivious)**.** *A routing algorithm is oblivious if the path taken by each packet depends only on source and destination of the packet (and not on other packets, or the congestion encountered).*

Theorem 28.4 (Lower Bound)**.** *Let G be a graph with n nodes and (maximum) degree d. Let A be any oblivious routing algorithm. Then there is a one-to-one routing problem for which A will need at least $\sqrt{n}/2d$ steps.*

Proof. Since A is oblivious, the path from node u to node v is $P_{u,v}$; A can be specified by n^2 paths. We must find k one-to-one paths that all use the same edge e. Then we can proof that A takes at least $k/2$ steps.

Let's look at the $n-1$ paths to destination node v. For any integer k let $S_k(v)$ be the set of edges in G where k or more of these paths pass through them. Also, let $S_k^*(v)$ be the nodes incident to $S_k(v)$. Since there are two nodes incident to each edge $|S_k^*(v)| \leq 2|S_k(v)|$. In the following we assume that $k \leq (n-1)/d$; then $v \in S_k^*(v)$, hence $|S_k^*(v)| > 0$.

We have

$$n - |S_k^*(v)| \leq (k-1)(d-1)|S_k^*(v)|$$

because every node u not in $S_k^*(v)$ is a start of a path $P_{u,v}$ that enters $S_k^*(v)$ from outside. In particular, for any node $u \notin S_k^*(v)$ there is an edge (w, w') in $P_{u,v}$ that enters $S_k^*(v)$. Since the edge $(w, w') \notin S_k(v)$, there are at most $(k-1)$ starting nodes u for edge (w, w'). Also there are at most $(d-1)$ edges adjacent to w' that are not in $S_k(v)$. We get

$$n \leq (k-1)(d-1)|S_k^*(v)| + |S_k^*(v)| \leq 2[1 + (k-1)(d-1)]|S_k(v)| \leq 2kd|S_k(v)|$$

Thus, $|S_k(v)| \geq \frac{n}{2kd}$. We set $k = \sqrt{n}/d$, and sum over all n nodes:

$$\sum_{v \in V} |S_k(v)| \geq \frac{n^2}{2kd} = \frac{n^{3/2}}{2}$$

Since there are at most $nd/2$ edges in G, this means that there is an edge e for at least

$$\frac{n^{3/2}/2}{nd/2} = \sqrt{n}/d = k$$

different values of v.

Since edge e is in at least k different paths in each set $S_k(v)$ we can construct a one-to-one permutation problem where edge e is used $\sqrt{n}/d$ times (directed: $\sqrt{n}/2d$ contention). □

Remarks:

- In fact, as many as $(\sqrt{n}/d)!$ one-to-one routing problems can be constructed with this method.
- The proof can be extended to the case where the one-to-one routing problem consists of R route requests. The lower bound is then $\Omega(\frac{R}{d\sqrt{n}})$.
- There is a node that needs to route $\Omega(\sqrt{n/d})$ packets.
- The lower bound can be extended to randomized oblivious algorithms... however, if we are allowed to use randomization, the lower bound gets much weaker. In fact, one can use Valiant's trick also in the butterfly: In a first phase, we route each packet on the greedy path to a random destination on level d, in the second phase on the same row back to level 0, and in a third phase on the greedy path to the destination. This way we can escape the bad one-to-one problems with high probability. (There are much more good one-to-one problems than bad one-to-one problems.) One can show that with this trick one can route any one-to-one end-to-end routing problem in asymptotically optimal $\mathcal{O}(\log n)$ time (with high probability).
- If a randomized algorithm fails (takes too long), simply re-run it. It will be likely to succeed then. On the other hand, if a deterministic algorithm fails in some rare instance, re-running it will not help!

28.3 Offline Routing

There are a variety of other aspects in routing. In this section we study one of them to gain further insights.

Definition 28.5 (Offline Routing)**.** *We are given a routing problem (graph and set of routing requests). An offline routing algorithm is a (not distributed) algorithm that sees the whole input (the routing problem).*

Remarks:

- Offline routing is worth being studied because the same communication pattern might appear whenever you run your (important!) (parallel) algorithm.
- In offline routing, path selection and scheduling can be studied independently.

Definition 28.6 (Path Selection)**.** *We are given a routing problem (a graph and a set of routing requests). A path selection algorithm selects a path (a route) for each request.*

Remarks:

- Path selection is efficient if the paths are "short" and do not interfere if they do not need to. Formally, this can be defined by congestion and dilation (see below).
- For some routing problems, path selection is easy. If the graph is a tree, for example, the best path between two nodes is the direct path. (Every route from a source to a destination includes at least all the links of the shortest path.)

Definition 28.7 (Dilation, Congestion)**.** *The dilation of a path selection is the length of a maximum path. The contention of an edge is the number of paths that use the edge. The congestion of a path selection is the load of a most contended edge.*

Remarks:

- A path selection should minimize congestion and dilation.
- Networking researchers have defined the "flow number" which is defined as the minimum max(congestion, dilation) over all possible path selections.
- Alternatively, congestion can be defined with directed edges, or nodes.

Definition 28.8 (Scheduling)**.** *We are given a set of source-destination paths. A scheduling algorithm specifies which messages traverse which link at which time step (for an appropriate model).*

Remarks:

- The most popular model is store-and-forward (with small queues). Other popular models have no queues at all: e.g. hot-potato routing or direct routing (where the source might delay the injection of a packet; once a packet is injected however, it will go to the destination without stop.)

Lemma 28.9 (Lower Bound)**.** *Scheduling takes at least $\Omega(C + D)$ steps, where C is the congestion and D is the dilation.*

Remarks:

- We aim for algorithms that are competitive with the lower bound. (As opposed to algorithms that finish in $\mathcal{O}(f(n))$ time; $C + D$ and n are generally not comparable.)

Theorem 28.11 (Analysis)**.** *Algorithm 28.10 terminates in $2C + D$ steps.*

Algorithm 28.10 Direct Tree Routing

We are given a tree, and a set of routing requests. (Since the graph is a tree each route request will take the direct path between source and destination; in other words, path selection is trivial.) Choose an arbitrary root r. Now sort all packets using the following order (breaking ties arbitrarily): packet p comes before packet q if the path of p reaches a node closer to r then the path of q. Now scan all packets in this order, and for each packet greedily assign its injection time to be the first that does not cause a conflict with any previous packet.

Proof. A packet p first goes up, then down the tree; thus turning at node u. Let e_u and e_d be the "up" resp. "down" edge on the path adjacent to u. The injection time of packet p is only delayed by packets that traverse e_u or e_d (if it contends with a packet q on another edge, and packet q has not a lower order, then it contends also on e_u or e_q). Since congestion is C, there are at most $2C-2$ many packets q. Thus the algorithm terminates after $2C+D$ steps. □

Remarks:

- [Leighton, Maggs, Rao 1988] have shown the existence of an $\mathcal{O}(C+D)$ schedule for any routing problem (on any graph!) using the Lovasz Local Lemma. Later the result was made more accessible by [Leighton, Maggs, Richa 1996] and others. Still it is too hard for this course...

Chapter Notes

See [BH85, LMR88, LM95, KKT91].

Bibliography

[BH85] Allan Borodin and John E. Hopcroft. Routing, Merging, and Sorting on Parallel Models of Computation. *J. Comput. Syst. Sci.*, 30(1):130–145, 1985.

[KKT91] Christos Kaklamanis, Danny Krizanc, and Thanasis Tsantilas. Tight Bounds for Oblivious Routing in the Hypercube. *Mathematical Systems Theory*, 24(4):223–232, 1991.

[LM95] T. Leighton and B. Maggs. Fast algorithms for finding O(congestion+dilation) packet routing schedules. In *Proceedings of the 28th Hawaii International Conference on System Sciences*, HICSS '95, pages 555–, Washington, DC, USA, 1995. IEEE Computer Society.

[LMR88] Frank Thomson Leighton, Bruce M. Maggs, and Satish Rao. Universal Packet Routing Algorithms (Extended Abstract). In *FOCS*, pages 256–269, 1988.