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:

if input is 1 **then**
  broadcast $\texttt{value}(1)_p$
  decide 1 and terminate
else
  decide 0 and terminate
end if

Code for all other nodes v:

for all rounds $i \in 1, \dots, f+1$ **do**
  S is the set of accepted messages $\texttt{value}(1)_u$.
  if $|S| \geq i$ and $\texttt{value}(1)_p \in S$ **then**
    broadcast $S \cup \{\texttt{value}(1)_v\}$
    decide 1 and terminate
  end if
end for
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

At time t client u wants to execute command c
Client u sends request `R` = `Request`$(c,t)_u$ to primary p
Primary p appends c to its local history, i.e., $h^p = (h^p, c)$
Primary p sends `OR` = `OrderedRequest`$(h^p, c,$ `R`$)_p$ to all replicas
Each replica r appends command c to local history $h^r = (h^r, c)$ and checks whether $h^r = h^p$
Each replica r runs command c_k and obtains result a
Each replica r sends `Response`$(a,$`OR`$)_r$ to client u
Client u collects the set S of received `Response`$(a,$`OR`$)_r$ messages
Client u checks if all histories h^r are consistent
if $|S| = 3f+1$ **then**
  Client u considers command c to be complete
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 $3f+1$ `Response`$(a,$`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)

if $2f+1 \leq |S| < 3f+1$ **then**
  Client u sends $\texttt{Commit}(S)_u$ to all replicas
  Each replica r replies with a $\texttt{LocalCommit}(S)_r$ message to u
  Client u collects at least $2f+1$ $\texttt{LocalCommit}(S)_r$ messages and considers c to be complete
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 $\texttt{R} = \texttt{Request}(c,t)_u$ to all replicas
3:   Each replica r sends a $\texttt{ConfirmRequest}(\texttt{R})_r$ 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 $\texttt{IHatePrimary}_r$ 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

All replicas continuously collect the set H of $\texttt{IHatePrimary}_r$ messages
if a replica r received $|H| > f$ messages or a valid `ViewChange` message **then**
  Replica r broadcasts $\texttt{ViewChange}(H^r,h^r,S_l^r)_r$
  Replica r stops participating in the current view
  Replica r switches to the next primary "$p = p+1$"
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

The new primary p collects the set C of $\texttt{ViewChange}(H^r,h^r,S_l^r)_r$ messages
if new primary p collected $|C| \geq 2f+1$ messages **then**
  New primary p sends $\texttt{NewView}(C)_p$ to all replicas
end if

if a replica r received a $\texttt{NewView}(C)_p$ message **then**
  Replica r recovers new history $h_{\texttt{new}}$ as shown in Algorithm 18.20
  Replica r broadcasts $\texttt{ViewConfirm}(h_{\texttt{new}})_r$ message to all replicas
end if

if a replica r received $2f+1$ $\texttt{ViewConfirm}(h_{\texttt{new}})_r$ messages **then**
  Replica r accepts $h^r = h_{\texttt{new}}$ as the history of the new view
  Replica r starts participating in the new view
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?

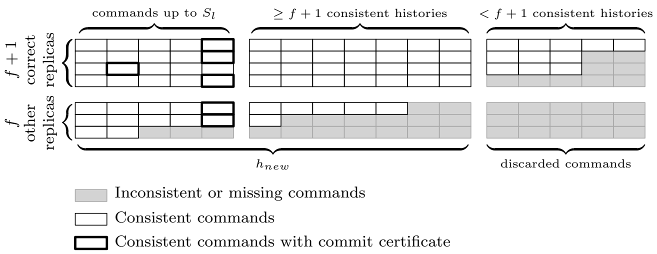

Figure 18.19: The structure of the data reported by different replicas in C. Commands up to the last commit certificate S_l were completed in either Algorithm 18.8 or Algorithm 18.9. After the last commit certificate S_l there may be commands that completed at a correct client in Algorithm 18.8. Algorithm 18.20 shows how the new history h_{new} is recovered such that no complete commands are lost.

Algorithm 18.20 Zyzzyva: History Recovery

C = set of $2f+1$ $\texttt{ViewChange}(H^r, h^r, S^r)_r$ messages in $\texttt{NewView}(C)_p$
R = set of replicas included in C
S_l = most recent commit certificate S_l^r reported in C
$h_{\texttt{new}}$ = history h_l contained in S_l
$k = l+1$, next sequence number
while command c_k exists in C **do**
  if c_k is reported by at least $f+1$ replicas in R **then**
    Remove replicas from R that do not support c_k
    $h_{\texttt{new}} = (h_{\texttt{new}}, c_k)$
  end if
  $k = k+1$
end while
return $h_{\texttt{new}}$

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.