IKS Information and Communication Systems Research Group

ETH Eidgenössische Technische Hochschule Zürich Swiss Federal Institute of Technology Zurich

Remote Procedure Call (RPC)

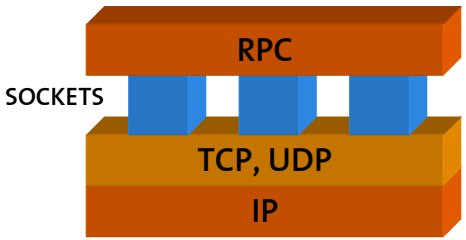
Cesare Pautasso (Gustavo Alonso)
Computer Science Department
Swiss Federal Institute of Technology (ETHZ)
pautasso@inf.ethz.ch
http://www.iks.inf.ethz.ch/

Contents – RPC

- Distributed design
  - Computer networks and communication at a high level
  - Basics of Client/Server architectures
    - programming concept
    - interoperability
    - binding to services
    - delivery guarantees
- Putting all together: RPC
  - programming languages
  - binding
  - interface definition language
  - programming RPC
- RPC in the context of large information systems (DCE, TP-Monitors)

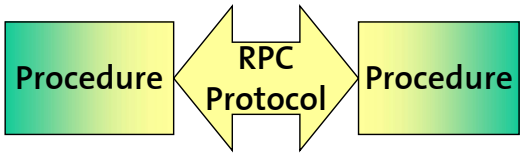
IP, TCP, UDP and RPC

- The most accepted standard for network communication is IP (Internet Protocol) which provides unreliable delivery of single packets to one-hop distant hosts
- IP was designed to be hidden behind other software layers:
  - TCP (Transport Control Protocol) implements connected, reliable message exchange
  - UDP (User Datagram Protocol) implements unreliable datagram based message exchanges
- TCP/IP and UDP/IP are visible to applications through sockets. The purpose of the socket interface was to provide a UNIX-like file abstraction
- Yet sockets are quite low level for many applications, thus, RPC (Remote Procedure Call) appeared as a way to
  - hide communication details behind a procedural call
  - bridge heterogeneous environments
- RPC is the standard for distributed (client-server) computing



Sockets vs. Remote Procedures

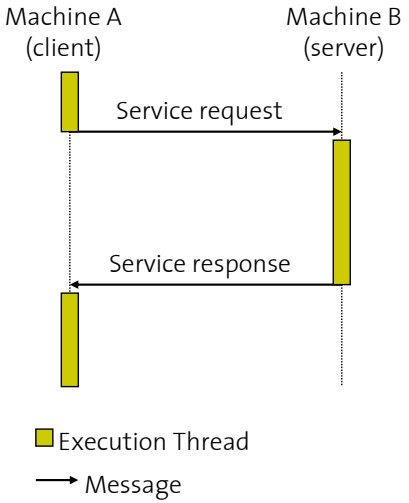
- Two alternatives for the design of distributed programs.
- **Bottom-Up**
1. First design network protocol
2. Build program that follows the protocol using sockets
- **Top-Down**
1. Design program first
2. Partition the program in different modules
3. Describe the set of procedures that make up the *module interface*
4. Place modules on different network hosts
5. Add the network protocol to make the procedures communicate



What is the advantage?
Using an *RPC Compiler* the communication protocol is generated automatically from the interface description

The basics of client/server

- Imagine we have a program (a server) that implements certain services. Imagine we have other programs (clients) that would like to invoke those services.
- To make the problem more interesting, assume as well that:
  - client and server can reside on different computers and run on different operating systems
  - the only form of communication is by sending messages (no shared memory, no shared disks, etc.)
  - some minimal guarantees are to be provided (handling of failures, call semantics, etc.)
  - we want a generic solution and not a one time hack
- Ideally, we want he programs to behave like this (sounds simple?, well, this idea is only 20 years old):



Problems to solve

1. How to make the service invocation part of the language in a more or less transparent manner.
   - Don't forget this important aspect: whatever you design, others will have to program and use
2. How to exchange data between machines that might use different representations for different data types. This involves two aspects:
   - data type formats (e.g., byte orders in different architectures)
   - data structures (need to be flattened and the reconstructed)
3. How to find the service one actually wants among a potentially large collection of services and servers.
   - The goal is that the client does not necessarily need to know where the server resides or even which server provides the service.
4. How to deal with errors in the service invocation in a more or less elegant manner:
   - server is down,
   - communication is down,
   - server busy,
   - duplicated requests ...

1. RPC as a Programming tool

- The notion of distributed service invocation became a reality at the beginning of the 80's when procedural languages (mainly C) were dominant.
- In procedural languages, the basic module is the procedure. A procedure implements a particular function or service that can be used anywhere within the program.
- It seemed natural to maintain this same notion when talking about distribution: the client makes a **procedure call** to a procedure that is implemented by the server.
- Since the client and server can be in different machines, the procedure call is **remote**.
- Client/Server architectures are based on Remote Procedure Calls (RPC)
- Once we are working with remote procedures in mind, there are several aspects that are immediately determined:
  - The input and output parameters of the procedure call are used for exchanging data
  - Pointers cannot be passed as parameters in RPC, opaque references are needed instead so that the client can use this reference to refer to the same data structure or entity at the server across different calls.

2. Interoperability

- When exchanging data between clients and servers residing in different environments (hardware or software), care must be taken that the data is in the appropriate format:
  - byte order: differences between little endian and big endian architectures (high order bytes first or last in basic data types)
  - data structures: like trees, hash tables, multidimensional arrays, or records need to be flattened (cast into a string so to speak) before being sent
- This is best done using an intermediate representation format
- The concept of transforming the data being sent to an intermediate representation format and back has different (equivalent) names:
  - marshalling/un-marshalling
  - serializing/de-serializing
- The non-standard intermediate representation format is typically system dependent. For instance:
  - SUN RPC: XDR (External Data Representation)
- Having an intermediate representation format simplifies the design, otherwise a node will need to be able to transform data to any possible format

Example (XDR in SUN RPC)

- Marshalling or serializing can be done by hand (although this is not desirable) using (in C) *sprintf* and *sscanf*:

```
Message= "Cesare" "ETHZ" "2006"

char *name="Cesare", place="ETHZ";
int year=2004;

sprintf(message, "%d %s %s %d %d",
  strlen(name), name, strlen(place), place,
  year);

Message after marshalling =
            "6 Cesare 4 ETHZ 2006"
```

- Remember that the type and number of parameters is known in advance, we only need to agree on the syntax ...

- SUN XDR follows a similar approach:
  - messages are transformed into a sequence of 4 byte objects, each byte being in ASCII code
  - it defines how to pack different data types into these objects, which end of an object is the most significant, and which byte of an object comes first
  - the idea is to simplify computation at the expense of bandwidth

6				String length
C	e	s	a	String content
r	e			
4				String length
E	T	H	Z	String content
2	o	o	6	Number

3. Binding

- A service is provided by a server located at a particular IP address and listening to a given port
- Binding is the process of mapping a service name to an address and port that can be used for communication purposes
- Binding can be done:
  - locally: the client must know the name (address) of the host of the server
  - distributed: there is a separate service (service location, name and directory services, etc.) in charge of mapping names and addresses. This service must be reachable by all participants

- With a distributed binder, several general operations are possible:
  - REGISTER (Exporting an interface): A server can register service names and the corresponding port
  - WITHDRAW: A server can withdraw a service
  - LOOKUP (Importing an interface): A client can ask the binder for the address and port of a given service
- There must also be a way to locate the binder (predefined location, environment variables, configuration file, broadcasting to all nodes looking for the binder)
- Clients usually cache binding information (rebinding is attempted on failures)

4. Call semantics

What happens when a LOCAL procedure is called?

- The procedure always runs once, exactly.

What happens when a REMOTE procedure is called?

- The procedure never runs because the server is down.
- The procedure does not run because the client is disconnected from the network.
- The procedure runs, but the client does not notice because the result is lost.
- The procedure runs, but the server crashes in the middle
- The procedure runs, *twice*, because the client has resent the request packet.
- If all goes well, the procedure runs once.

How many times should this procedure run?

Deposit(MyAccount, $99);

Reminder:
it looks like
a procedure call,
but the parameters are
sent back and forth
on the network!

Defining Call semantics

- A client makes an RPC to a service at a given server. After a time-out expires, the client may decide to re-send the request. If after several tries there is no success, what may have happened depends on the call semantics:

- Maybe: no guarantees. The procedure may have been executed (the response message(s) was lost) or may have not been executed (the request message(s) was lost). It is very difficult to write programs based on this type of best effort semantics since the programmer has to take care of all possibilities

- At least-once: the procedure will be executed if the server does not fail, but it is possible that it is executed more than once. This may happen, for instance, if the client *re-sends the request after a time-out*. If the server is designed so that service calls are idempotent (produce the same outcome given the same input), this might be acceptable.
- At most-once: the procedure will be executed either once or not at all. Re-sending the request will not result in the procedure executing several times. The server must perform some kind of **duplicate detection** and filtering and reply retransmission

RPC Error semantics

Semantics	Type of failure		
	Normal execution	Network failure	Server failure
Maybe	Request sent: 1 Execution: 1 Result sent: 1 Result received: 1	Request sent: 0/1 Execution: 0/1 Result sent: 0/1 Result received: 0/1	Request sent: 1 Execution: 0/1 Result sent: 0/1 Result received: 0/1
At-Least-Once	Request sent: 1 Execution: 1 Result sent: 1 Result received: 1	Request sent: 1/N Execution: 1/N Result sent: 1/N Result received: 1	Request sent: 1/N Execution: 0/N Result sent: 0/N Result received: 0/1
At-Most-Once	Request sent: 1 Execution: 1 Result sent: 1 Result received: 1	Request sent: 1/N Execution: 1 Result sent: 1/N Result received: 1	Request sent: 1/N Execution: 0/1 Result sent: 0/N Result received: 0/1

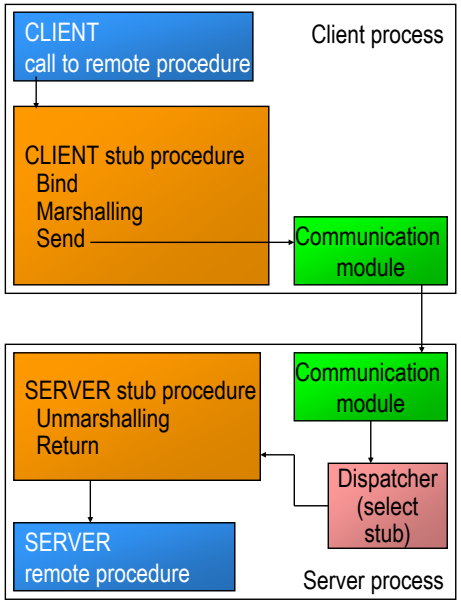
IKS Information and Communication Systems Research Group

ETH Eidgenössische Technische Hochschule Zürich Swiss Federal Institute of Technology Zurich

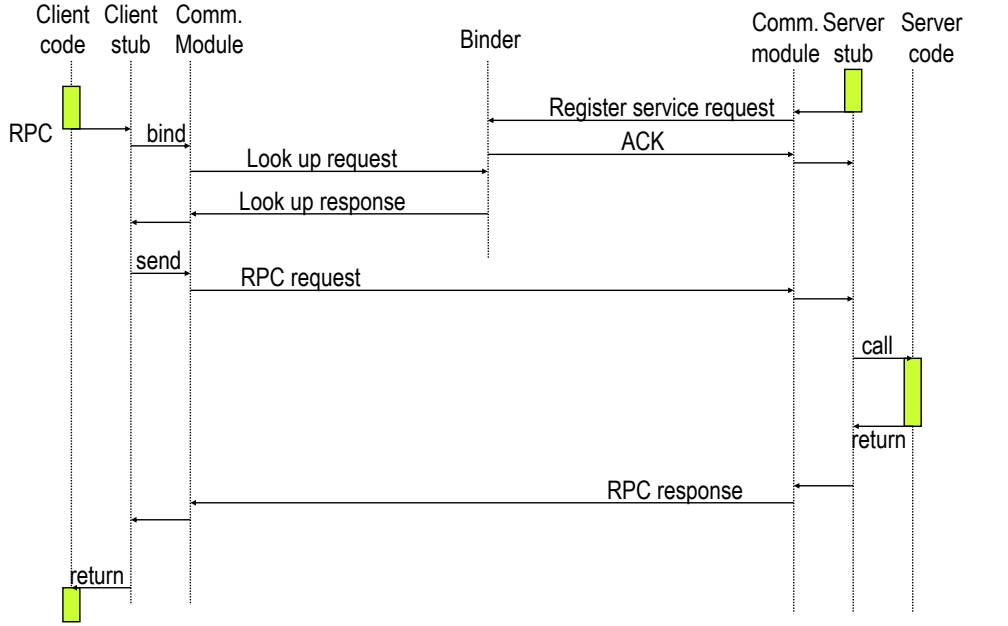
How RPC works

Making it work in practice

- One cannot expect the programmer to implement all these mechanisms every time a distributed application is developed. Instead, they are provided by a so called RPC system (a first example of low level middleware)
- What does an RPC system do?
  - Provides an interface definition language (IDL) to describe the services
  - Generates all the additional code necessary to make a procedure call remote and to deal with all the communication aspects
  - Provides a binder in case it has a distributed name and directory service system



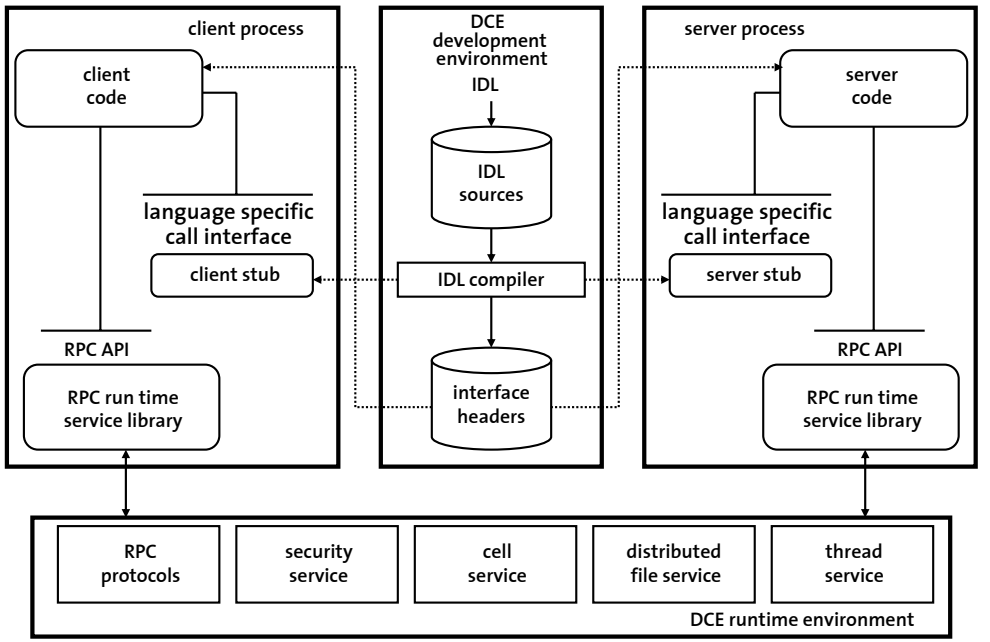
In more detail



IDL (Interface Definition Language)

- All RPC systems come with a language that allows to describe services in an abstract manner (independent of the programming language used). This language has the generic name of IDL (e.g., the IDL of SUN RPC is XDR)
- The IDL allows to define each service in terms of their names, and input and output parameters (plus maybe other relevant aspects).
- An interface compiler is then used to generate the stubs for clients and servers (*rpcgen* in SUN RPC). It might also generate procedure headings that the programmer can then use to fill out the details of the server-side implementation.
- Given an IDL specification, the interface compiler performs a variety of tasks to generate the stubs in a target programming language (like C):
1. Generates the **client stub** procedure for each procedure signature in the interface. The stub will be then compiled and linked with the client code
2. Generates a **server stub**. It can also create a server *main*, with the stub and the dispatcher compiled and linked into it. This code can then be extended by the developer by writing the implementation of the procedures
3. It might generate a *.h file for importing the interface and all the necessary constants and types

Putting it all together



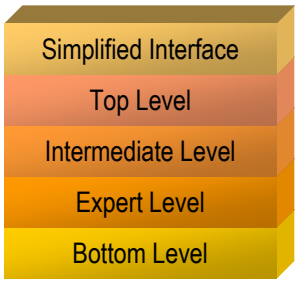
RPC in pseudocode

```
//your client code
result = function(parameters)

//client side stub
function(parameters) {
    address a = bind("function");
    socket s = connect(a);
    send(s,"function");
    send(s,parameters);
    receive(s,result); //blocking
    return result;
}
```

```
//rpc server main loop
void rpc_server() {
    register("function",address);
    while (true) {
        socket s = accept(); //blocking
        receive(s,id);
        if (id == "function")
            dispatch_function(s);
        close(s);
    }
}

//server side stub
void dispatch_function(socket s) {
    receive(s,parameters);
    result = function(parameters);
    send(s,result);
}
```

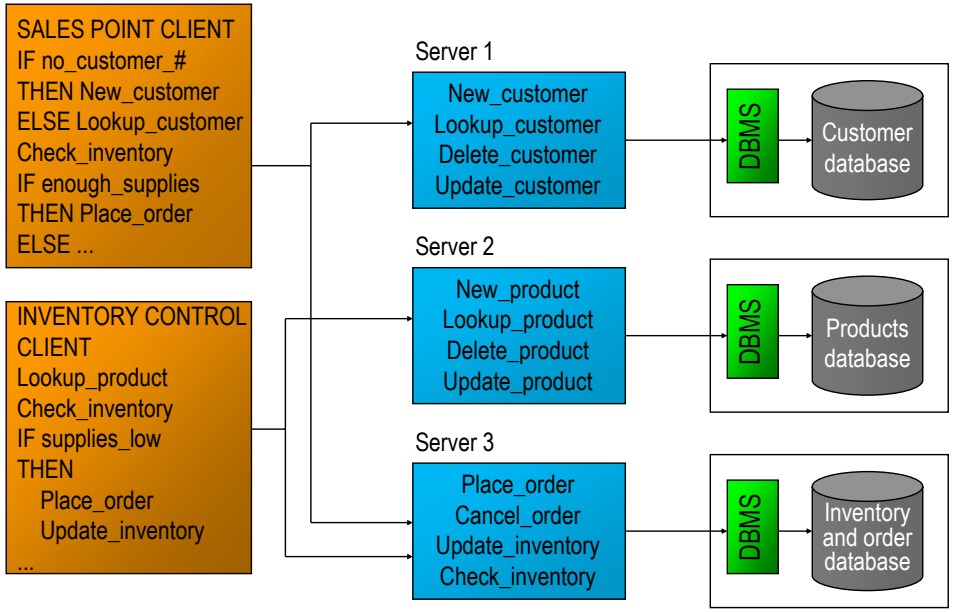

Programming RPC directly

- RPC usually provides different levels of interaction to provide different degrees of control over the system:



- Each level adds more complexity to the interface and requires the programmer to take care of more aspects of a distributed system
- The Simplified Interface (in SUN RPC) has only three calls:
  - rpc_reg() registers a procedure as a remote procedure and returns a unique, system-wide identifier for the procedure
  - rpc_call() given a procedure identifier and a host, it makes a call to that procedure
  - rpc_broadcast() is similar to rpc_call() but broadcasts the message instead
- The IDL compiler automatically generates the stubs calling the RPC library using defaults.
- Direct access allow more control of transport protocols, security, marshalling, binding, asynchronous procedures, etc.

RPC Application Example



RPC in perspective

ADVANTAGES

- RPC provided a mechanism to implement distributed applications in a simple and efficient manner
- RPC followed the programming techniques of the time (procedural languages) and fitted quite well with the most typical programming languages (C), thereby facilitating its adoption by system designers
- RPC allowed the modular and hierarchical design of large distributed systems:
  - client and server are separate
  - the server encapsulates and hides the details of the back end systems (such as databases)

DISADVANTAGES

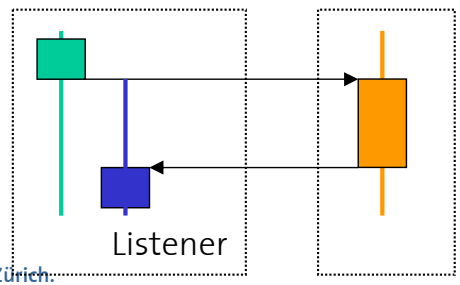
- RPC is not a standard, it is an idea that has been implemented in many different ways (not necessarily compatible)
- RPC allows designers to build distributed systems but does not solve many of the problems distribution creates. In that regard, it is only a low level construct
- RPC was designed with only one type of interaction in mind: client/server. This reflected the hardware architectures at the time when distribution meant small terminals connected to a mainframe. As hardware and networks evolve, more flexibility was needed

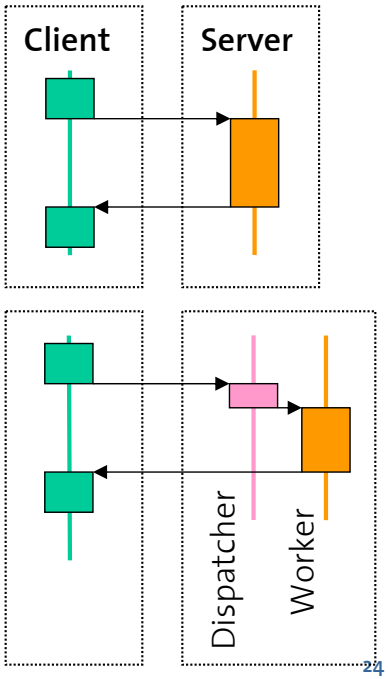
RPC system issues

- RPC was one of the first tools that allowed the modular design of distributed applications
- RPC implementations tend to be quite efficient in that they do not add too much overhead. However, a remote procedure is always slower than a local procedure:
  - should a remote procedure be transparent (identical to a local procedure)? (yes: easy of use; no: increase programmer awareness)
  - should location be transparent? (yes: flexibility and fault tolerance; no: easier design, less overhead)
  - should there be a centralized name server (binder)?
- RPC can be used to build systems with many layers of abstraction.
- However, every RPC call implies:
  - Several messages through the network
  - At least one context switch (at the client when it places the call, but there might be more)
  - Threads are typically used in the server to handle concurrent requests
- When a distributed application is complex, deep RPC chains are to be avoided

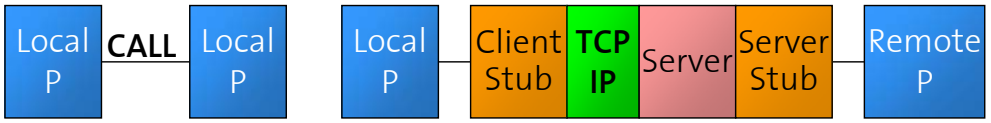
RPC and Concurrency

- A local procedure call happens within the same thread of control.
- A remote procedure call involves at least two different threads (one on the client and one on the server host)
- The server may use two threads: the **dispatcher** listens for requests and passes them to a **worker** thread for processing
- The client may not block and use a **listener** thread to wait for the results





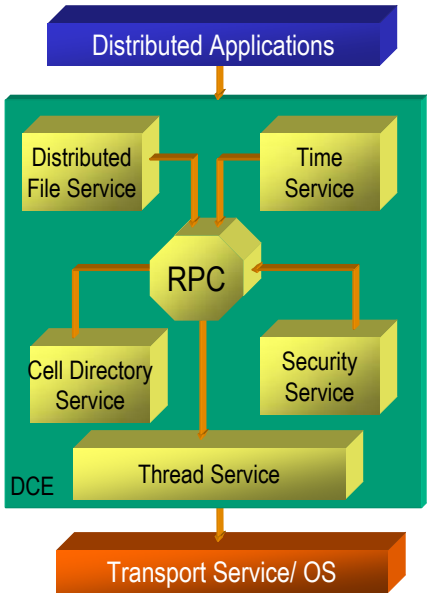
RPC Pitfalls



- Although RPC strives to keep the remote call transparent, there are a number of limitations
- **No Shared Memory** between client and server
- Arguments and Results are passed *by copy (not by reference)*.
- Difficult to exchange pointers and complex data structures
- Cannot pass a file opened on the client as a parameter to the server (and viceversa)
- Remote calls are orders of magnitude slower than local ones, do not call too often!
- Calls may fail due to network problems
- The server address must be configured on the client, unless dynamic binding is used
- Nobody can snoop the parameters of a local call (unless you use a debugger or you force a core dump...), but all parameters of every RPC are visible on the network
- The caller of a local procedure can be trusted because it is the same program. Can the server trust the client of a remote procedure in the same way?

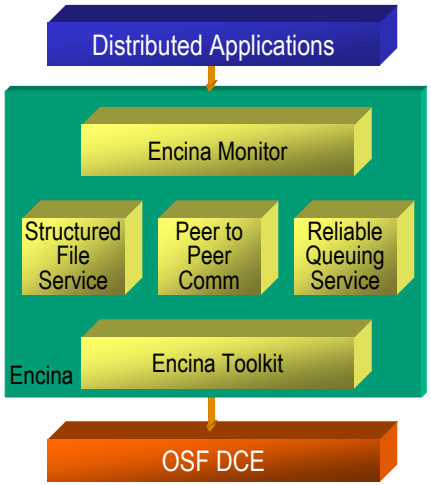
DCE

- The Distributed Computing Environment is a standard implementation of RPC and a distributed run-time environment provided by the Open Software Foundation (OSF). It provides:
  - RPC
  - Cell Directory: A sophisticated Name and Directory Service
  - Time: for clock synchronization across all nodes
  - Security: secure and authenticated communication
  - Distributed File: enables sharing of files across a DCE environment
  - Threads: support for threads and multiprocessor architectures



DCE's model and goals

- Not intended as a final product but as a basic platform to build more sophisticated middleware tools
- Its services are provided as the most basic services needed in any distributed system. Any other functionality needs to be implemented on top of it
- DCE is not just an specification of a standard (e.g., CORBA) but an implementation that acts as the standard. Since the API is the same across all platforms, interoperability is always guaranteed
- DCE is packaged in a modular way so that services that are not used do not need to be licensed
- Microsoft DCOM is built on top of DCE RPC.



From RPC we go to ...

Stored procedures

- Two tier architectures are, in fact, client/server systems. They need some sort of interface to allow clients to invoke the functionality of the server. RPC is the ideal interface for client/server interactions on a LAN
- To add flexibility to their servers, software vendors added to them the possibility of programming procedures that will run inside the server and that could be invoked through RPC
- This turned out to be very useful for databases where such procedures could be used to hide the schema and the SQL programming from the clients. The result was stored procedures, a common mechanism found in all database systems

Distributed environments

- When designing distributed applications, there are a lot of crucial aspects common to all of them. RPC does not address any of these issues
- To support the design and deployment of distributed systems, programming and run time environments started to be created. These environments provide, on top of RPC, much of the functionality needed to build and run a distributed application
- The notion of distributed environment is what gave rise to middleware.
- Web Services (with SOAP) are an example of extending the notion of RPC to call services located across the Web.