Chapter 2:
Remote Procedure Call (RPC)

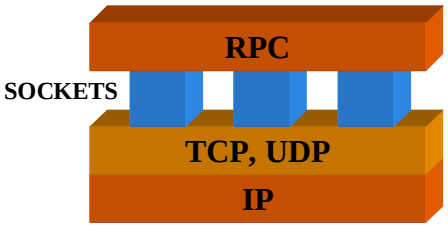
Gustavo Alonso
Computer Science Department
Swiss Federal Institute of Technology (ETHZ)
alonso@inf.ethz.ch
http://www.iks.inf.ethz.ch/

Contents - Chapter 2 - 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)

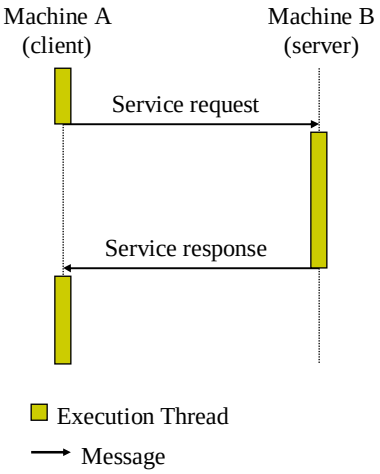
IP, TCP, UDP and RPC

- 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 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



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

- 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
- 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)
- 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.
- 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 ...

Programming languages

- 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 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:
  - data exchange is done as input and output parameters of the procedure call
  - 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. *The server is responsible for providing this opaque references.*

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 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

Binding

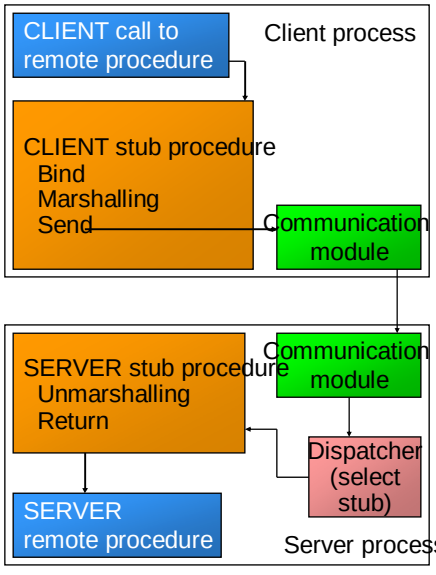
- A service is provided by a server located at a particular IP address and listening to a given port
- Binding is the process of mapping a service name to an address and port that can be used for communication purposes
- Binding can be done:
  - locally: the client must know the name (address) of the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

Call semantics

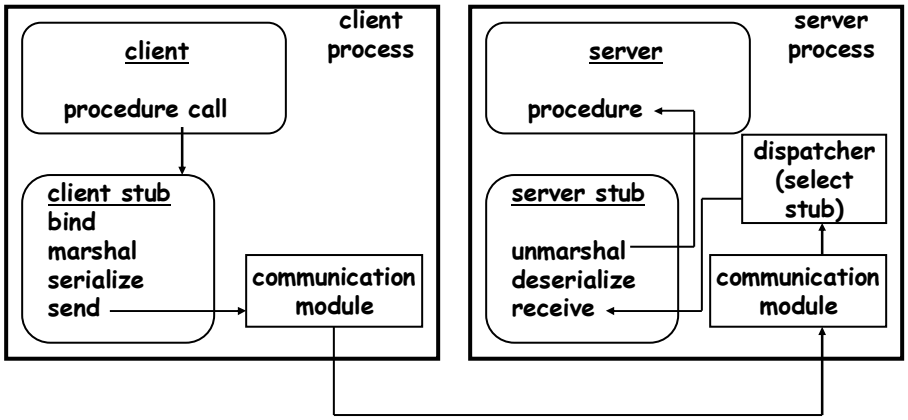
- A client makes an RPC to a service at a given server. After a time-out expires, the client-side RPC library may decide to re-send the request. If after several tries there is no success, what may have happened depends on the call semantics:
- Maybe: no guarantees. The procedure may have been executed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

Making it work in practice

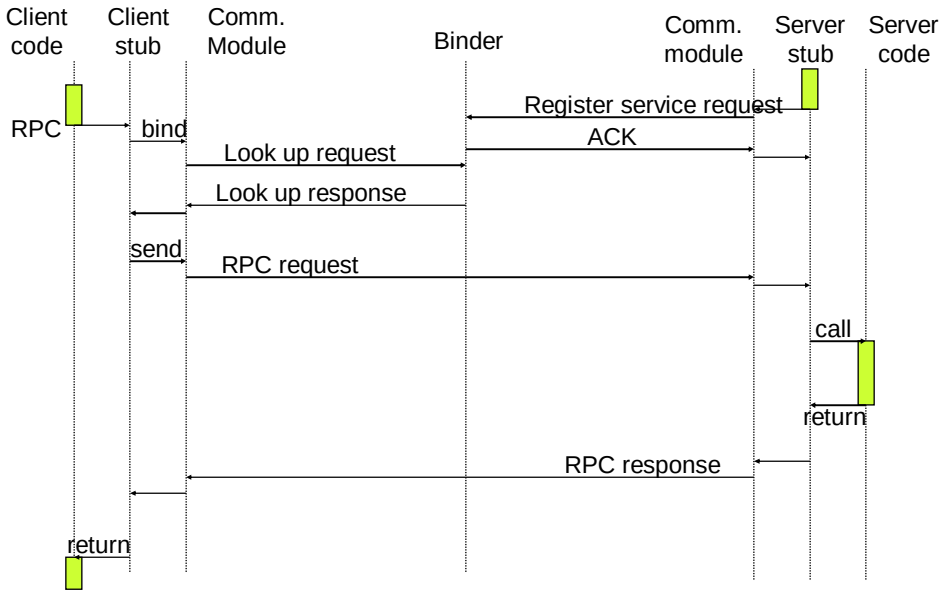
- One cannot expect the programmer to implement all these mechanisms every time a distributed application is developed. Instead, they are provided by a so called RPC system (our first example of low level middleware)
- What does an RPC system do?
  - Provides an interface definition language (IDL) to 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



Making it work in practice



In more detail

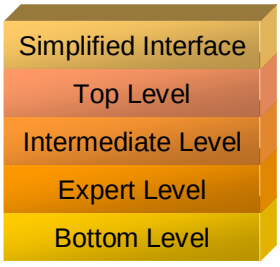


IDL (Interface Definition Language)

- All RPC systems have 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
- 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 used to fill out the details of the implementation.
- Given an IDL specification, the interface compiler performs a variety of tasks:
- generates the client stub procedure for each procedure signature in the interface. The stub will be then compiled and linked with the client code
- 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 designer by writing the implementation of the procedures
- It might generate a *.h file for importing the interface and all the necessary constants and types

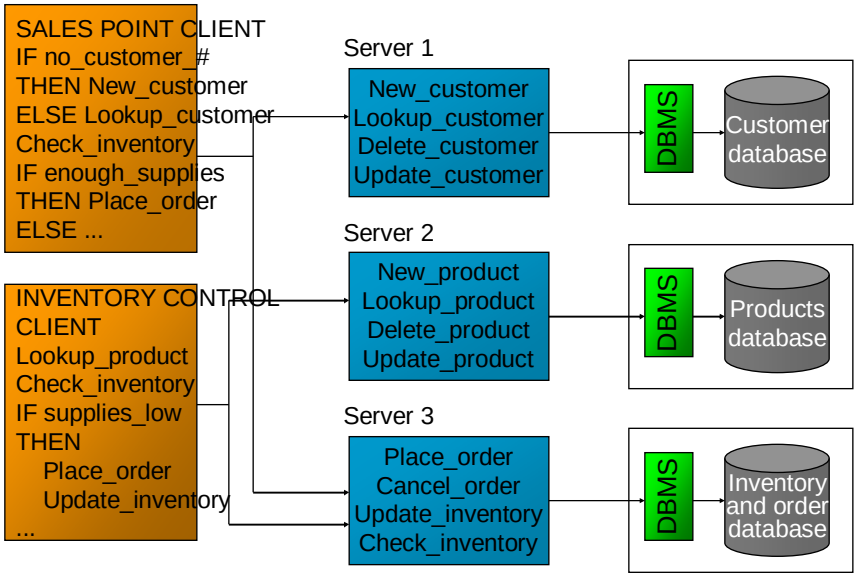
Programming RPC

- 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
- Additional levels allow more control of transport protocols, binding procedures, etc.

RPC Application



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 entities
  - 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)
- When a distributed application is complex, deep RPC chains are to be avoided

From RPC we go to ...

Stored procedures

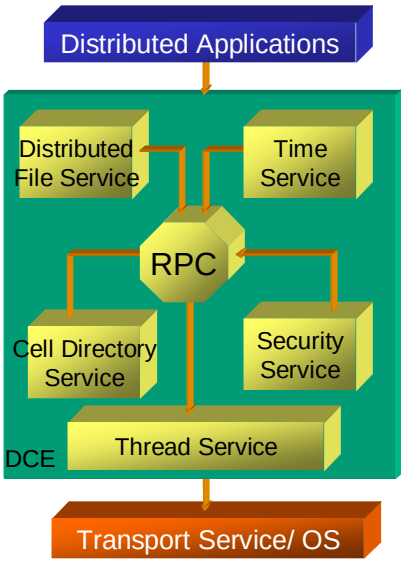
- Two tier architectures are, in fact, client/server systems. They need some sort of interface to allow clients to invoke the functionality of the server. RPC is the ideal interface for client/server interactions on a LAN
- To add flexibility to their servers, software vendors added to them the possibility 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

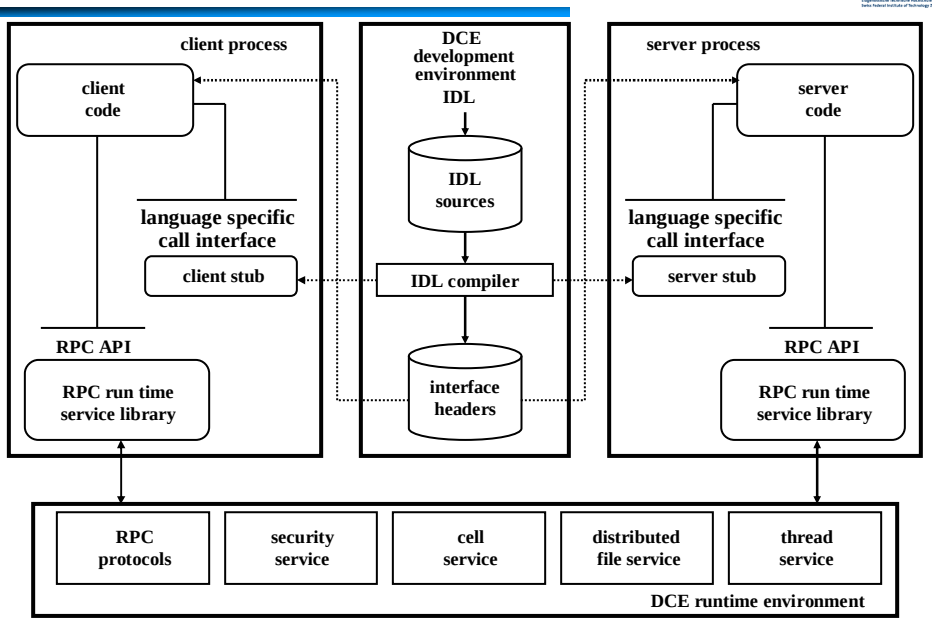
- When designing distributed applications, there are a lot of crucial aspects common to all of them. RPC does not address any of these issues
- To support the design and deployment of distributed systems, programming and run time environments started to be created. These environments provide, on top 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. During the course, we will see many examples of such environments.

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 architecture



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

- Encina (a TP-Monitor) is an example of an extension of DCE: