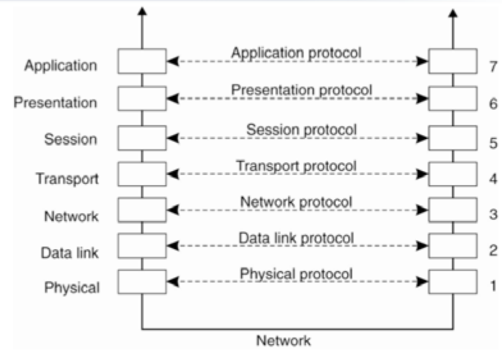
CIS 505: Software Systems

Lecture Note on Remote Procedure Call

Insup Lee
Department of Computer and Information Science
University of Pennsylvania

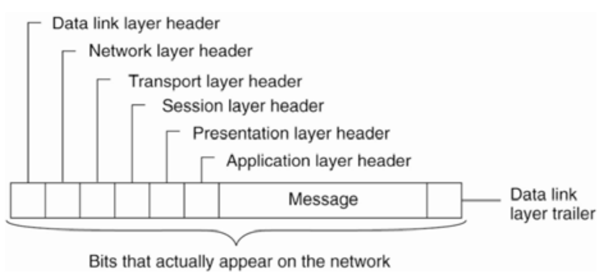
Spring 2007

Layered Protocols (1)



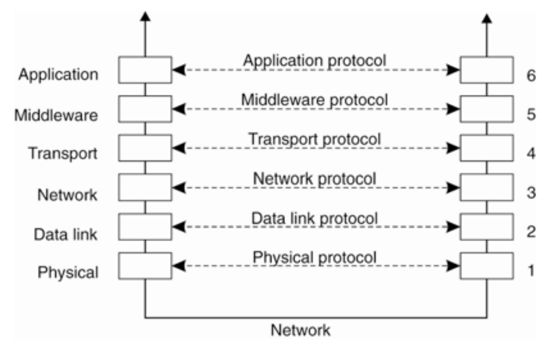
- Figure 4-1. Layers, interfaces, and protocols in the OSI model.

Layered Protocols (2)



- Figure 4-2. A typical message as it appears on the network.

Middleware Protocols



- Figure 4-3. An adapted reference model for networked communication.

A brief history

- DP (Distributed Processes) - Language support for distributed programming, Per Brinch Hansen in 1978
- Birrell and Nelson in 1980, based on work done at Xerox PARC.
- Similar idea used in RMI, CORBA or COM standards
- Core of many client-server systems
- Transparency is the goal!

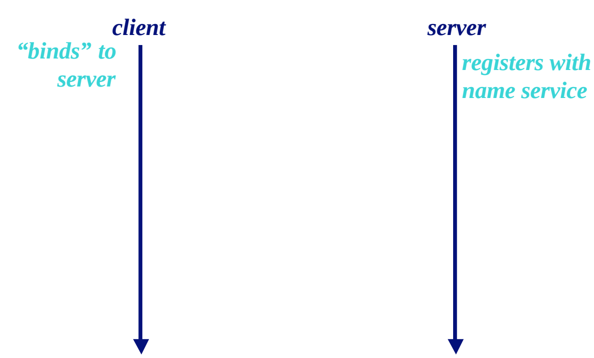
Remote procedure call

- A remote procedure call makes a call to a remote service look like a local call
  - RPC makes transparent whether server is local or remote
  - RPC allows applications to become distributed transparently
  - RPC makes architecture of remote machine transparent

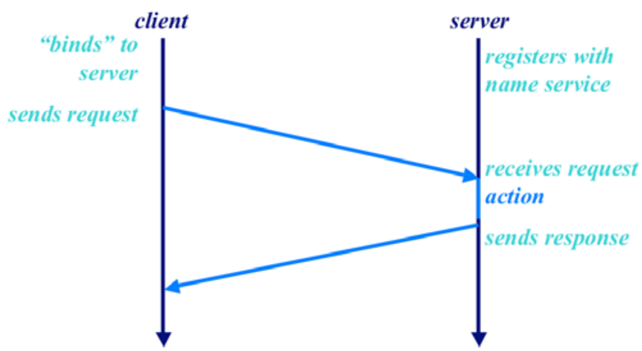
Clients and Servers

- A common model for structuring distributed computation
- Server
  - A program (or collection of programs) that provide some *service*, e.g., file service, name service, …
  - The server may exist on one or more nodes.
- Client
  - A program that uses the service.

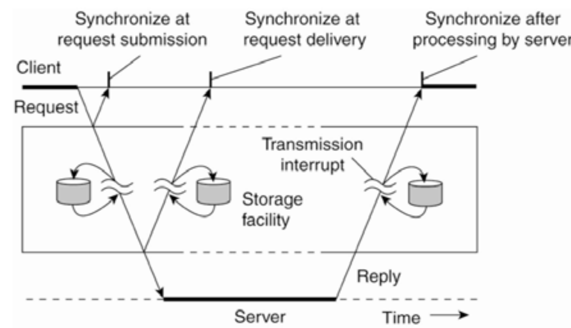
Client-server interaction



Client-server interaction



Types of Communication



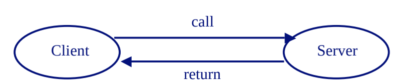
- Figure 4-4. Viewing middleware as an intermediate (distributed) service in application-level communication.

The Problem with Messages

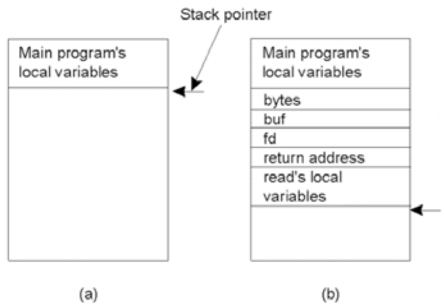
- Messages are flexible, but
- They are not a natural programming model
  - Programmers have to worry about message formats
  - messages must be packed and unpacked
  - messages have to be decoded by server to figure out what is requested
  - messages are often asynchronous
  - they may require special error handling functions

Procedure Call

- A more natural way to communicate
  - every programming language supports it
  - semantics are well defined and understood
  - natural for programmers to use
- Basic idea: define a server as a module that *exports* a set of procedures that can be called by client programs.



Conventional Procedure Call



a) Parameter passing in a local procedure call: the stack before the call to read (fd,buf,nbytes)
b) The stack while the called procedure is active

(Remote) Procedure Call

- Use procedure call as a model for distributed communication.
- Many issues:
  - how do we make this invisible to the programmer?
  - what are the semantics of parameter passing?
  - how is binding done (locating the server)?
  - how do we support heterogeneity (OS, arch., language)
  - etc.

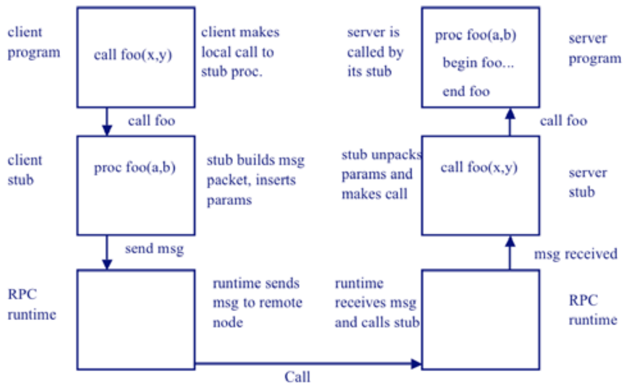
Remote Procedure Call

- Goal – make RPC look as much like local PC as possible.
- Uses computer/language support.
- There are 3 components on each side:
  - a user program (client or server)
  - a set of *stub* procedures
  - RPC runtime support

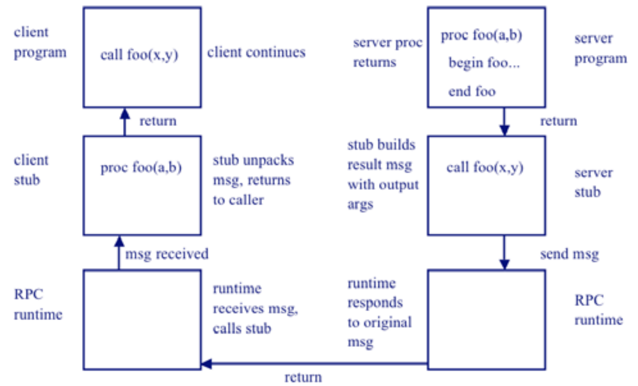
RPC Stubs

- A client-side stub is a procedure that looks to the client as if it were a callable server procedure.
- A server-side stub looks to the server as if it's a calling client.
- The stubs send messages to each other to make RPC happen.

RPC Call Structure (1)



RPC Return Structure (2)



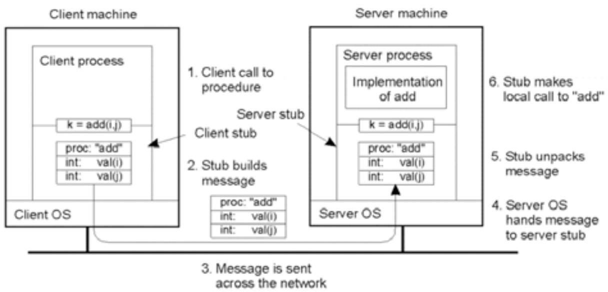
Stub Generation

- Server program defines the server's interface using an *interface definition language* (IDL)
  - o Define names, parameters, and types
- A *stub compiler* reads the IDL and produces two stub procedures for each server procedure
  - o The server program links it with the server-side stubs.
  - o The client program links with the client-side stubs.

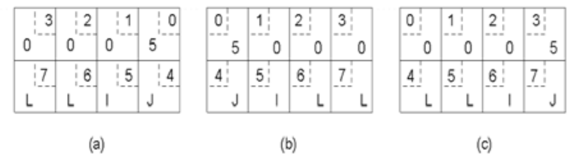
RPC Marshalling

- The packing of procedure parameters into a message packet.
- The RPC stubs call type-specific procedures to marshall (or unmarshall) all of the parameters to the call.
- Representation needs to deal with byte ordering issues (big-endian versus little endian), strings (some CPUs require padding), alignment, etc

Passing Value Parameters



Representation Conversion during Passing Value Parameters



a) Original message on the Pentium
b) The message after receipt on the SPARC
c) The message after being inverted. The little numbers in boxes indicate the address of each byte

RPC Parameter passing

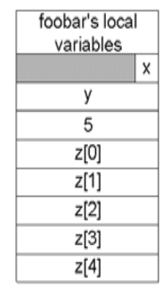
- By value
- By reference
- Size limit?

Parameter Specification and Stub Generation

a) A procedure
b) The corresponding message.

```
foobar( char x; float y; int z[5] )
{
  ....
}
```

(a)

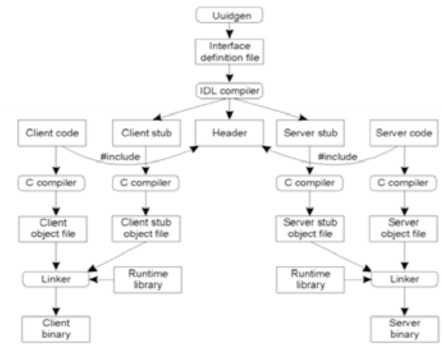


(b)

RPC Binding

- The process of connecting the client and server
- The server, when it starts up, *exports* its interface, identifying itself to a network name server .
- The client, before issuing any calls, *imports* the server, which causes the RPC runtime to lookup the server through the name service and contact the requested server to setup a connection.
- The *import* and *export* are explicit calls in the code.

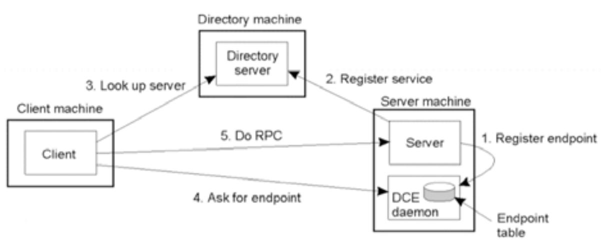
Writing a Client and a Server



- The steps in writing a client and a server in DCE RPC.

Binding a Client to a Server

- Client-to-server binding in DCE.



Transparencies in RPC

- A remote procedure call makes a call to a remote service look like a local call
  - RPC makes transparent whether server is local or remote
  - RPC allows applications to become distributed transparently
  - RPC makes architecture of remote machine transparent

Stubs: obtaining transparency

- Compiler generates from API stubs for a procedure on the client and server
- Client stub
  - Marshals arguments into machine-independent format
  - Sends request to server
  - Waits for response
  - Unmarshals result and returns to caller
- Server stub
  - Unmarshals arguments and builds stack frame
  - Calls procedure
  - Server stub marshalls results and sends reply

RPC vs. PC

- 3 properties of distributed computing that make achieving transparency difficult:
  - Memory access
    - Pointers are local to address space
    - How to implement pass by reference?
  - Latency
    - Orders of magnitude larger than PCs
  - Partial failures

Partial failures

- In local computing:
  - if machine fails, application fails
- In distributed computing:
  - if a machine fails, part of application fails
  - one cannot tell the difference between a machine failure and network failure
- How to make partial failures transparent to client?

Strawman solution

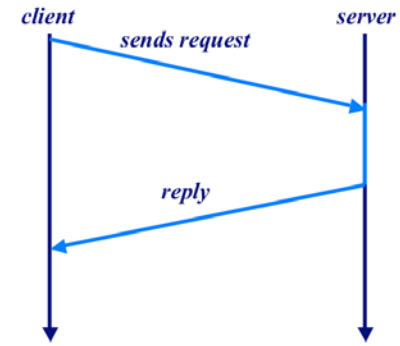
- Make remote behavior identical to local behavior:
  - Every partial failure results in complete failure
    - You abort and reboot the whole system
    - Many catastrophic failures
  - You wait patiently until system is repaired
    - Clients block for long periods
    - System might not be able to recover

Real solution: break transparency

- Expose RPC properties to client, since you cannot hide them
- Application writers have to decide how to deal with partial failures
- Which invocation semantics to provide?

Request-reply

Partial Failures: what and when can fail?



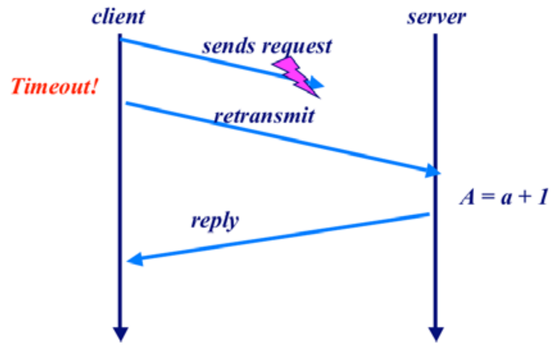
Invocation semantics

- Maybe
  - Don't know, or once
- At least once:
  - Don't know, or at least once
- At most once
  - Don't know, or once
- Exactly-once
  - Impossible in practice
- Zero or once
  - Transactional semantics
- At-most-once most practical
  - But different from PC

Maybe

- Add timeout
  - One request is sent
  - Timeout
- What's wrong with this?

Overcoming lost packets



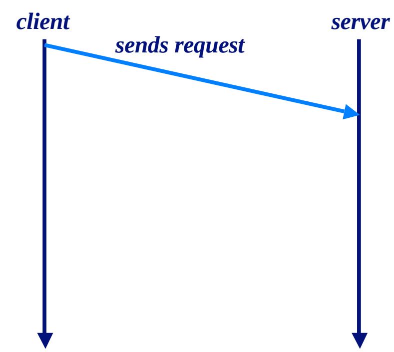
At least once

- Successful return
  - Executed at lease once.
- Only for idempotent functions, or need to suppress/compensate duplicated executions

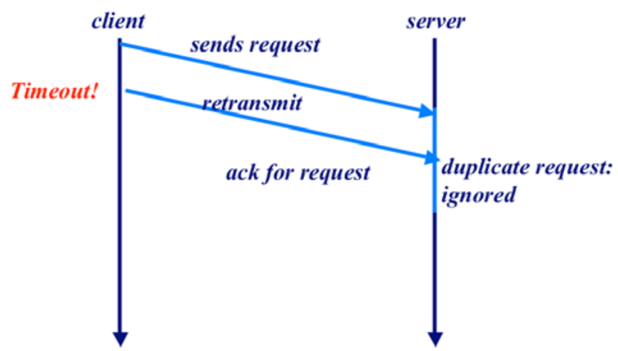
At most once

- Suppress duplicated requests
- Client
  - Each request has an unique id
- Server
  - Saves request results

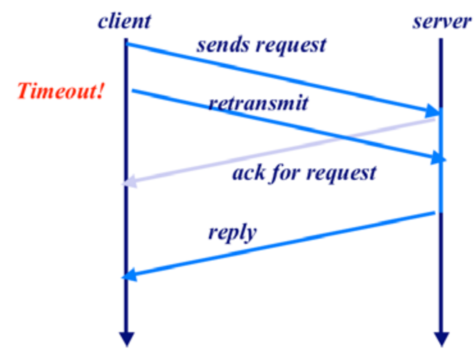
Cost of fault-tolerant version



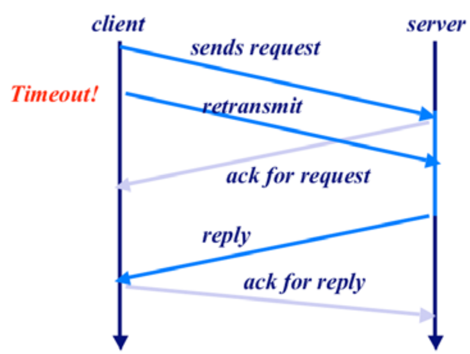
Overcoming lost packets



Overcoming lost packets



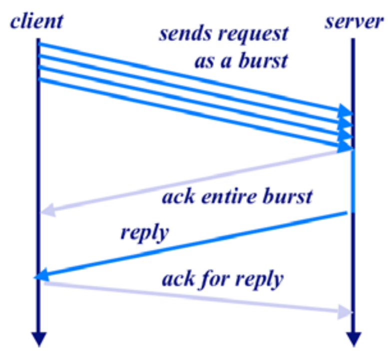
Overcoming lost packets



Costs in fault-tolerant version?

- Acks are expensive.
  - Try and avoid them, e.g., if the reply will be sent quickly suppress the initial ack
- Retransmission is costly.
  - Try and tune the delay to be “optimal”
- For big messages, send packets in bursts and ack a burst at a time, not one by one

Big packets



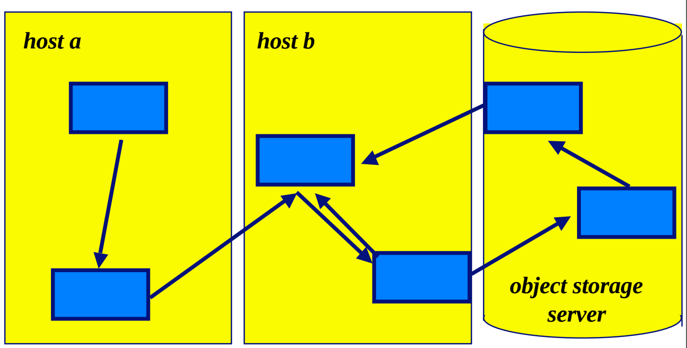
RPC versus local procedure call

- Restrictions on argument sizes and types
- New error cases:
  - Bind operation failed
  - Request timed out
  - Argument "too large" can occur if, e.g., a table grows
- Costs may be very high
- ... so RPC is actually not very transparent!

RMI for Distributed Objects

- An object could be a program or a data object
- A program object can *invoke* an operation on some other kind of object if it knows its type and has a *handle* on an instance of it.
- RMI (Remote Method Invocation)
- Each object has:
  - Type
  - Interface
  - "state"
  - Server location
  - Unique handle or identifier

Distributed objects



Pass by reference in RMI

- Objects can be declared remote
- Pointers to remote objects are passed in RMI
- Garbage Collection of Distributed Objects
- RMI Lease based protocol
  - Remote reference is valid for a limited time
  - Client has to renew lease.