Distributed Systems – Theory
4. Remote Procedure Call

Client-server model vs. RPC

- Client-server:
  - building everything around I/O
  - all communication built in send/receive
  - distributed computing look like centralized computing
- RPC allow to call procedures located on other machines

RPC principle

- When a process on machine A calls a procedure on machine B,
  - the calling process on A is suspended, and
  - execution of the called procedure takes place on B.
- Information can be transported from the caller to the callee in the parameters and can come back in the procedure result.
- No message passing or I/O at all is visible to the programmer.

Problems

- calling and called procedures run on different machines, they execute in different address spaces, -> complications.
- Parameters and results also have to be passed, which can be complicated, especially if the machines are not identical.
- both machines can crash, and each of the possible failures causes different problems

Basic RPC operation - steps

1. The client procedure calls the client stub in the normal way.
2. The client stub builds a message and traps to the kernel.
3. The kernel sends the message to the remote kernel.
4. The remote kernel gives the message to the server stub.
5. The server stub unpacks the parameters and calls the server.
6. The server does the work and returns the result to the stub.
7. The server stub packs it in a message and traps to the kernel.
8. The remote kernel sends the message to the client's kernel.
9. The client's kernel gives the message to the client stub.
10. The stub unpacks the result and returns to the client.

Parameter Passing (1)

- client stub: take the parameters, pack them into a message, and send it to the server stub.
- packing parameters into a message is called *parameter marshaling.*
- Example: *sum(i,j):* two integer parameters and returns their arithmetic sum. The client stub:
  - takes its two parameters and puts them in a message as indicated.
  - also puts the name or number of the procedure to be called in the message because the server might support several different calls, and it has to be told which one is required
  - … (see the 10 steps)

Parameter Passing (2)

- Model works fine as long as the client and server *machines are identical* and all the parameters and results are scalar types, such as integers, characters, and Booleans.
- Multiple machine type -> potential problems!
  - Example: some machines (e.g. Intel), number their bytes from right to left (format named *little endian*), whereas others (e.g. Sun), number them the other way (format named *big endian**)
  - Solution: *a standard* has been agreed upon for representing each of the basic data types

*the format names are given after the politicians in Gulliver's Travels who went to war over which end of an egg to break.

Using standards in parameter passing

- a network standard or canonical form for integers, characters, Booleans, floating-point numbers,
- require all senders to convert their internal representation to this form while marshaling
- Problem: sometimes inefficient
  - Example: big endian to big endian marshalling into a lilttle endian.
- Second approach: the client uses its own native format and indicates in the first byte of the message which format this is.
  - server stub converts if needed

Where the stub procedures come from?

- in many RPC-based systems, they are generated automatically
- given a specification of the server procedure and the encoding rules, the message format is uniquely determined
- a compiler read the server specification and generate a client stub that packs its parameters into the officially approved message format
- similarly, the compiler can also produce a server stub that unpacks them and calls the server
- both stub procedures are generated from a single formal specification of the server
  - makes life easier for the programmers,
  - reduces the chance of error
  - makes the system transparent with respect to differences in internal representation of data items.

Pointers and reference parameters? (1)

- Example: the client stub knows that the second parameter points to an array of characters and it knows how big the array is.
- First strategy:
  - copy the array into the message and send it to the server.
  - changes the server makes using the pointer (e.g. storing data into it) directly affect the message buffer inside the server stub.
  - when the server finishes, the original message can be sent back to the client stub, which then copies it back to the client.
  - call-by-reference has been replaced by copy/restore.

Pointers and reference parameters? (2)

- Optimization:
  - if the stubs know whether the buffer is an input parameter or an output parameter to the server, one of the copies can be eliminated.
  - if the array is input to the server (e.g. in a call to write) it need not be copied back.
  - if it is output, it need not be sent over in the first place.
  - the way to tell them is in the formal specification of the server procedure.
- Formal specification of the procedure
  - written in some kind of specification language,
  - telling what the parameters are,
  - which are input and which are output (or both),
  - and what their (maximum) sizes are.

General case of a pointer to an arbitrary data structure

- Example. A pointer to a complex graph.
- Approach:
  - actually passing the pointer to the server stub and generating special code in the server procedure for using pointers.
  - A pointer is dereferenced (put into a registry and indirect through registry) by sending a message back to the client stub asking it to fetch and send the item being pointed to (reads) or store a value at the address pointed to (writes).
  - Method highly inefficient:
    - Imagine having the file server store the bytes in the buffer by sending back each one in a separate message.

Why Dynamic Binding?

- Question: how the client locates the server?
- One method is just to the network address of the server into the client.
  - approach extremely inflexible!
    - If the server moves or if the server is replicated or if the interface changes, numerous programs will have to be found and recompiled.
- To avoid all these problems, some DS use what is *dynamic binding* to match up clients and servers.

Server's formal specification

- Example: the server with the specification that tells the name of the server (e.g. file_server), the version number (e.g. 3.1), and a list of procedures provided by the server (read, write, create, and delete).
- For each procedure, the types of the parameters are given.
- Each parameter is specified as being an in parameter, an out parameter, or an in out parameter. The direction is relative to the server.
  - An in parameter, such as the file name, is sent from the client to the server.
  - An out parameter such as buf in read, is sent from the server to the client. Buf is the place where the file server puts the data that the client has requested.
  - An in-out parameter would be sent from the client to the server, modified there, and then sent back to the client (copy/restore)

Binder and register

- When the server begins executing, the call to initialize outside the main loop *exports* the server interface, i.e.:
  - the server sends a message to a program called a *binder*, to make its existence known.
- this process of sending the message is referred to as *registering* the server.
- to register, the server gives the binder
  - its name,
  - its version number,
  - a unique identifier, typically 32 bits long, and
  - a *handle* used to locate it.
    - The handle is system dependent, and might be an Ethernet address, an IP address, an X.500 address, a sparse process identifier, or something else.
  - other information, e.g. concerning authentication,
- A server can also deregister with the binder when it is no longer prepared to offer service.

The binder interface

Call	*Input*	*Output*
Register	Name,version,handle, unique id	
Deregister	Name,version,unique id	
Lookup	Name, version	Handle, unique id

How the client locates the server

1. Calls one of the remote procedures for the first time, say, read,
2. The client stub sees that it is not yet bound to a server, so it sends a message to the binder asking to import version 3.1 of the file_server interface.
3. The binder checks to see if one or more servers have already exported an interface with this name and version number.
4. If no currently running server is willing to support this interface, the read call fails.
5. If a suitable server exists, the binder gives its handle and unique identifier to the client stub.
6. The client stub uses the handle as the address to send the request message to.
7. The message contains the parameters and the unique identifier, which the server's kernel uses to direct the incoming message to the correct server in the event that several servers are running on that machine.

Dynamic binding: advantages vs. disadvantages

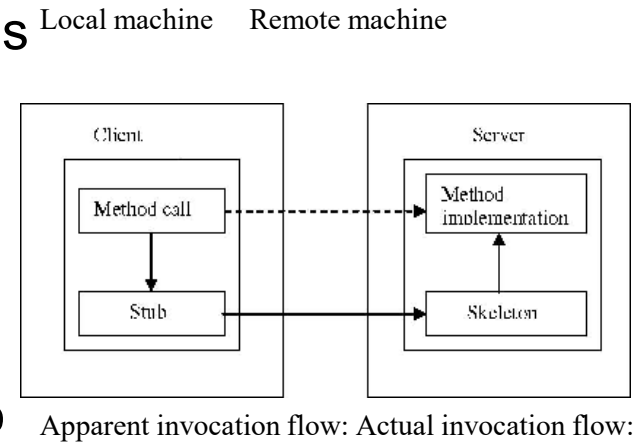
- Advantages:
  - Can handle multiple servers that support the same interface.
  - The binder can spread the clients randomly over the servers to even load.
  - The binder can poll the servers periodically, automatically deregistering any server that fails to respond, to achieve a degree of fault tolerance.
  - The binder can also assist in authentication.
    - A server could specify, for example, that it only wished to be used by a specific list of users, in which case the binder would refuse to tell users not on the list about it.
  - The binder can also verify that both client and server are using the same version of the interface.
- Disadvantages:
  - The extra overhead of exporting and importing interfaces costs time.
  - Since many client processes are short lived and each process has to start all over again, the effect may be significant.
  - In a large DS, the binder may become a bottleneck,
    - multiple binders are needed.
      - whenever an interface is registered or deregistered, a substantial number of messages will be needed to keep all the binders synchronized and up to date, creating even more overhead.

Connection-oriented protocol vs. connectionless protocol

- Connection-oriented protocol:
  - at the time the client is bound to the server, a connection is established between them.
  - All traffic, in both directions, uses this connection.
  - Advantage: communication becomes much easier.
    - When a kernel sends a message, it does not have to worry about it getting lost, nor does it have to deal with acknowledgements (handled by the software that supports the connection)
  - Disadvantage: the performance loss.
    - All that extra software gets in the way.
  - The main advantage (no lost packets) is hardly needed on a LAN, since LANs are so reliable.
- As a consequence, most DS that are intended for use in a single building or campus use connectionless protocols.

Remote Method Invocation (RMI)

- Invoke methods on remote objects (i.e., on objects located on other systems)
- Networking details required by explicit programming of streams and sockets disappear and the fact that an object is located remotely is almost transparent to the OO programmer
- The server program that has control of the remote object registers an interface with a naming service, thereby making this interface accessible by client programs.
- The interface contains the signatures for those methods of the object that the server wishes to make publicly available.
- A client program can then use the same naming service to obtain a reference to this interface in the form of a stub.
- The stub is effectively a local surrogate (a 'stand-in' or placeholder) for the remote object.
- On the remote system, there will be another surrogate, a skeleton.

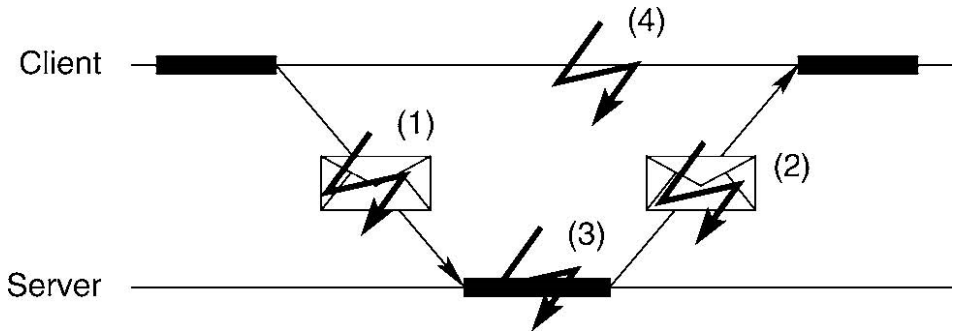


RMI details

- When the client program invokes a method of the remote object, it appears to the client as though the method is being invoked directly on the object.
- An equivalent method is being called in the stub.
- The stub then forwards the call and any parameters to the skeleton on the remote machine.
- In Java only primitive types and those reference types that implement the *Serializable* interface may be used as parameters - the serializing of these parameters is called marshalling.
- Upon receipt of the byte stream, the skeleton converts this stream into the original method call and associated parameters (the deserialization of parameters being referred to as unmarshalling)
- Finally, the skeleton calls the implementation of the method on the server.

Failures in communications

- loss of messages
- crash of a process



1. Loss of request message
2. Loss of result message
3. Server breakdown
4. Client breakdown

Failures types (1/2)

1. Loss of request message:
   - the client must retransmit the message after a timeout
   - problem: the client cannot differentiate between different types of failures
     - E.g. if the result message is the one that is lost, a retransmission of the request message could result in the procedure being executed twice
     - E.g. long procedures when too short a timeout is selected.
2. Loss of result message:
   - the client retransmits the request after a timeout.
   - problem: if the server does not recognize what happened, it executes the procedure again

Failure types (2/2)

3. Server breakdown:
   - If the server breaks down due to a failure, it has to be determined whether a partial execution of the procedure had already produced side effects in the state.
     - E.g. if the content of a database is modified during the procedure, it is not trivial to allow the execution to recover and continue in an ordered way after the crash of the server.
4. Client breakdown:
   - A client process that breaks down during the execution of a RPC is referred to as an *orphaned invocation*
   - Problem: what the server does with the results or where it should send them.

Failure semantics

- Different applications have different requirements for QoS (Quality of Service) in terms of failure detection and recovery

Failure semantics	*Fault-free operation*	*Message loss*	*Server breakdown*
Maybe	Execution: 1 Result: 1	Execution: 0/1 Result: 0	Execution: 0/1 Result: 0
At-least-once	Execution: 1 Result: 1	Execution: >=1 Result: >=1	Execution: >=0 Result: >=0
At-most-once	Execution: 1 Result: 1	Execution: 1 Result: 1	Execution: 0/1 Result: 0
Exactly once	Execution: 1 Result: 1	Execution: 1 Result: 1	Execution: 1 Result: 1

Maybe semantics

- Referred also as *best-effort*.
- Provide **no** mechanism for lost messages or process break downs
- E.g. RPC can be carried out zero times or once on the server side
- The client receives at most one result
- Provide **no** guarantees.
- So long as no failures occur, RPC is properly carried out

At-least-once semantics

- RPC will be executed on the server side at least once in the event of message loss
- After a timeout, the client repeats the RPC until it receives a response from the server.
- A procedure can be carried out several times on the server
- Possible that a client will receive several responses due to the repeated executions.
- Do not provide a confirmation if the server breaks down.
- Appropriate with idempotent procedures that do not cause state changes on the server and can be executed more than once without any harm.

At-most-once semantics

- the procedure will be executed at most once—both in the case of message loss and server breakdown
- If the server does not break down, exactly one execution and exactly one result are even guaranteed.
- Require a complex protocol with message buffering and numbering

Exactly once semantics

- Ideal case, not easy to achieve
- The invocation by a client will result in exactly one execution on the part of the server and also only delivers one result
- Particularly desirable for bank transactions
- The simple case: idempotent operations
  - The case of a simple information terminal that only read data from a remote server without changing the state of the server
    - repeated executions and numerous result messages would not be a problem