TDTS04 Distributed Systems
Part B

Slides derived from "Distributed Systems: Principles and Paradigms", by Andrew S. Tanenbaum and Maarten Van Steen, Pearson Int. Ed., as well as material from other instructors including Niklas Carlsson and Juha Takkinen.

MIKAEL ASPLUND
REAL-TIME SYSTEMS LABORATORY
DEPARTMENT OF COMPUTER AND INFORMATION SCIENCE

Communication

- How do distributed components talk to each other?

Communication in distributed systems

- "Distributed" processes
  - Located on different machines
- Need communication mechanisms
- Goal: Hide distributed nature as far as possible

Communication in distributed systems

- Networking primitives and protocols (e.g.: TCP/IP)
- Advanced communication models: Built on networking primitives
  - Messages
  - Streams
  - Remote Procedure Calls (RPC)
  - Remote Method Invocation (RMI)

Messages and streams

- We have already seen many protocols
  - Connection or connection less
  - Different layers
- Communication paradigm
  - Unicast
  - Multicast
  - Broadcast, limited flooding
  - Anycast
  - Publish-subscribe

Connection-oriented socket (TCP)

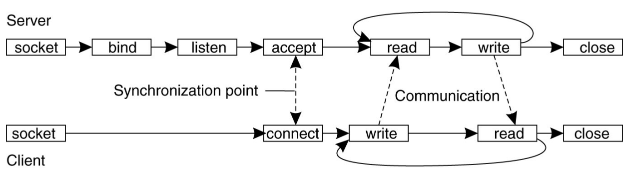


Figure 4-15. Connection-oriented communication pattern using sockets.

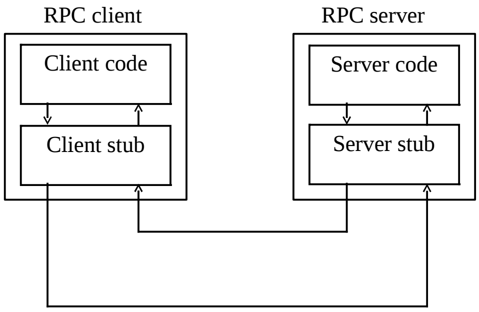
Remote procedure calls (RPC)

- **Goal:** Make distributed computation look like centralized computation
- **Idea:** Allow processes to call procedures on other machines
  - Make it appear like normal procedure calls

RPC operation

- Challenges:
  - Hide details of communication
  - Pass parameters transparently
- Stubs
  - Hide communication details
  - Client and server stubs
- Marshalling
  - Flattening and parameter passing

RPC operation



Stubs

- Code that communicates with the remote side
- Client stub:
  - Converts function call to remote communication
  - Passes parameters to server machine
  - Receives results
- Server stub:
  - Receives parameters and request from client
  - Calls the desired server function
  - Returns results to client

Client and server stubs

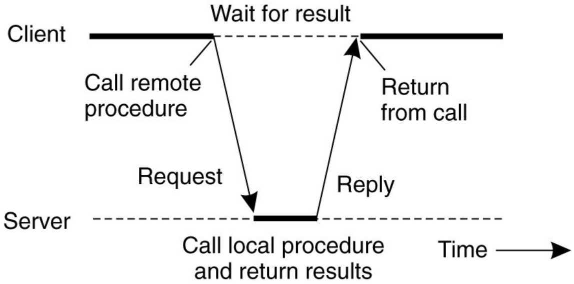


Figure 4-6. Principle of RPC between a client and server program.

Remote Procedure Calls (1)

A remote procedure call occurs in the following steps:

1. The client procedure calls the client stub in the normal way.
2. The client stub builds a message and calls the local operating system.
3. The client's OS sends the message to the remote OS.
4. The remote OS gives the message to the server stub.
5. The server stub unpacks the parameters and calls the server.

Continued …

Remote Procedure Calls (2)

A remote procedure call occurs in the following steps (continued):

6. The server does the work and returns the result to the stub.
7. The server stub packs it in a message and calls its local OS.
8. The server's OS sends the message to the client's OS.
9. The client's OS gives the message to the client stub.
10. The stub unpacks the result and returns to the client.

Parameter passing: Local procedures

- Pass-by-value
  - Original variable is not modified
  - E.g.: integers, chars
- Pass-by-reference
  - Passing a pointer
  - Value may be changed
  - E.g.: Arrays
- Pass-by-copy/restore
  - Copy is modified and overwritten to the original

Passing value parameters

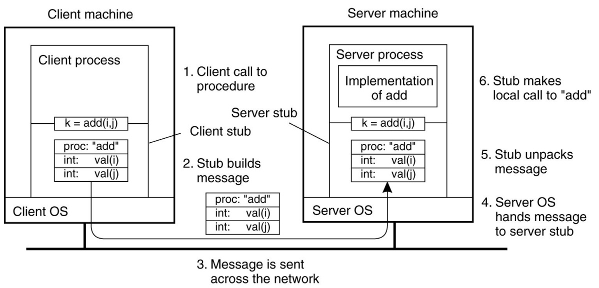


Figure 4-7. The steps involved in a doing a remote computation through RPC.

Marshalling

- Converting parameters into a byte stream
- Problems:
  - Heterogeneous data formats: Big-endian vs. little-endian
  - Type of parameter passing: By-value vs. by-reference

Stub generation

- Most stubs are similar in functionality
  - Handle communication and marshalling
  - Differences are in the main server-client code
- Application needs to know only stub interface
- Interface Definition Language (IDL)
  - Allows interface specification
  - IDL compiler generates the stubs automatically

Writing a Client and a Server (1)

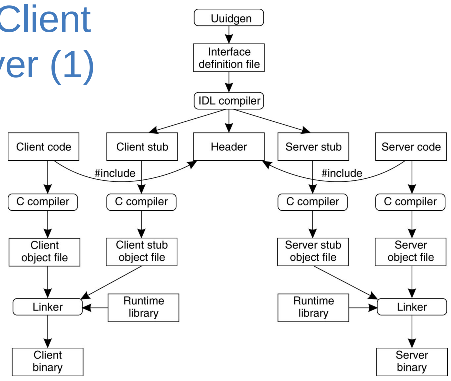


Figure 4-12. The steps in writing a client and a server in DCE RPC.

Writing a Client and a Server (2)

Three files output by the IDL compiler:

- A header file (e.g., interface.h, in C terms).
- The client stub.
- The server stub.

Binding

- How does the client stub find the server stub?
  - Needs to know remote IP address/port no.
- Port mapper
  - Daemon on server machine maintaining server bindings
  - Listens on a well-known port
- Server stub registers its port no. and service name with portmapper
  - Client gets this binding by querying portmapper

Binding a Client to a Server (1)

- Registration of a server makes it possible for a client to locate the server and bind to it.

- Server location is done in two steps:
  1. Locate the server's machine.
  2. Locate the server on that machine.

Binding a Client to a Server (2)

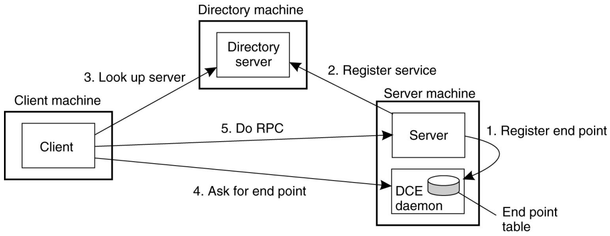


Figure 4-13. Client-to-server binding in DCE.

RPC issues

- Basic RPC performed in a synchronous manner
  - What if client wants to do something else?
- What if things fail?

Types of communication

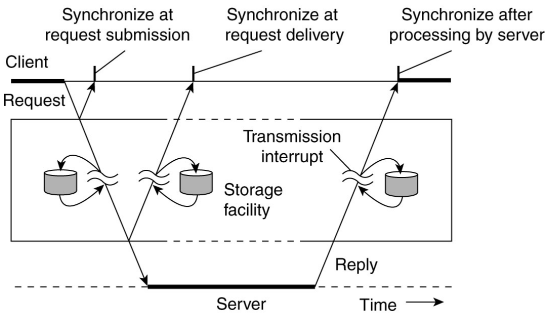


Figure 4-4. **Persistent vs. transient communication** and synchronous/asynchronous communication.

Types of communication

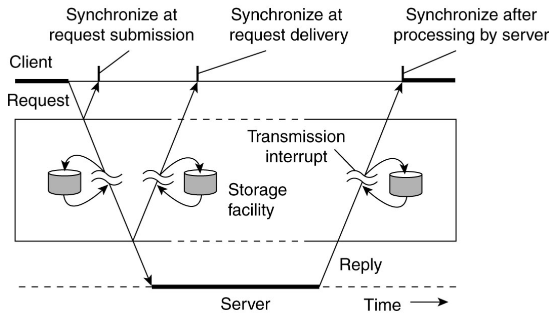


Figure 4-4. Persistent vs. transient communication and **synchronous/asynchronous communication**.

Asynchronous RPC

- Basic RPC
  - Client blocks until results come back
- Asynchronous RPC
  - Server sends ACK as soon as request is received
  - Executes procedure later
- Deferred synchronous RPC
  - Use two asynchronous RPCs
  - Server sends reply via second asynchronous RPC
- One-way RPC
  - Client does not even wait for an ACK from the server

Client and Server Stubs

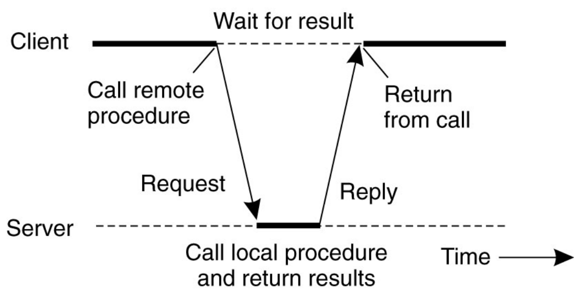
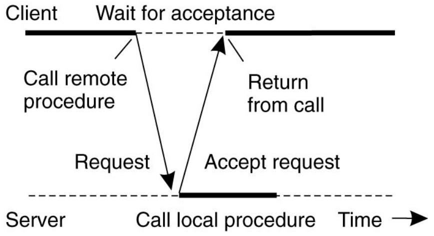


Figure 4-6. Principle of RPC between a client and server program.

Asynchronous RPC (2)



(b)

Figure 4-10. (b) The interaction using asynchronous RPC.

Asynchronous RPC (3)

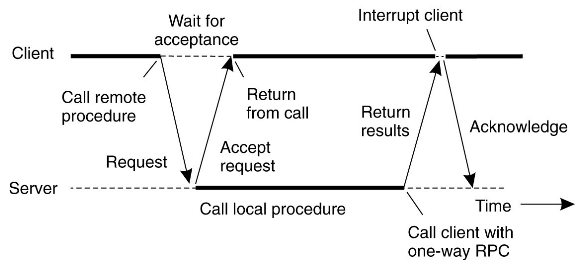


Figure 4-11. A client and server interacting through two asynchronous RPCs.

RPC: Network failure

- Client unable to locate server:
- Lost requests/replies:

RPC: Network failure

- Client unable to locate server:
  - Return error or raise exception
- Lost requests/replies:

RPC: Network failure

- Client unable to locate server:
  - Return error or raise exception
- Lost requests/replies:
  - Timeout mechanisms
  - Make operation idempotent (does not change the results beyond initial operation)
  - Use sequence numbers, mark retransmissions

RPC: Server failure

- Server may crash during RPC
  - Did failure occur before or after operation?
- Operation semantics

RPC: Server failure

- Server may crash during RPC
  - Did failure occur before or after operation?
- Operation semantics
  - Exactly once: desirable but impossible to achieve
  - At least once
  - At most once
  - No guarantee

RPC: Client failure

- Client crashes while server is computing
  - Server computation becomes orphan
- Possible actions

RPC: Client failure

- Client crashes while server is computing
  - Server computation becomes orphan
- Possible actions
  - Extermination: log at client stub and explicitly kill orphans
  - Reincarnation: Divide time into epochs between failures and delete computations from old epochs
  - Expiration: give each RPC a fixed quantum T; explicitly request extensions

Remote method invocation (RMI)

- RPCs applied to distributed objects
- Class: object-oriented abstraction
- Object: instance of class
  - Encapsulates data
  - Exports methods: operations on data
  - Separation between interface and implementation

Distributed objects

- Interface resides on one machine, object on another
- RMIs allow invoking methods of remote objects
- Use proxies, skeletons, binding
- Allow passing of object references as parameters

Distributed objects

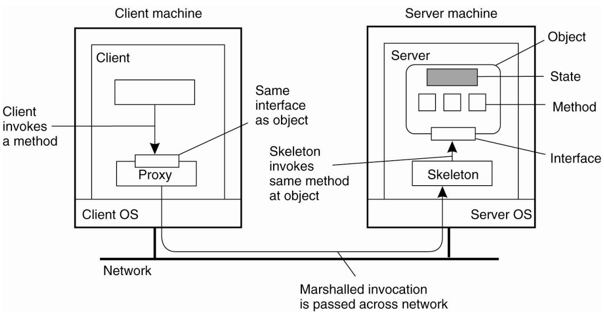


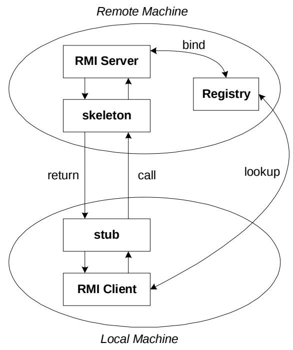
Figure 10-1. Common organization of a remote object with client-side proxy.

Proxies and skeletons

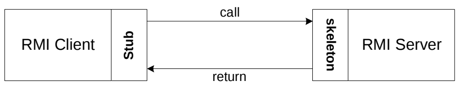
- Proxy: client stub
  - Maintains server ID, endpoint, object ID
  - Does parameter marshalling
  - In practice, can be downloaded/constructed on the fly
- Skeleton: server stub
  - Does demarshalling and passes parameters to server
  - Sends result to proxy

The General RMI Architecture

1. The server must first bind its name to the registry
2. The client looks up the server name in the registry to establish remote references.
3. …



The Stub and Skeleton



A client invokes a **remote method**, the call is first forwarded to the stub.

The stub is responsible for sending the remote call over to the server-side skeleton

The stub opens a socket to the remote server, **marshals** (Java: serializes) the object parameters and forwards the data stream to the skeleton.

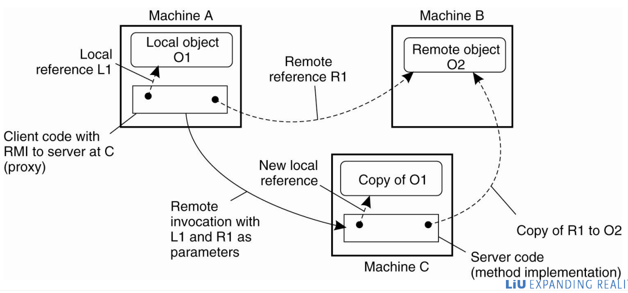
A skeleton contains a method that receives the remote calls, **unmarshals** the parameters, and invokes the actual remote object implementation.

Binding a client to an object

- Loading a proxy in client address space
- Implicit binding:
  - Bound automatically on object reference resolution
- Explicit binding:
  - Client has to first bind object
  - Call method after binding

Parameter passing

- Less restrictive than RPCs
  - Supports system-wide object references
  - Pass local objects by value, remote objects by reference



Steps for Developing an RMI System

1. Define the remote interface
2. Develop the remote object by implementing the remote interface.
3. Develop the client program.
4. Compile the Java source files.
5. Generate the client stubs and server skeletons.
6. Start the RMI registry.
7. Start the remote server objects.
8. Run the client

Object-based messaging

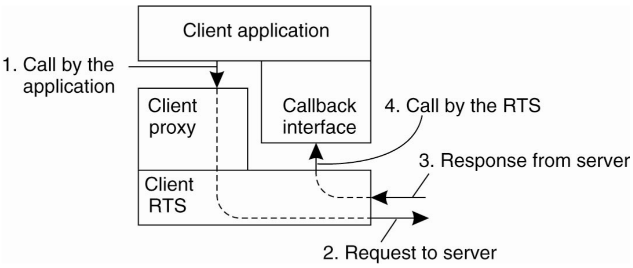


Figure 10-9. CORBA's callback model for asynchronous method invocation.

Object-based messaging

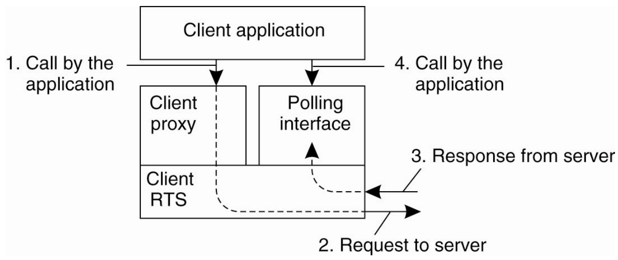


Figure 10-10. CORBA's polling model for asynchronous method invocation.

Naming: CORBA Object References

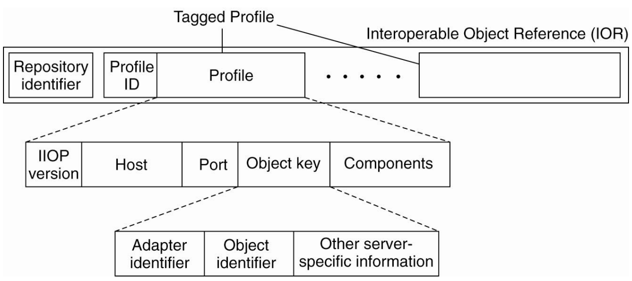


Figure 10-11. The organization of an IOR with specific information for IIOP.