5. Distributed objects and remote invocation

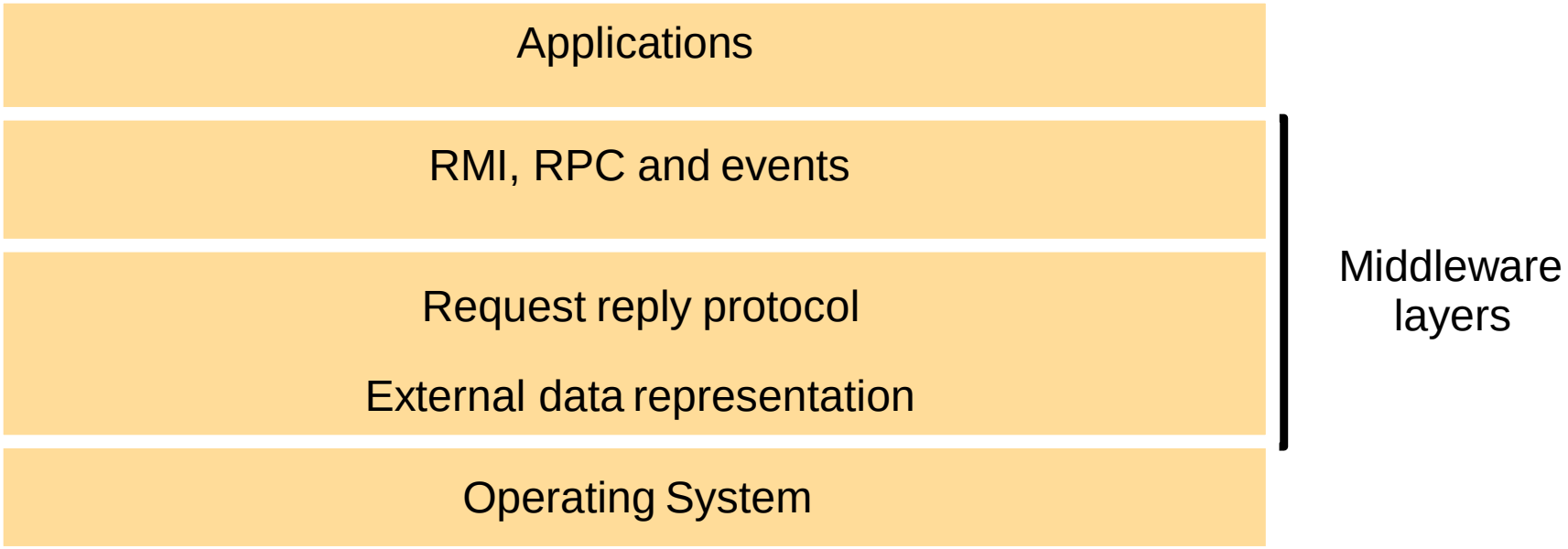
- Road Map
  - 5.1. Introduction
  - 5.2. Communication between distributed objects
  - 5.3. Remote procedure call (RPC)

5.1. Introduction

- Programming models for distributed programs/applications: applications composed of cooperating programs running in several different processes. Such programs need to invoke operations in other processes.
  - RPC – client programs call procedures in server programs, running in separate and remote computers (e.g., Unix RPC)
    - Extended from procedure call
  - RMI – extensions of object-oriented programming models to allow a local method (of a local object) to make a remote invocation of objects in a remote process (e.g., Java RMI)
    - Objected oriented version of RPC

5.1. Introduction

- Middleware
  - A suite of API software that uses underlying processes and communication (message passing) protocols to provide its higher level abstracts such as remote invocations and events
    - E.g., remote method invocation abstraction is based on the http protocol discussed in 4.4
  - The middleware provides location transparency, protocol abstraction, OS, and hardware independence, and multi-language support



5.1. Introduction

- An important aspect of middleware: provision of location transparency and independence from the details of communication protocols, OS and hardware
  - Location transparency: RPC, RMI, EBP
  - Protocol transparency: protocols supporting the middleware abstractions are independent of underlying transport protocols
    - request-reply protocol can be built on top of lower-level TCP or UDP
  - OS: all three paradigms could run on top of any OS platform
  - Hardware transparency: Issues with different data representations, conversions, and instruction set are transparent. Discussed in 4.3, marshalling & unmarshalling

5.1. Introduction

- Modern programming languages organize a program as a set of modules that communicate with one another
- Interface of a module specifies the procedures and the variables that can be accessed from other variables
- Interfaces hide the details of modules providing the services

5.1. Introduction

- Remote Object Interfaces
  - Modules can run in separate processes
  - Access to module variables is only indirectly
  - Parameter passing mechanisms, call by value/reference, used in local procedure call are not suitable when the caller is in a different process
  - Instead, describe the parameters as input or output
  - Input parameters are passed to remote module by sending values of the arguments in the request message
  - Output parameters are returned in the reply message to replace the values of the corresponding variables in the calling environment

5.1. Introduction

- RPC (client-server): service interface
  - Specifying the procedures offered by a server
  - Defining types of input/output arguments
- RMI (distributed object model): remote interface
  - Specifying methods of an object that are available for invocation by objects in other processes
  - Defining types of input/output arguments
  - Can pass objects as arguments; references to remote object may also be passed. Not to confuse them with pointers, which refer to specific memory locations

5.1. Introduction

- RMI mechanism can be integrated with a particular language: Java RMI
  - All parts of a distributed application need to be written in the same language
  - Convenient - allows programmer to use a single language for local and remote invocation
- However, many existing useful services are written in C++ or other languages…
- Interface definition languages: allow objects implemented in different languages to invoke one another
- provides a notation for defining interfaces: input, output, types
  - E.g., CORBA IDL (Fig 5.2) for RMI, Sun XDR for RPC

5.2. Communication between distributed objects

- By means of RMI:
  - The object model: OOP, Java or C++, review
  - Distributed objects: the object model is very appropriate for distributed systems
  - The distributed object model: extensions of the basic object model for distributed object implementation
  - The design issues of RMI: local once-or-nothing invocation semantics vs. remote invocation semantics – similarities or differences
  - The implementation issues: mapping the middleware to lower-layer facilities
  - Distributed garbage collection issues

5.2. Communication between distributed objects

- **The object model**
  - Objects (in classes) encapsulate methods and data variables, with some variables being directly accessible; and communication via passing arguments and receiving results from (locally) invoked objects
  - Object references: objects can be accessed via references. Accessing *target/receiver* objects requires – reference.methodname(args); and references can be passed as args, too.
  - Interfaces: provides a definition of the signatures of a set of object methods
    – arg type, return values, and exceptions. A class may implement several 'interfaces,' and an interface may be implemented by any class

5.2. Communication between distributed objects

- **The object model (cont..)**
  - Actions: is initiated by an object invoking a method in another object. An invocation can include additional information(arguments) needed to carry out the method execution. The receiver executes the appropriate method and then returns control to invoking object sometimes supplying the result. .
  - Exceptions: Provide a clean way to deal with error conditions without complicating the code. *throw* and *catch*
  - Garbage collection: reclaiming freed object spaces – Java (automatic), C++

5.2. Communication between distributed objects

- **Distributed objects**
  - State of an object: current values of its variables
  - State of program: partitioned into separate parts, each of which is associated with an object – locally partitioned
  - As a natural extension, objects are physically distributed into different processes or computers in a distributed system. Therefore, the object model is very appropriate for distributed systems
  - For C-S architecture, objects are managed by servers, clients invoke their methods using remote method invocation

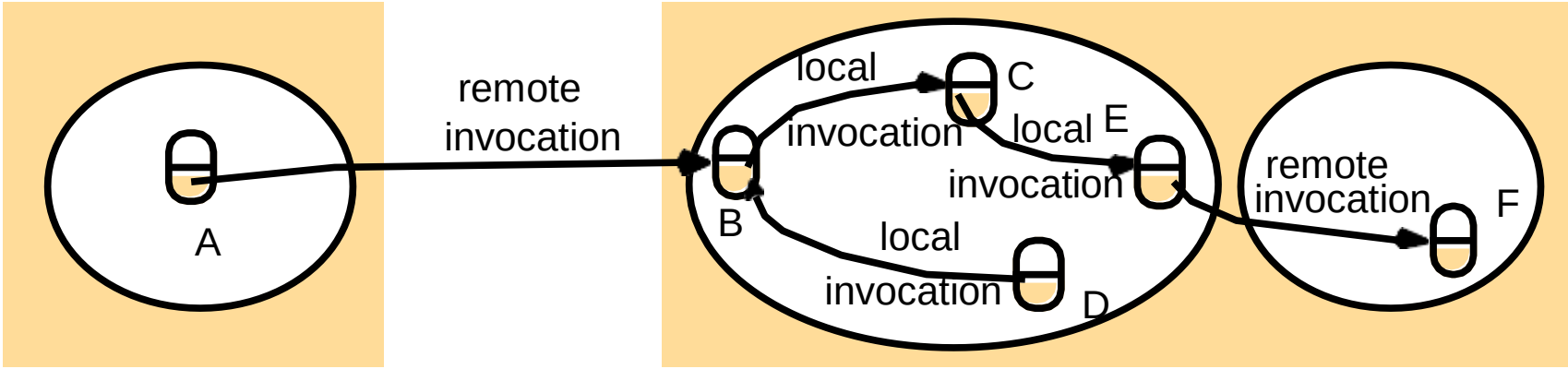
5.2. Communication between distributed objects

- **Distributed objects (cont..)**
  - In RMI, request is sent in a message to the server, the server execute it, and send result back to the client via a message
  - There are other architectures … (unimportant)
  - Distributed objects in different processes enforces encapsulation: the state of an object can be accessed only by the methods of the object
    - Only accept authorized methods to act on the state
    - Possibility to handle concurrent access to distributed objects
    - Allows heterogeneity: different data formats may be used at different sites

5.2. Communication between distributed objects

- **The distributed object model**
  - Discusses extensions to the basic object model to make it applicable to distributed objects
  - Show RMI is a natural extension of local method invocation
- RMI: invocations between objects in *different* processes (either on same or different computers)
  - Invocations within the *same* process are local
- Each process contains objects, some of which can receive remote invocations, others only local invocations
- Those that can receive remote invocations are called *remote objects*
- Objects need to know the *remote object reference* of an object in another process in order to invoke its methods. How do they get it?
- the *remote interface* specifies which methods can be invoked remotely

5.2. Communication between distributed objects(cont…)



- Objects receiving remote invocations (service objects) are remote objects, e.g., B and F
- Object references are required for invocation, e.g., C must have E's reference for local invc or B must have A's reference for remote invc
- B and F must have remote interfaces (of their accessible methods)

5.2. Communication between distributed objects(cont..)

- Remote object references
  - An unique identifier of a remote object, used throughout a distributed system
  - The remote object reference (including the 'interface' list of methods) can be passed as arguments or results in RMI
- Remote interfaces
  - The class of remote objects implements the methods of its remote interface
    - In Java, for example, as public instance methods
  - Local objects can access methods in an interface plus methods implemented by remote objects

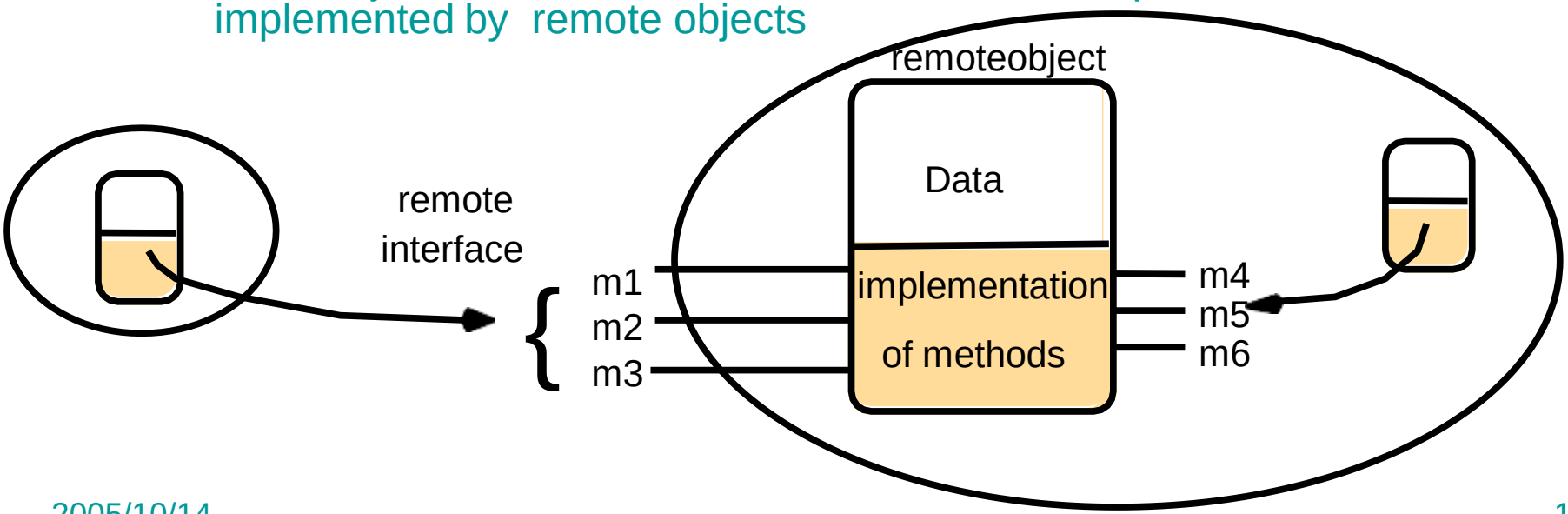
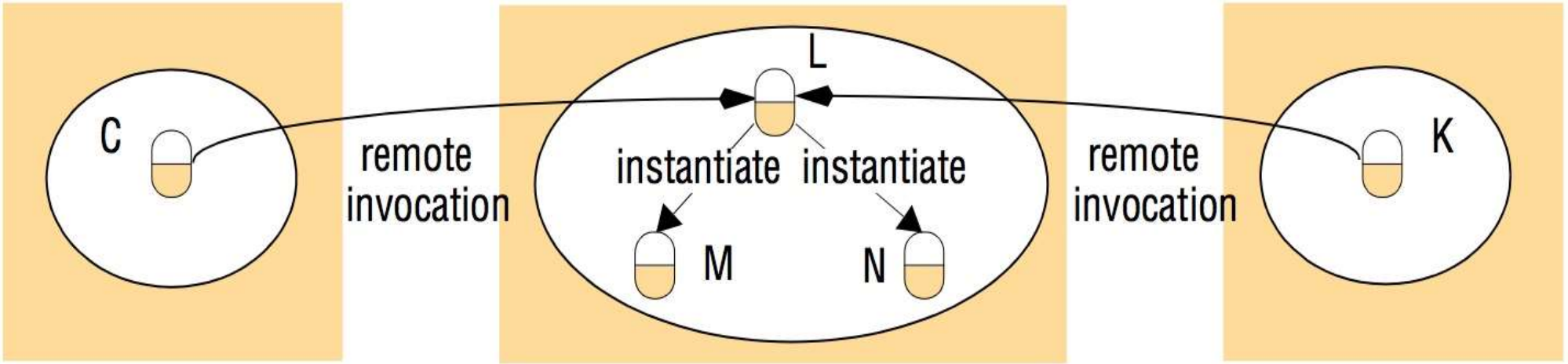


Figure 5.14
Instantiation of remote objects



5.2. Communication between distributed objects

- **Design Issues of RMI**
  - Choice of invocation semantics
  - Level of transparency that is desirable for RMI (self-read)

- **Local invocation semantics**: exactly-once
  - Every method is executed exactly once

- **RMI invocation semantics**:
  - **Maybe**: the remote method maybe executed once or not at all
  - **At least once:** invoker receives either a result, in which case invoker knows the method was executed at least once, or an exception informing no result was received
  - **At most once:** invoker receives either a result, in which case the invoker knows the method was executed exactly once, or an exception informing no result was received, in which case the method will have been executed either once or not at all

5.2. Communication between distributed objects

Request-reply protocol offers different choices of delivery guarantees:

- **Retry request message** – retransmit until reply is received or on server failure
- **Duplicate message filtering** – discard duplicates at server (seq #s or ReqID)
- **Retransmission of results:** Buffer result messages at server for retransmission
  - avoids redo of requests (even for idempotent ops)
    - History: record of transmitted messages

5.2. Communication between distributed objects

Fault tolerance measures			*Invocation semantics*
Retransmit request message	*Duplicate filtering*	*Re-execute procedure or retransmit reply*	
No	Not applicable	Not applicable	*Maybe*
Yes	No	Re-execute procedure	*At-least-once*
Yes	Yes	Retransmit reply	*At-most-once*

5.2. Communication between distributed objects

Invocation semantics: failure model

- *Maybe*, *At-least-once* and *At-most-once* can suffer from crash failures when the server containing the remote object fails.

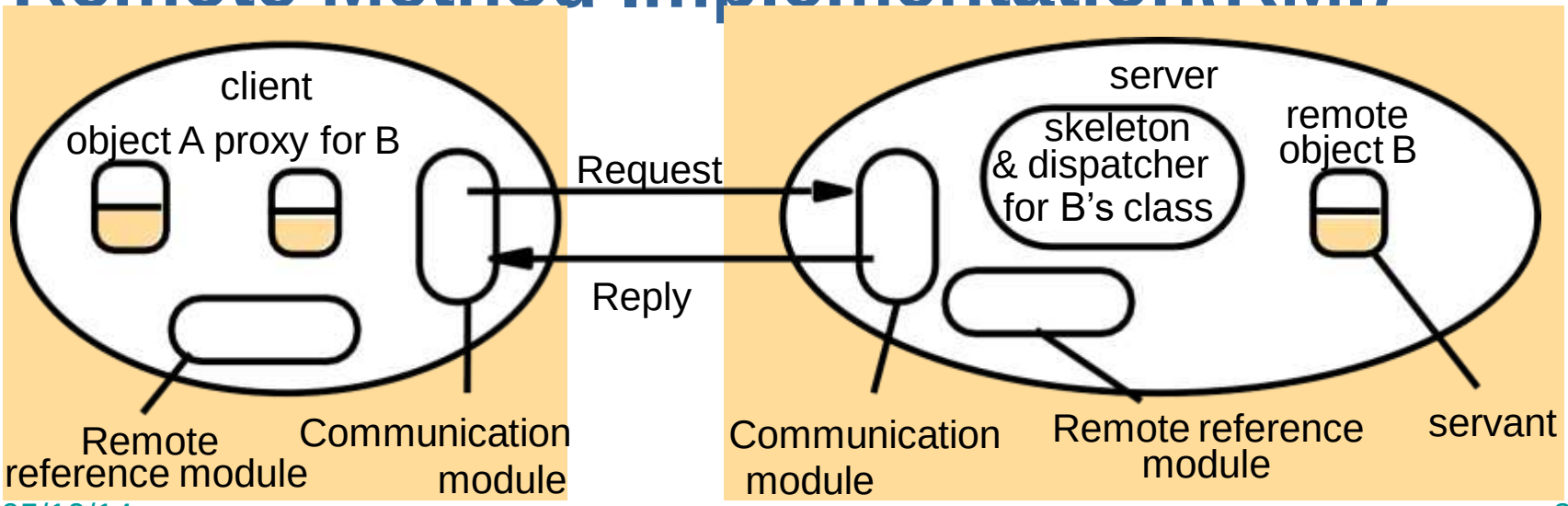
- ***Maybe*** – executed once or not. If no reply, the client does not know if method was executed or not
  - Suffers from omission failures if the invocation or result message is lost

5.2. Communication between distributed objects

Invocation semantics: failure model

- ***At-least-once*** **-** the client gets a result (and the method was executed at least once) or an exception (no result)
  - arbitrary failures. If the invocation message is retransmitted, the remote object may execute the method more than once, possibly causing wrong values to be stored or returned.
  - if *idempotent* operations are used, arbitrary failures will not occur
- ***At-most-once*** **-** the client gets a result (and the method was executed exactly once) or an exception (instead of a result, in which case, the method was executed once or not at all)
  - Java RMI

5.2. Communication between distributed objects
Remote Method Implementation(RMI)



RMI cont..

Implementation of RMI:

- **Communication module**: carry out request-reply protocol which transmits request and reply messages between client and server. It uses the message type ,request id and remote reference of the object to be invoked.
- **Remote reference module**: translating between local and remote object references and for creating remote object references.
- Uses remote object table (remote object ref. <-> local object ref.)

  The table includes
  - An entry for all remote objects held by the process Ex Object B entry
  - An entry for each local proxy Ex the proxy for B will be recorded in the table at the client

RMI(Cont..)

- **Servant:** an instance of a class which implements methods in remote interface, eventually handles the remote request

- **RMI software:** layer of software between (application level) and (communication & remote reference modules**)**

  - **Proxy:** local (client side) representative for remote (server side) object; one proxy for one remote obj. It hides the details of the remote object, marshalling of arguments, unmarshalling of results and sending and receiving of messages from the client.

  - **Dispatcher:** one for each class of a remote object. On receiving request, uses methodId to select matching method in the skeleton

  - **Skeleton:** one for each class of a remote object. Implements methods in remote interface but in diff.way
    - Unmarshal request; invoke servant; await result; marshal into reply; send

RMI(cont..)

Some other concepts:

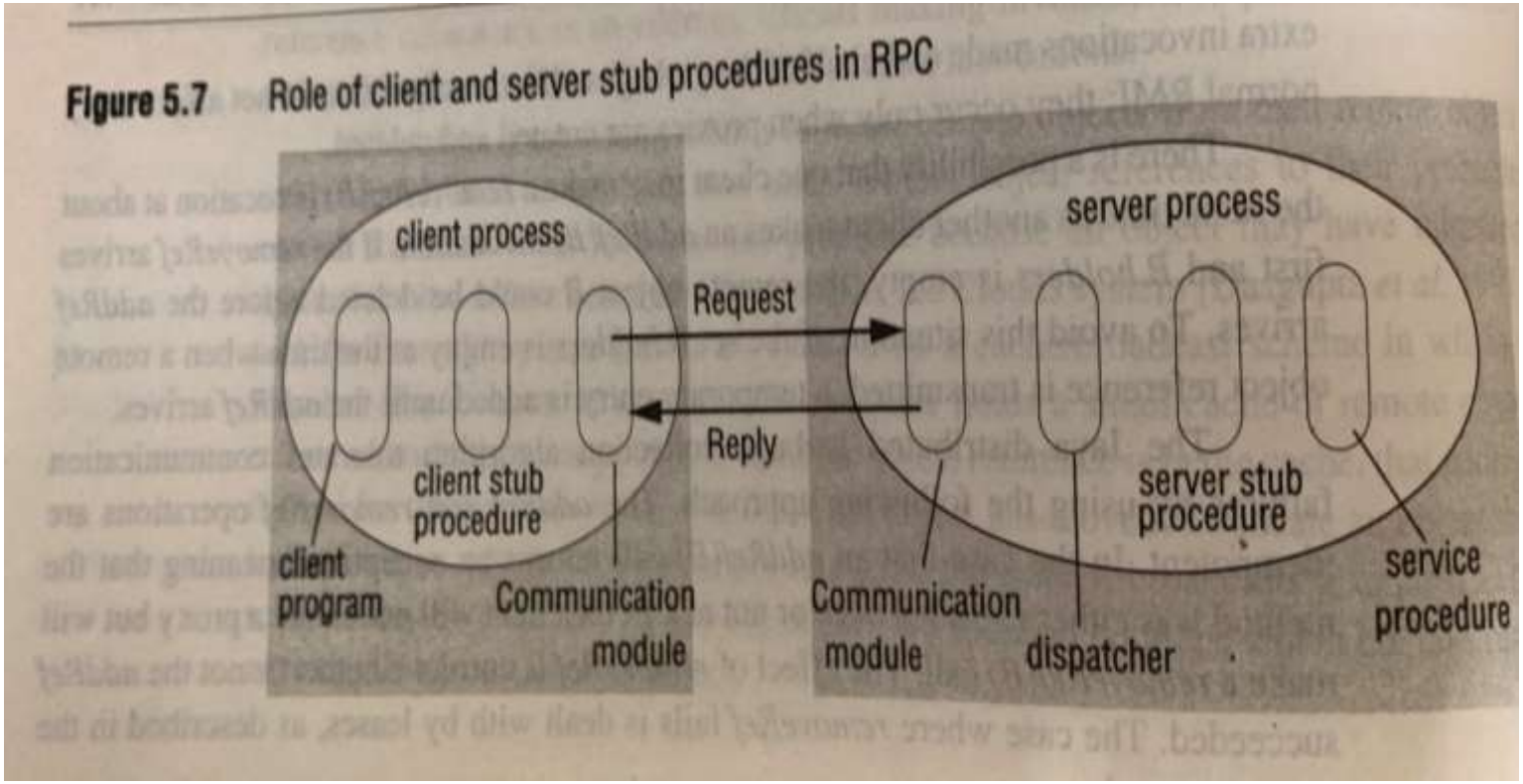
binder

- client programs require a way of obtaining a remote object reference. A binder is a separate service that maintains a table containing mappings from textual names to remote object references
- used by servers to register their remote objects by name
- used by clients to look them up
- e.g. Java binder: RMIregistry

Server and client programs

- **client program:** contain the classes of the proxies for all remote objects it will invoke; it can use binders to look up remote object references
- **server program:** contains the classes for the dispatchers and skeletons, together with implementations of the classes of all servants that it supports; it uses binders to register servants

Remote Procedure Call (RPC)



Remote Procedure Call(Cont..)

•Software is similar to that shown in RMI except that no remote reference model are required since procedure call is not concerned with objects and object references

•The client that accesses a service includes one **stub procedure** for each procedure in the service interface.The role of stub procedure is similar to the proxy method.It behaves like a local procedure o the client but instead of executing the call it marshals the process identifier and the arguments into request message which it sends via communication module to the server When reply message arrives it unmarshals the results.

•The server process contains a dispatcher together with one server stub procedure and one service procedure for each procedure in the service interface.The dispatcher selects one of the server stub procedures according to the procedure identifier in the request message.

RPC Cont..

A server stub procedure is like a skeleton method in that it unmarshals the arguments in the request message calls the service procedure and marshals the return values for the reply message.

The service procedure implements the procedure in the service interface