# Computer Science 425
# Distributed Systems

## *CS 425 / CSE 424 / ECE 428*

## Fall 2012

Indranil Gupta (Indy)

September 20, 2012

Lecture 8

**RPCs and Distributed Objects**

Reading: Section 4.3, parts of Chapter 5

# RMI/RPC - Motivation

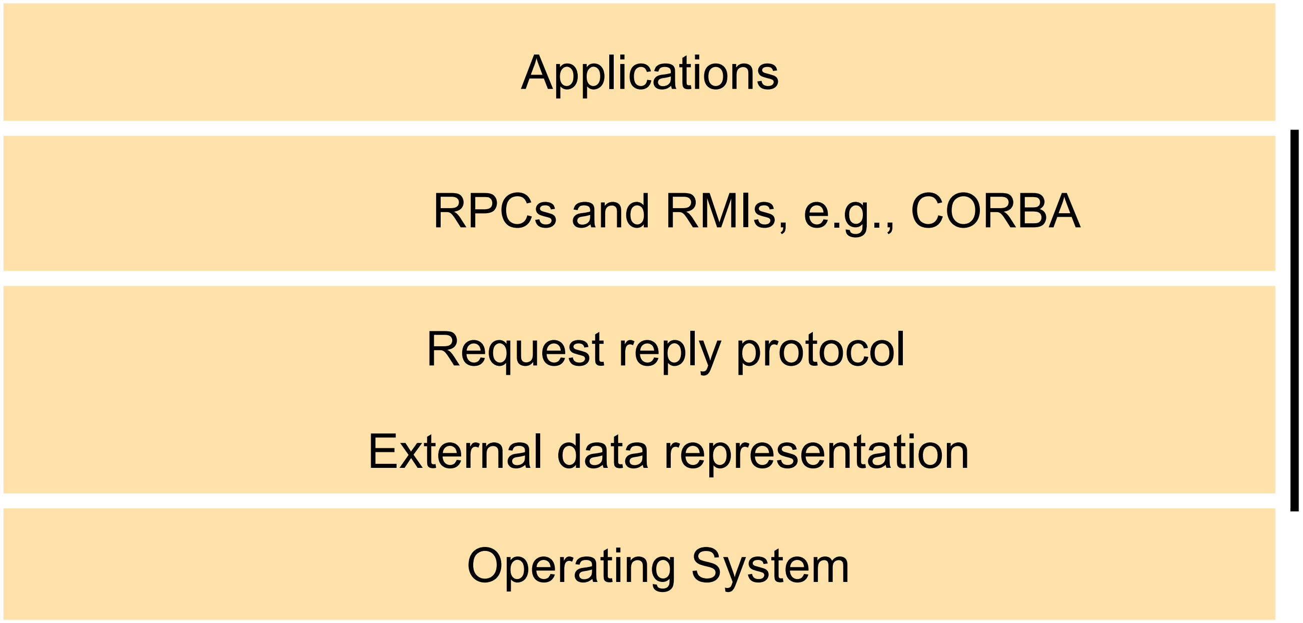
- You write a program where objects call each other
- Works well if the program runs on one process
- What if you split your objects across multiple processes?
- Can Object1's still call Object2.MethodA()?
- Why (not)?
- Solution
  - RMIs: Remote Method Invocations (Object-based)
  - RPCs: Remote Procedure Calls (non-Object-based)

❖ Access libraries of reusable code across hosts

❖ Pros

  - ❑ Supports code reuse
  - ❑ Standard interface, independent of applications and OS's

# *Middleware Layers*



Applications

RPCs and RMIs, e.g., CORBA

Request reply protocol

External data representation

Operating System

Middleware layers= *Provide support to the application*

Run at all servers @user level

RPC = Remote Procedure Call (Procedure = Function)
RMI=Remote Method Invocation
CORBA=Common Object Request Brokerage Architecture

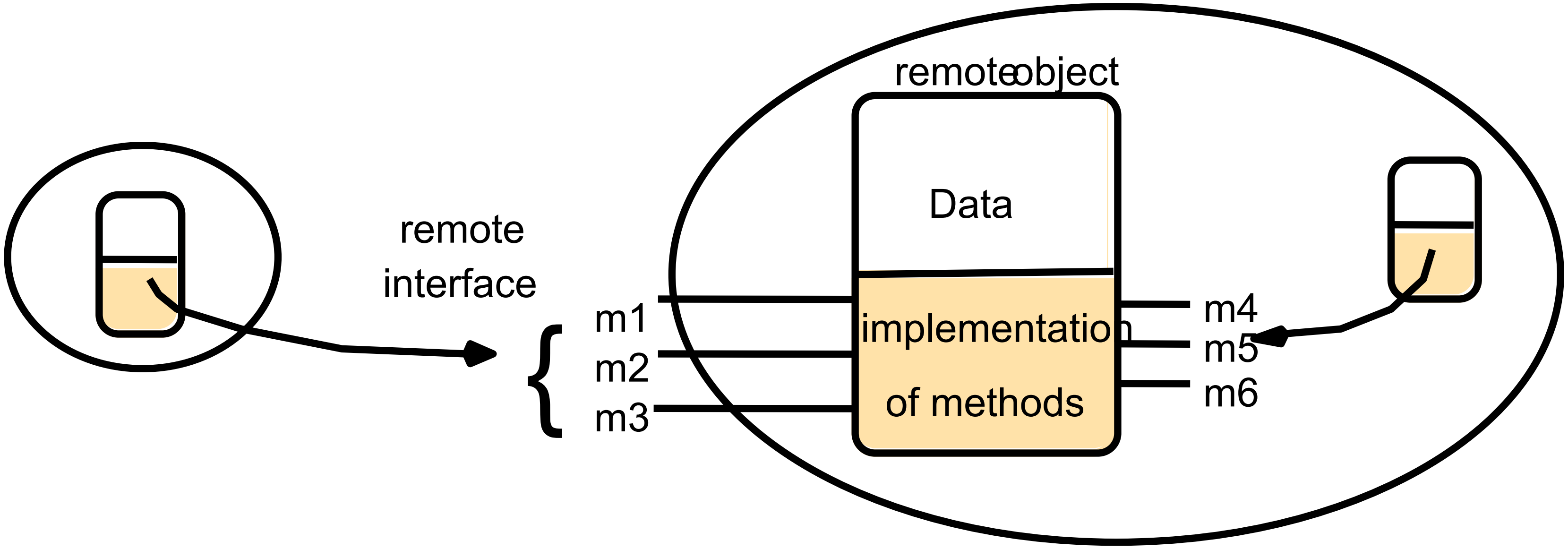
# *Local Objects*

- Within one process' address space
- Object
  - consists of a set of data and a set of methods.
  - E.g., C++ object, Java object.
- Object reference
  - an identifier via which objects can be accessed.
  - i.e., a *pointer* (e.g., virtual memory address within process)
- Interface
  - provides a definition of the signatures of a set of methods (i.e., the types of their arguments, return values, and exceptions) without specifying their implementation.

# *Remote Objects*

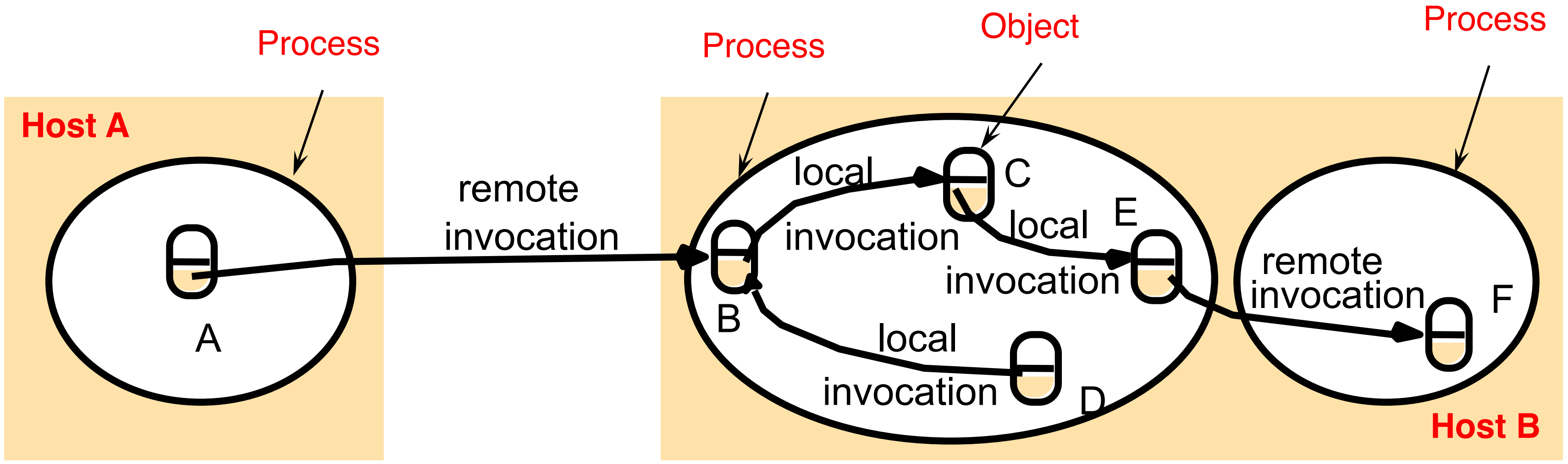
- **May cross multiple process' address spaces**
- **Remote method invocation**
  - **method invocations between objects *in different processes (processes may be on the same or different host)*.**
  - **Remote Procedure Call (RPC): procedure call between functions on different processes in non-object-based system**
- **Remote objects**
  - **objects that can receive remote invocations.**
- **Remote object reference**
  - **an identifier that can be used globally *throughout a distributed system* to refer to a particular unique remote object.**
- **Remote interface**
  - **Every remote object has a remote interface that specifies which of its methods can be invoked remotely. E.g., CORBA interface definition language (IDL).**

# A Remote Object and Its Remote Interface



Example Remote Object reference=(IP,port,objectnumber,signature,time)

# *Remote and Local Method Invocations*



Local invocation=between objects on same process.

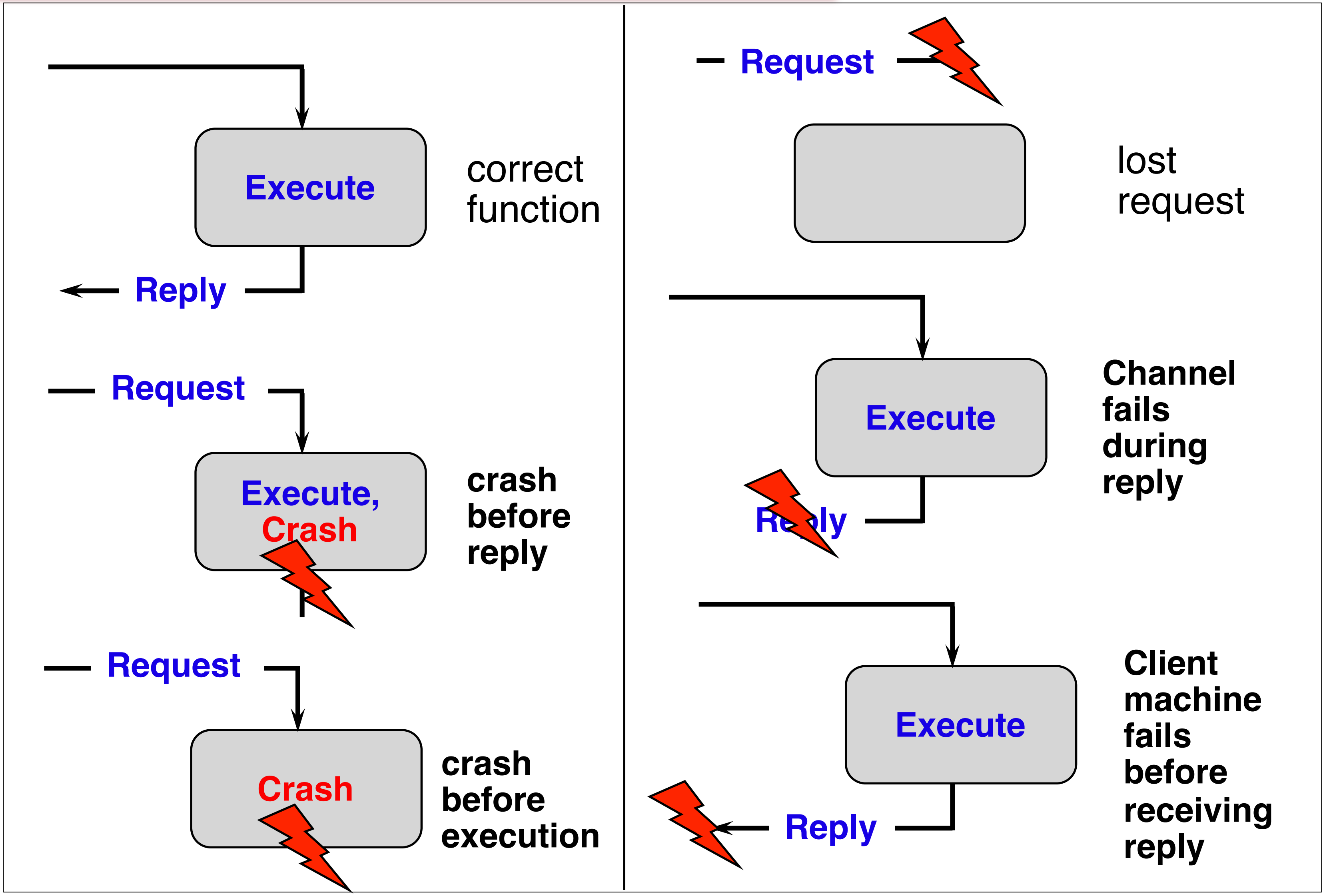
Has *exactly once* semantics

Remote invocation=between objects on different processes.

Ideally also want *exactly once* semantics for remote invocations

But difficult (why?)

# Failure Modes of RMI/RPC



(and if request is received more than once?)

# *Invocation Semantics*

Transparency=remote invocation has same behavior as local invocation
[Birrell and Nelson, inventors of RPC, 1984]
Very difficult to implement in asynchronous network…

Whether or not to retransmit the request message until either a reply is received or the server is assumed to be failed

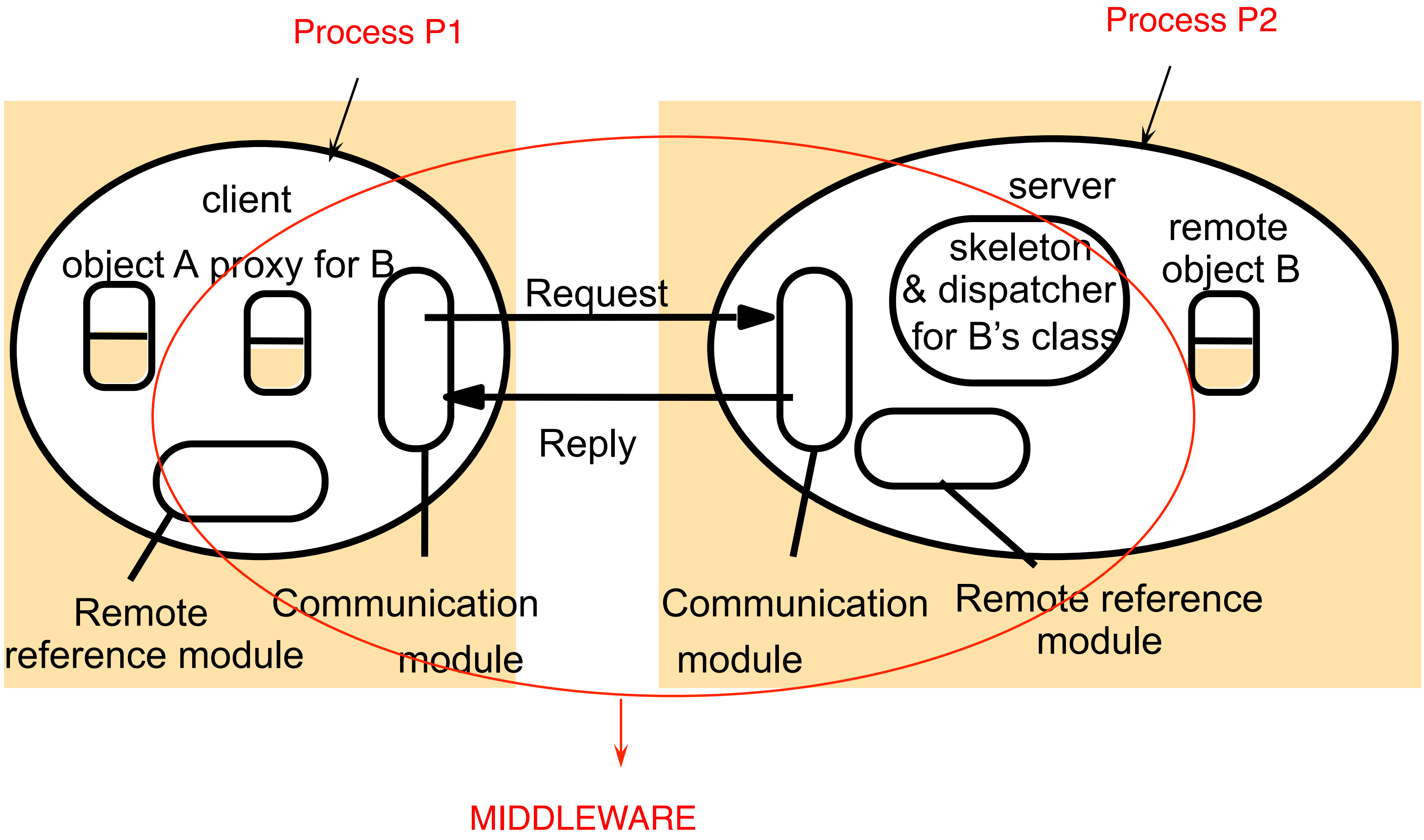
when retransmissions are used, whether to filter out duplicate requests at the server.

whether to keep a history of result messages to enable lost results to be retransmitted without re-executing the operations

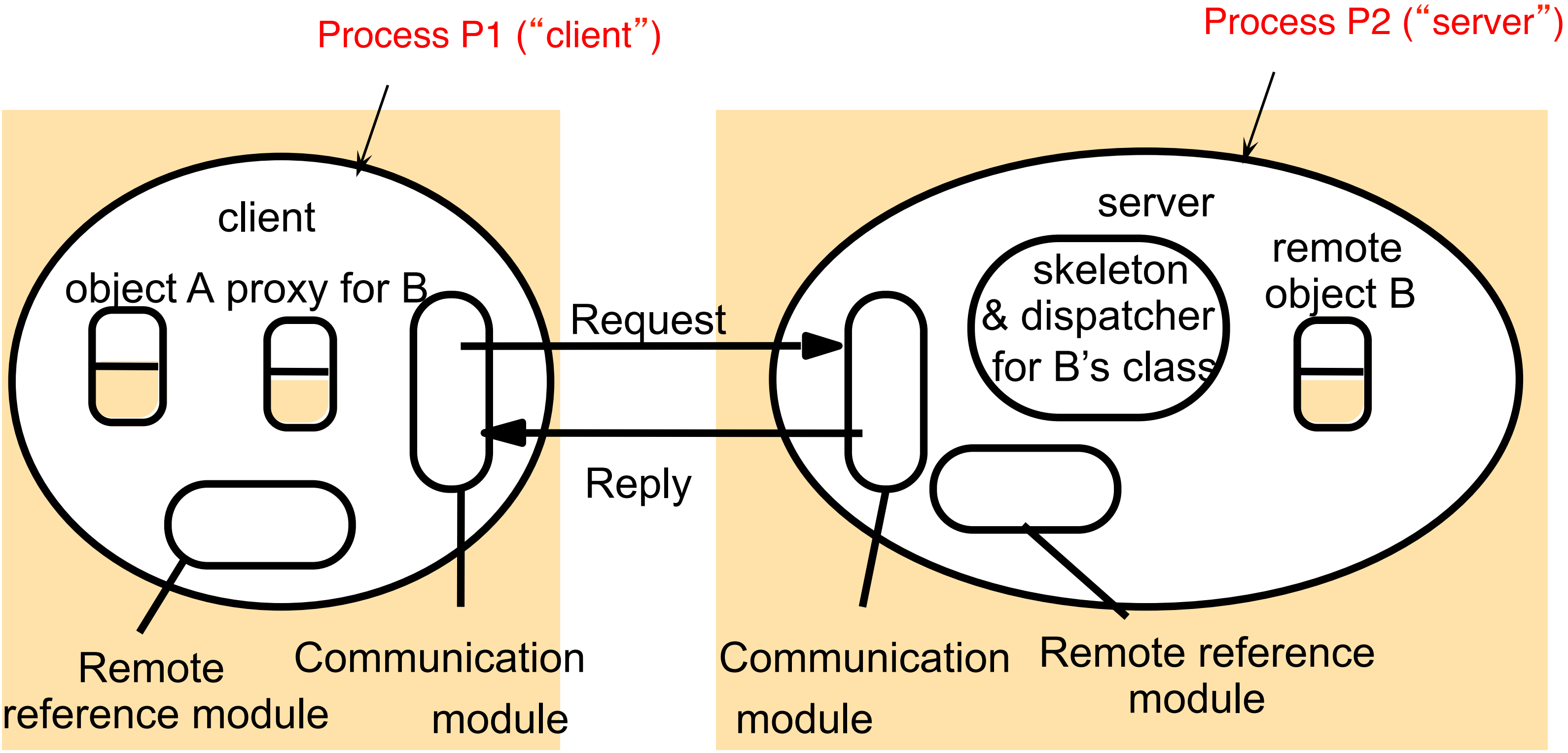
| | *Fault tolerance measures* | | | *Invocation semantics* |
|---|---|---|---|---|
| | *Retransmit request message* | *Duplicate filtering* | *Re-execute procedure or retransmit reply* | |
| CORBA → | No | Not applicable | Not applicable | *Maybe* |
| Sun RPC → | Yes | No | (ok for *idempotent* operations)<br>→ Re-execute procedure | *At-least-once* |
| Java RMI, CORBA → | Yes | Yes | Retransmit old reply | *At-most-once* |

Idempotent=same result if applied repeatedly, w/o side effects

# *Proxy and Skeleton in Remote Method Invocation*



# *Proxy and Skeleton in Remote Method Invocation*



# Proxy

- **Is responsible for making RMI transparent to clients by behaving like a local object to the invoker.**
  - **The proxy *implements* (Java term, not literally) the methods in the interface of the remote object that it represents. But,…**
- **Instead of executing an invocation, the proxy forwards it to a remote object.**
  - **On invocation, a method of the proxy *marshals* the following into a request message: (i) a reference to the target object, (ii) its own method id and (iii) the argument values. Request message is sent to the target, then proxy awaits the reply message, *un-marshals* it and returns the results to the invoker.**
  - **Invoked object unmarshals arguments from request message, and when done marshals return values into reply message.**

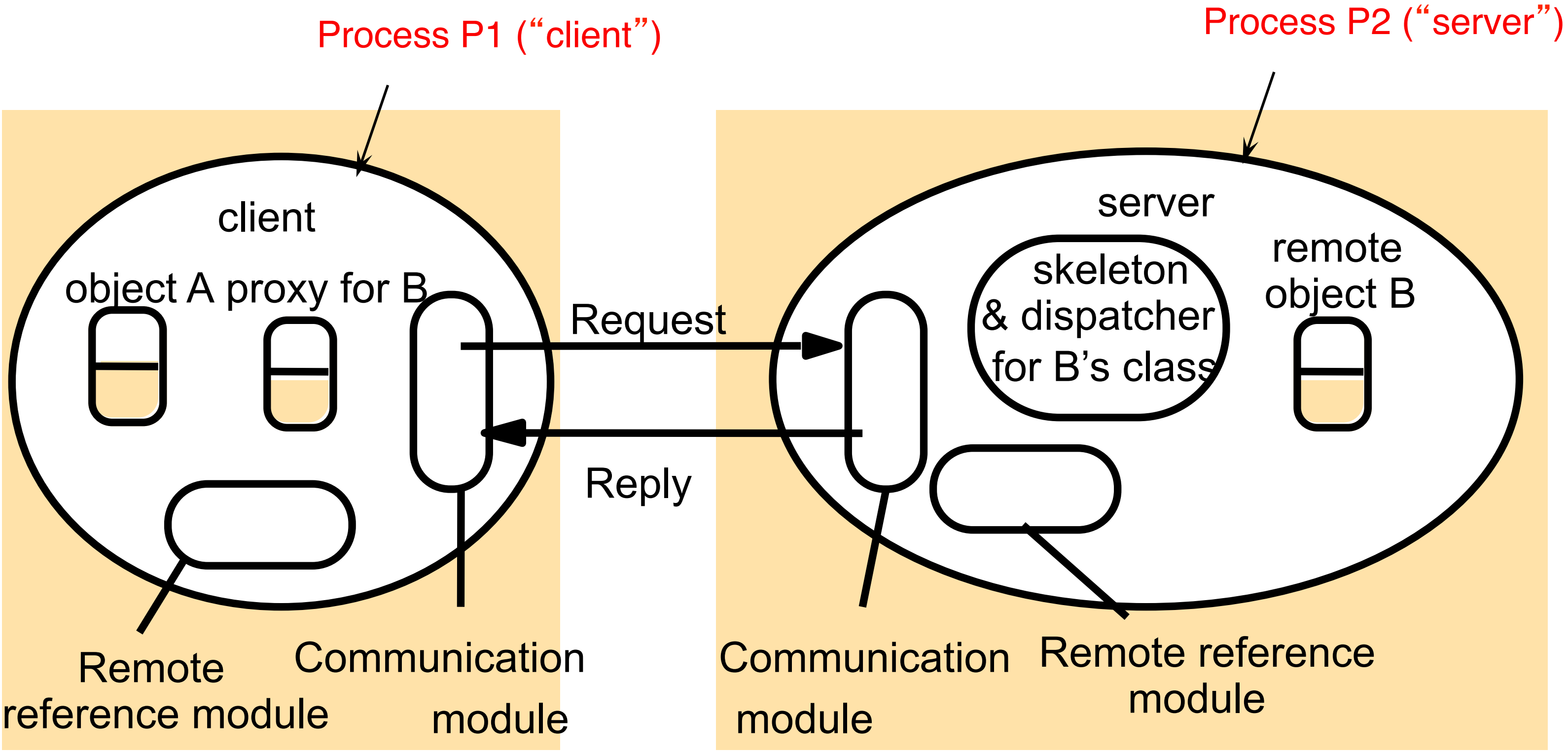
# Marshalling & Unmarshalling

- A Windows client sends an RMI to a Unix/Mac server
  - won’t work because Windows is little endian while Unix/Mac is big-endian
- External data representation: an agreed, platform-independent, standard for the representation of data structures and primitive values.
  - CORBA Common Data Representation (CDR)
  - Allows a Windows client (little endian) to interact with a Unix server or Mac server (big endian).
- Marshalling: the act of taking a collection of data items (platform dependent) and assembling them into the external data representation (platform independent).
- Unmarshalling: the process of disassembling data that is in external data representation form, into a locally interpretable form.

# *Remote Reference Module*

- Is responsible for translating between local and remote object references and for creating remote object references.
- Has a *remote object table*
  - An entry for each remote object held by any process. E.g., B at P2.
  - An entry for each local proxy. E.g., proxy-B at P1.
- When a new remote object is seen by the remote reference module, it creates a remote object reference and adds it to the table.
- When a remote object reference arrives in a request or reply message, the remote reference module is asked for the corresponding local object reference, which may refer to either a proxy or to a local object.
- In case the remote object reference is not in the table, the RMI software creates a new proxy and asks the remote reference module to add it to the table.

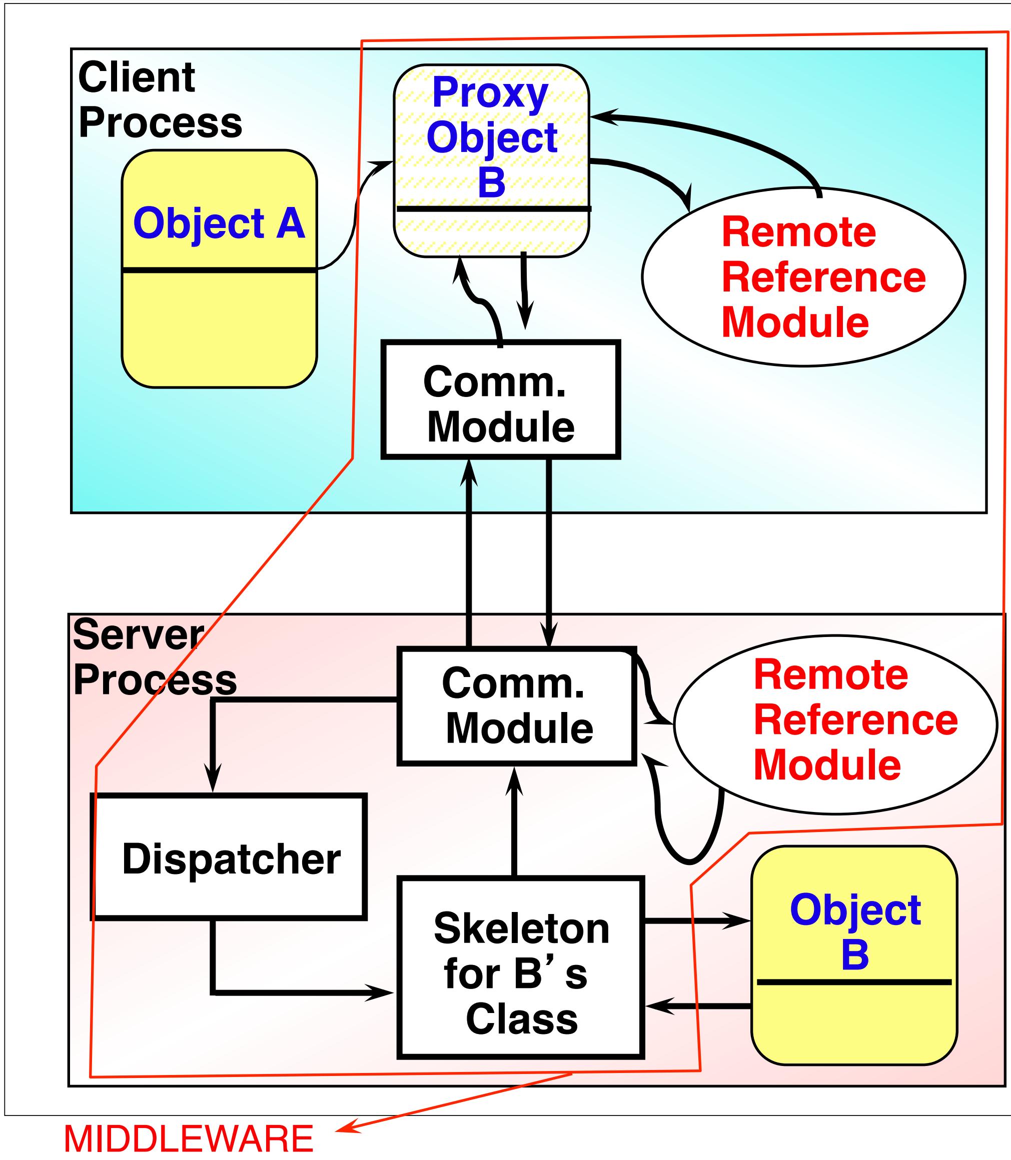
# Proxy and Skeleton in Remote Method Invocation



# *What about Server Side? Dispatcher and Skeleton*

- **Each process has one dispatcher. And a skeleton for each local object (actually, for the class).**
- **The dispatcher receives all request messages from the communication module.**
  - **For the request message, it uses the method id to select the appropriate method in the appropriate skeleton, passing on the request message.**
- **Skeleton "implements" the methods in the remote interface.**
  - **A skeleton method un-marshals the arguments in the request message and invokes the corresponding method in the local object (the actual object).**
  - **It waits for the invocation to complete and marshals the result, together with any exceptions, into a reply message.**

# Summary of Remote Method Invocation (RMI)



Proxy object is a hollow container of Method names.

Remote Reference Module translates between local and remote object references.

Dispatcher sends the request to Skeleton Object

Skeleton unmarshals parameters, sends it to the object, & marshals the results for return

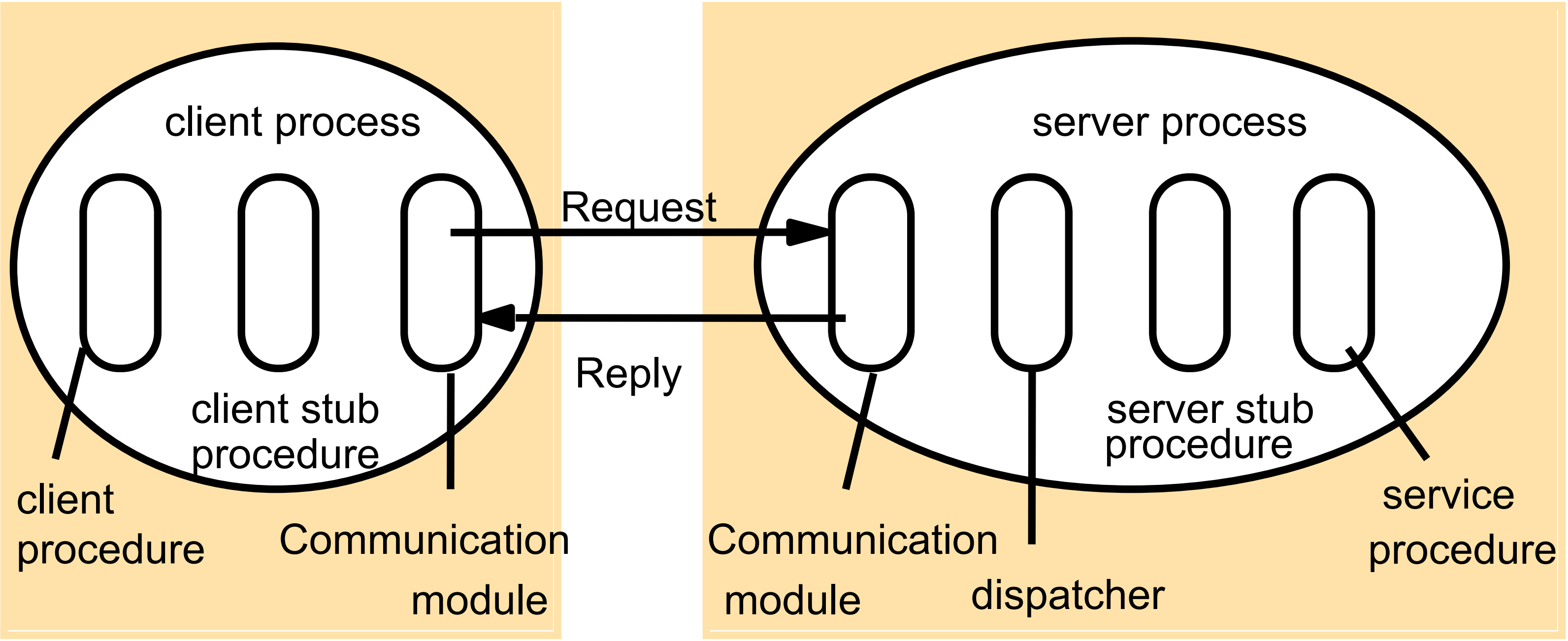
# *Generation of Proxies, Dispatchers and Skeletons*

- **Programmer only writes object implementations and interfaces**
- **Proxies, Dispatchers and Skeletons generated automatically from the specified interfaces**
- **In CORBA, programmer specifies interfaces of remote objects in CORBA IDL; then, the interface compiler <u>automatically</u> generates code for proxies, dispatchers and skeletons.**
- **In Java RMI**
  - **The programmer defines the set of methods offered by a remote object as a Java interface implemented in the remote object.**
  - **The Java RMI compiler generates the proxy, dispatcher and skeleton classes from the class of the remote object.**

# *Remote Procedure Call (RPC)*

- **Similar to RMIs, but for non-OO/non-object-based scenarios**
- **Procedure call that crosses process boundary**
- **Client process calls for invocation of a procedure at the server process.**
  - □ **Semantics are similar to RMIs – at least once, at most once, maybe**
  - □ **Format of the message is standard, uses request-reply**

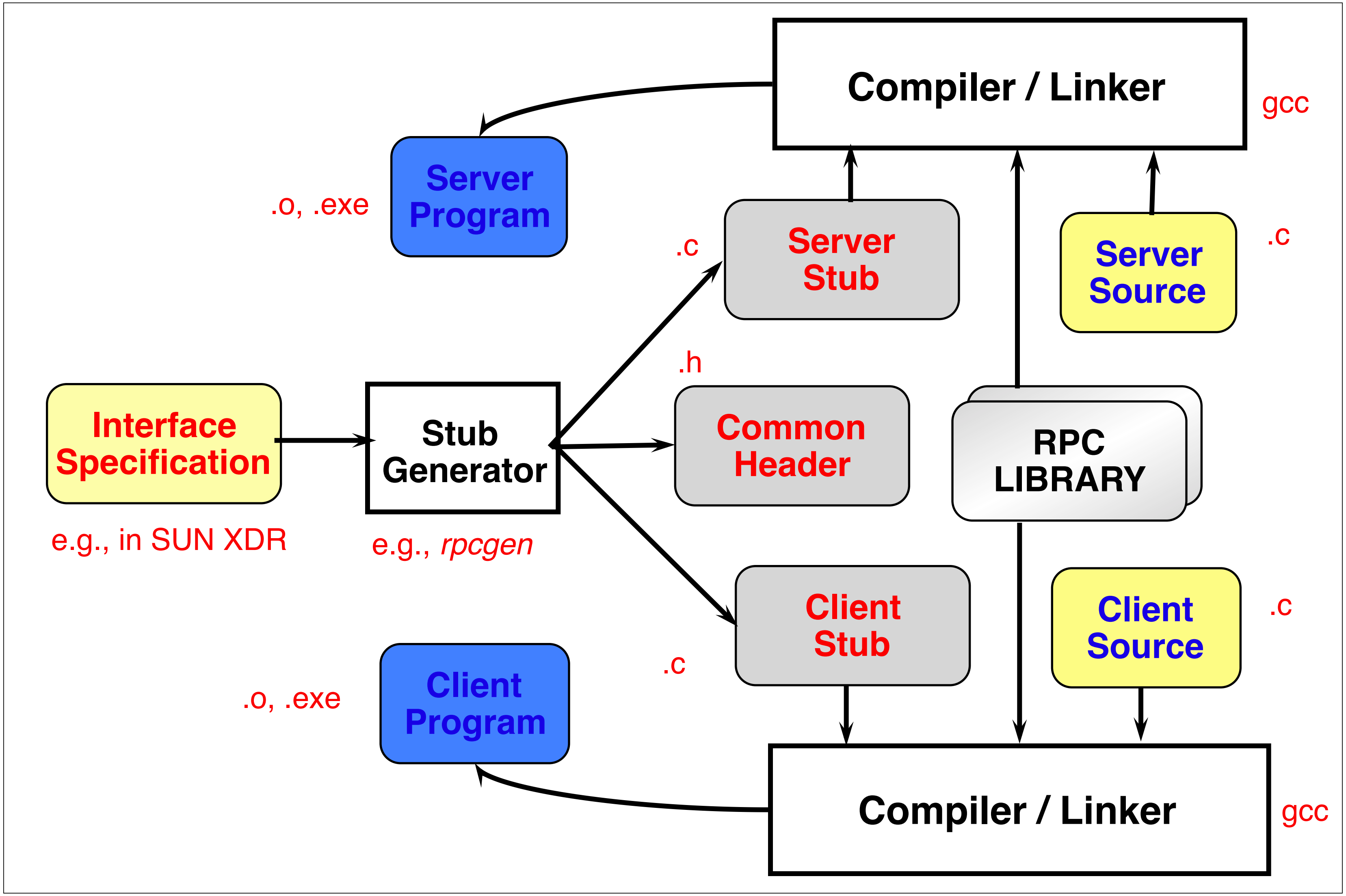
# Client and Server Stub Procedures in RPC



# Stubs

- Stubs are generated automatically from interface specifications.
- Stubs hide details of (un)marshalling from application programmer & library code developer.
- Client Stubs perform marshalling into request messages and unmarshalling from reply messages
- Server Stubs perform unmarshalling from request messages and marshalling into reply messages
- Stubs also take care of invocation

# The Stub Generation Process



# Announcements

- Next Friday Sep 28 – Tours of Blue Waters Datacenter!
  - Signup sheet link will be posted soon on Piazza
- HW2 released soon.
- MP2 already released.
- Next week: P2P systems!

# Optional Slides

# Files Interface in Sun XDR

Available with most Sun systems, and NFS

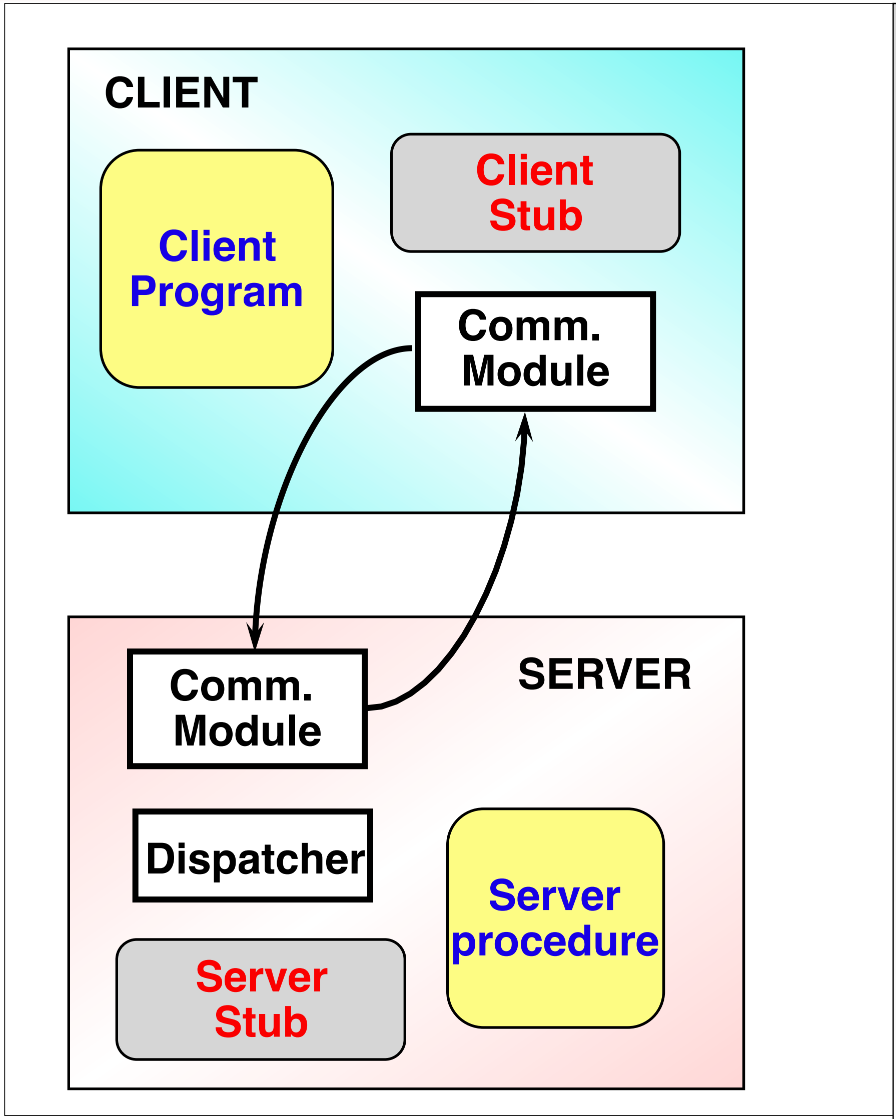
```
const MAX = 1000;
typedef int FileIdentifier;
typedef int FilePointer;
typedef int Length;
struct Data {
      int length;
      char buffer[MAX];
};
struct writeargs {
      FileIdentifier f;
      FilePointer position;
      Data data;
};
```

```
struct readargs {
      FileIdentifier f;
      FilePointer position;
      Length length;
};

program FILEREADWRITE {
   version VERSION {
      void WRITE(writeargs)=1;
      Data READ(readargs)=2;
   }=2;        Version number
} = 9999;      Program number
```

Only one argument allowed
Can specify as struct

# Finding RPCs



## Finding An RPC:

RPCs live on specific hosts at specific ports.

Port mapper on the host maps from RPC name to port#

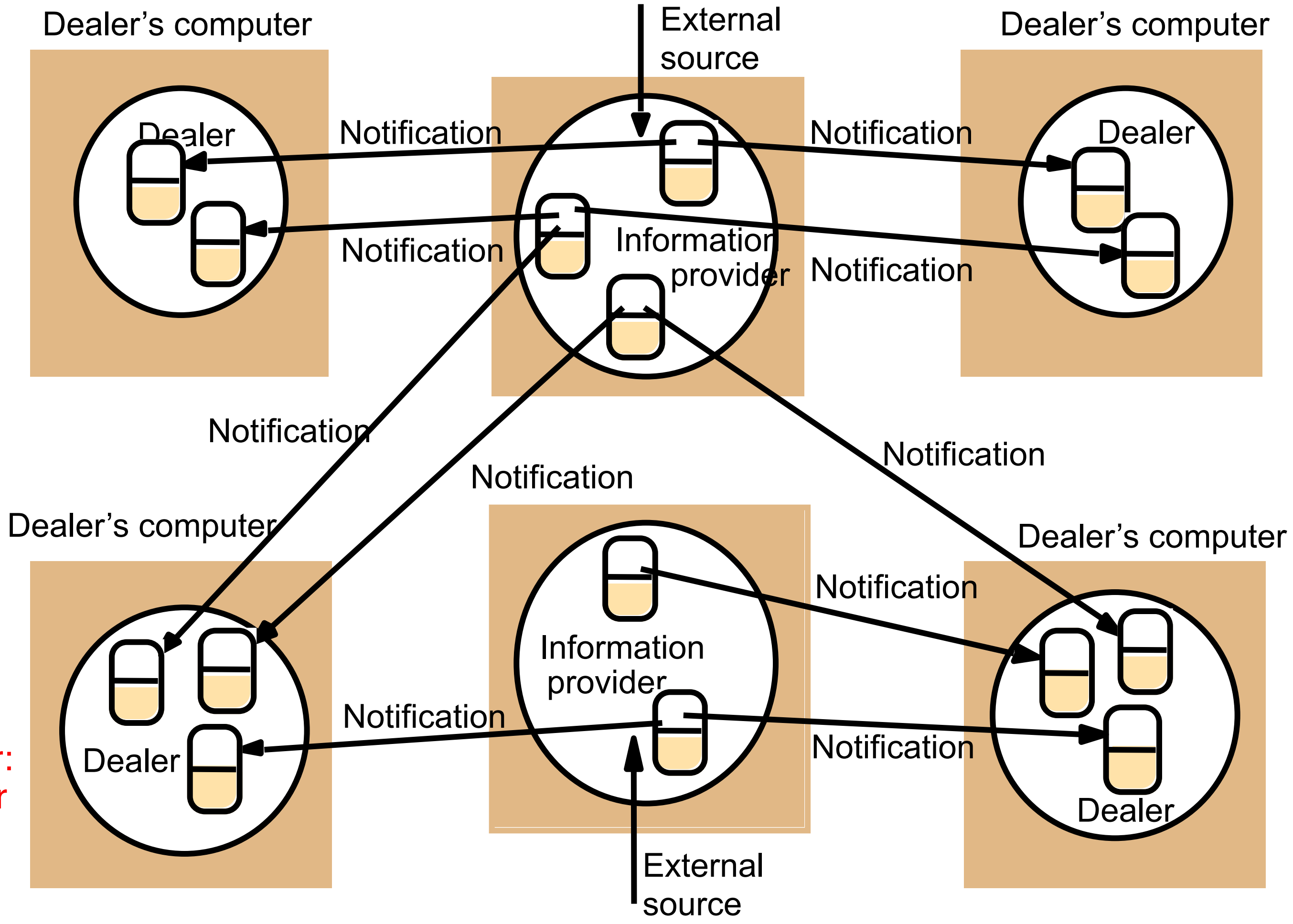
When a server process is initialized, it registers its RPCs (handle) with the port mapper  on the server

A client first connects to port mapper (daemon on standard port) to get this handle

The call to RPC is then made by connecting to the corresponding port

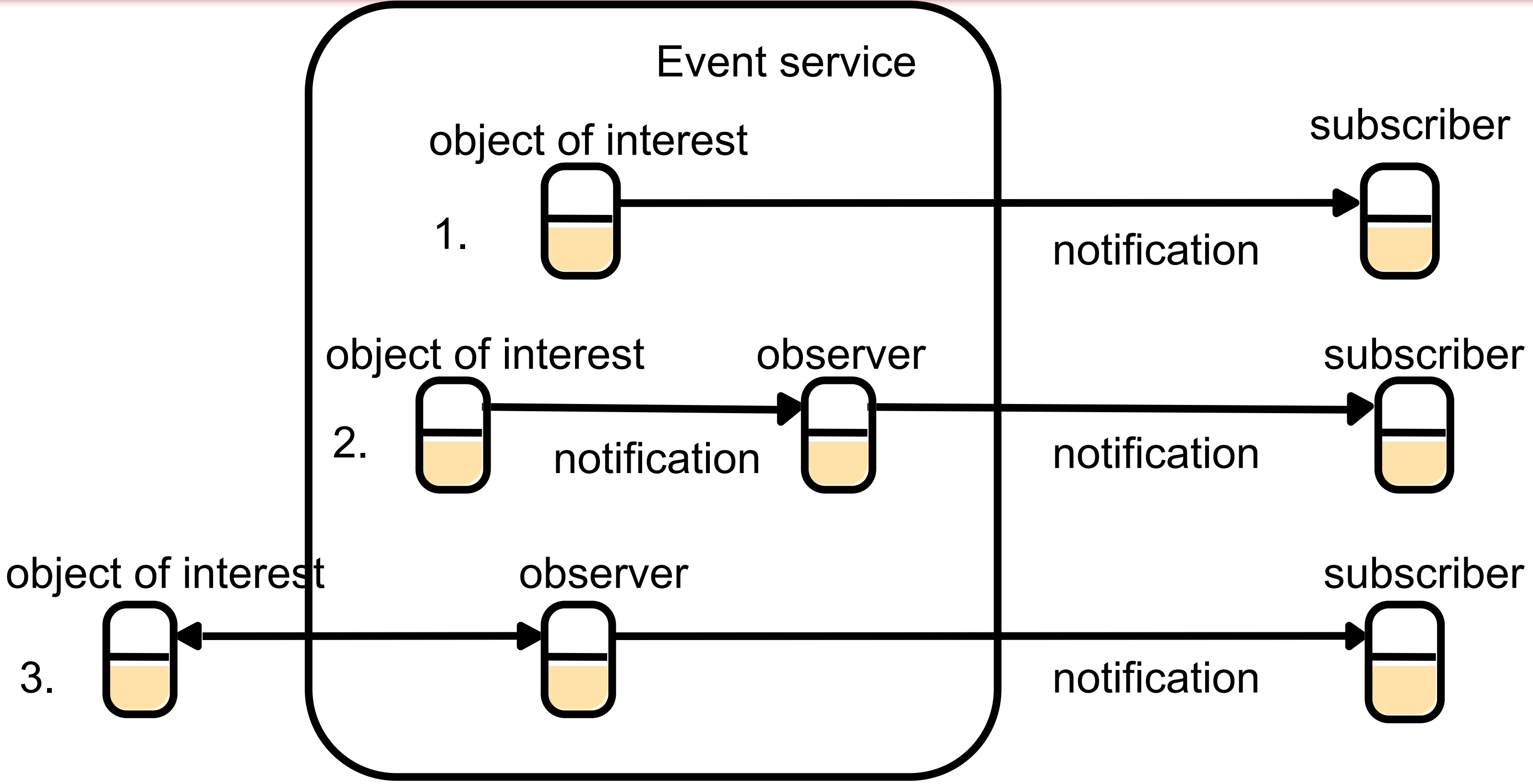
# Dealing Room System

*[Publish-Subscribe System]*
*e.g, stock market*



At each dealer:
One object per stock type of interest

# *Architecture for Distributed Event Notification*



# *Binder and Activator*

- **Binder: A separate service that maintains a table containing mappings from textual names to remote object references. (sort of like DNS, but for the specific middleware)**
  - **Used by servers to register their remote objects by name. Used by clients to look them up. E.g., Java RMI Registry, CORBA Naming Svc.**
- **Activation of remote objects**
  - **A remote object is *active* when it is available for invocation within a running process.**
  - **A *passive* object consists of (i) implementation of its methods; and (ii) its state in the marshalled form (a form in which it is shippable).**
  - ***Activation* creates a new instance of the class of a passive object and initializes its instance variables. It is called on-demand.**
  - **An *activator* is responsible for**
    - » **Registering passive objects at the binder**
    - » **Starting named server processes and activating remote objects in them.**
    - » **Keeping track of the locations of the servers for remote objects it has already activated**
  - **E.g., Activator=Inetd, Passive Object/service=FTP (invoked on demand)**

# Etc.

- Persistent Object = an object that survives between simultaneous invocation of a process. E.g., Persistent Java, PerDIS, Khazana.
- If objects migrate, may not be a good idea to have remote object reference=(IP,port,…)
  - Location service= maps a remote object reference to its likely current location
  - Allows the object to migrate from host to host, without changing remote object reference
  - Example: Akamai is a location service for web objects. It “migrates” web objects using the DNS location service