CSD511 – Distributed Systems

分散式系統

Chapter 5 Distributed Objects and Remote Invocation

吳俊興

國立高雄大學 資訊工程學系

Chapter 5 Distributed Objects and Remote Invocation

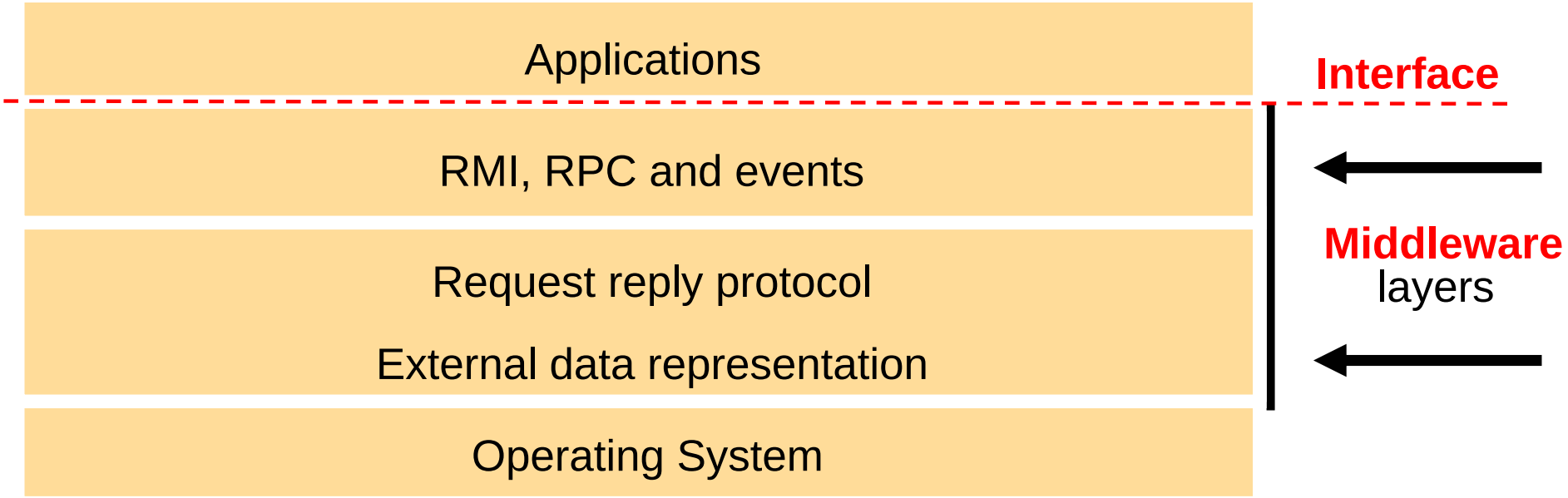
5.1 Introduction

- Programming Models for Distributed Communications
  - Remote Procedure Calls – Client programs call procedures in server programs
  - Remote Method Invocation – Objects invoke methods of objects on distributed hosts
  - Event-based Programming Model – Objects receive notice of events in other objects in which they have interest

Middleware

- Middleware: software that allows a level of programming beyond processes and message passing
  - Uses protocols based on messages between processes to provide its higher-level abstractions such as remote invocation and events
  - Supports location transparency
  - Usually uses an ***interface definition language (IDL)*** to define interfaces

Figure 5.1 Middleware Layer



Interfaces

- Interfaces in Programming Languages
  - Current PL allow programs to be developed as a set of modules that communicate with each other. Permitted interactions between modules are defined by ***interfaces***
  - A specified interface can be implemented by different modules without the need to modify other modules using the interface
- Interfaces in Distributed Systems
  - When modules are in different processes or on different hosts there are limitations on the interactions that can occur. Only actions with parameters that are fully specified and understood can communicate effectively to request or provide services to modules in another process
  - A service interface allows a client to request and a server to provide particular services
  - A remote interface allows objects to be passed as arguments to and results from distributed modules
- Object Interfaces
  - An interface defines the signatures of a set of methods, including arguments, argument types, return values and exceptions. Implementation details are not included in an interface. A class may implement an interface by specifying behavior for each method in the interface. Interfaces do not have constructors

5.2 Communication between Distributed Objects

Topics to be addressed

- The object model
  - Object references
  - Interfaces
  - Actions
  - Exceptions
  - Garbage collection
- Distributed objects
- The distributed object model
- Design issues
- Implementation
- Distributed garbage collection

5.2.1 The Object Model

Five Parts of the Object Model

- An object-oriented program consists of a collection of interacting objects
  - Objects consist of a set of data and a set of methods
  - In DS, object's data should be accessible only via methods

•Object References

- Objects are accessed by object references
- Object references can be assigned to variables, passed as arguments, and returned as the result of a method
- Can also specify a method to be invoked on that object

•Interfaces

- Provide a definition of the signatures of a set of methods without specifying their implementation
- Define types that can be used to declare the type of variables or of the parameters and return values of methods

The Object Model (Cont.)

- Actions
  - Objects invoke methods in other objects
  - An invocation can include additional information as arguments to perform the behavior specified by the method
  - Effects of invoking a method
    1. The state of the receiving object may be changed
    2. A new object may be instantiated
    3. Further invocations on methods in other objects may occur
    4. An exception may be generated if there is a problem encountered
- Exceptions
  - Provide a clean way to deal with unexpected events or errors
  - A block of code can be defined to *throw* an exception when errors or unexpected conditions occur. Then control passes to code that *catch*es the exception
- Garbage Collection
  - Provide a means of freeing the space that is no longer needed
  - Java (automatic), C++ (user supplied)

5.2.2 Distributed Objects

- Physical distribution of objects into different processes or computers in a distributed system
  - Object state consists of the values of its instance variables
  - Object methods invoked by remote method invocation (RMI)
  - Object encapsulation: object state accessed only by the object methods
- Usually adopt the client-server architecture
  - Basic model
    - Objects are managed by servers and
    - Their clients invoke their methods using RMI
  - Steps
    1. The client sends the RMI request in a message to the server
    2. The server executes the invoked method of the object
    3. The server returns the result to the client in another message
  - Other models
    - Chains of related invocations: objects in servers may become clients of objects in other servers
    - Object replication: objects can be replicated for fault tolerance and performance
    - Object migration: objects can be migrated to enhancing performance and availability

5.2.3 The Distributed Object Model

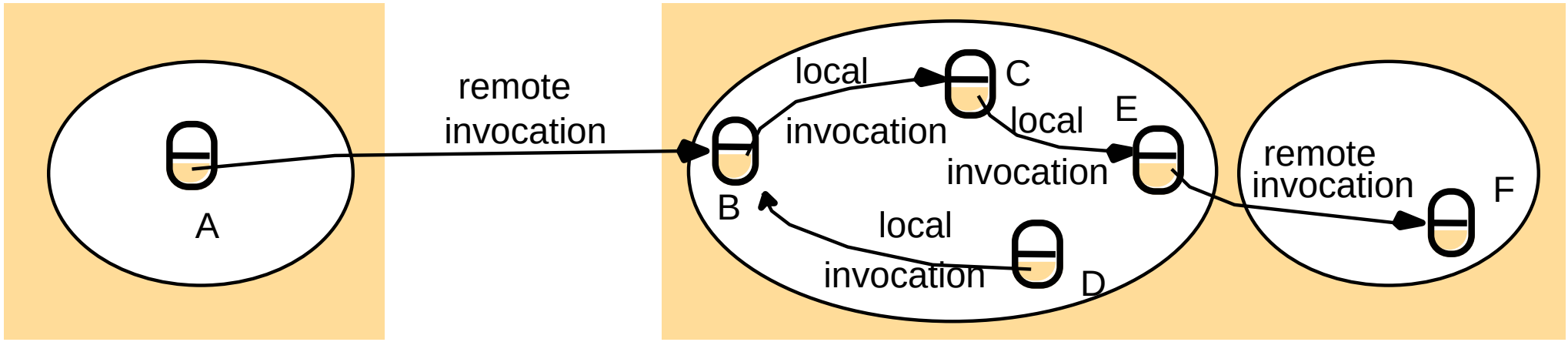


Figure 5.3 Remote and local method invocations

Two fundamental concepts: Remote Object Reference and Remote Interface

- Each process contains objects, some of which can receive remote invocations are called *remote objects* (B, F), others only local invocations
- Objects need to know the *remote object reference* of an object in another process in order to invoke its methods, called *remote method invocations*
- Every remote object has a *remote interface* that specifies which of its methods can be invoked remotely

Five Parts of Distributed Object Model

- Remote Object References
  - accessing the remote object
  - identifier throughout a distributed system
  - can be passed as arguments
- Remote Interfaces
  - specifying which methods can be invoked remotely
  - name, arguments, return type
  - Interface Definition Language (IDL) used for defining remote interface

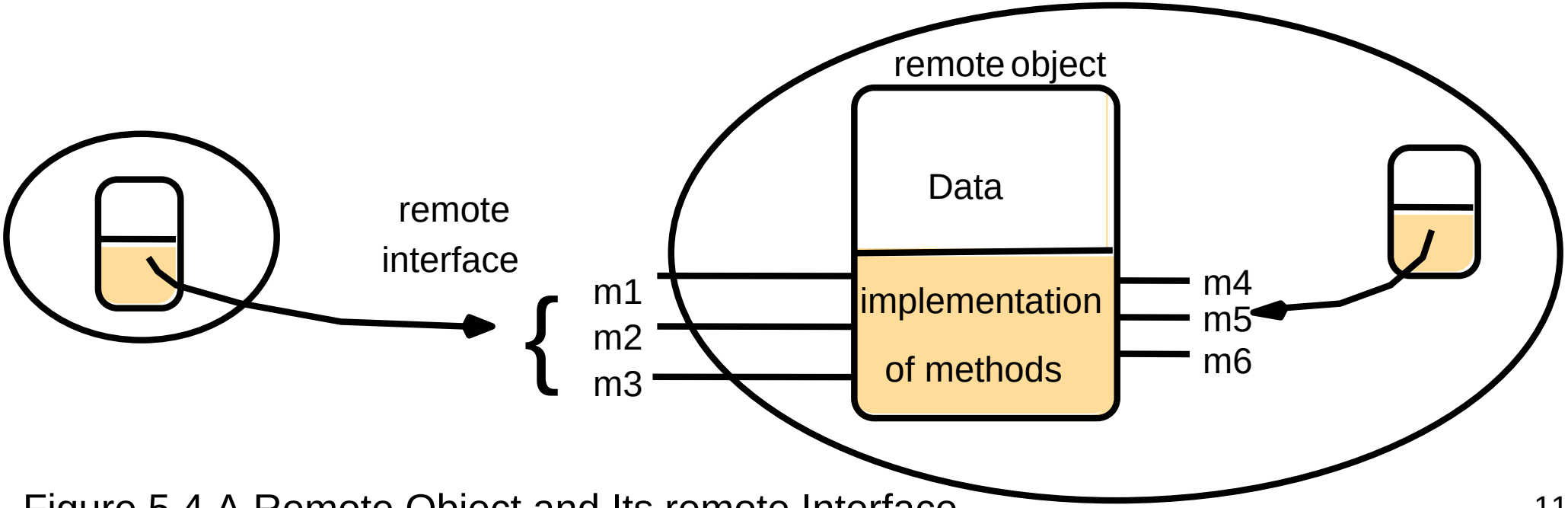


Figure 5.4 A Remote Object and Its remote Interface

Five Parts of Distributed Object Model (cont.)

- Actions
  - An action initiated by a method invocation may result in further invocations on methods in other objects located in difference processes or computers
  - Remote invocations could lead to the instantiation of new objects, ie. objects M and N of Figure 5.5
- Exceptions
  - More kinds of exceptions: i.e. timeout exception
    - RMI should be able to raise exceptions such as timeouts that are due to distribution as well as those raised during the execution of the method invoked
- Garbage Collection
    - Distributed garbage collections is generally achieved by cooperation between the existing local garbage collector and an added module that carries out a form of distributed garbage collection, usually based on reference counting

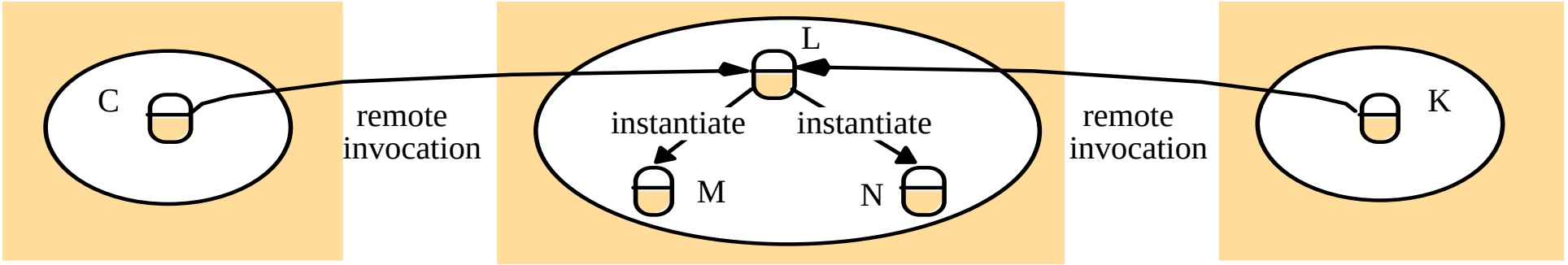


Figure 5.5 Instantiating Remote Objects

5.2.4 Design Issues for RMI

- Two design issues that arise in extension of local method invocation for RMI
  - The choice of invocation semantics
    - Although local invocations are executed exactly once, this cannot always be the case for RMI due to transmission error
      - Either request or reply message may be lost
      - Either server or client may be crashed
  - The level of transparency
    - Make remote invocation as much like local invocation as possible

RMI Design Issues: Invocation Semantics

- Error handling for delivery guarantees
  - Retry request message: whether to retransmit the request message until either a reply is received or the server is assumed to have failed
  - Duplicate filtering: when retransmissions are used, whether to filter out duplicate requests at the server
  - Retransmission of results: whether to keep a history of result messages to enable lost results to be retransmitted without re-executing the operations
- Choices of invocation semantics
  - Maybe: the method executed once or not at all (no retry nor retransmit)
  - At-least-once: the method executed **at least** once
  - At-most-once: the method executed **exactly** once

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*

Figure 5.6 Invocation semantics: choices of interest

RMI Design Issues: Transparency

- Transparent remote invocation: like a local call
  - marshalling/unmarshalling
  - locating remote objects
  - accessing/syntax
- Differences between local and remote invocations
  - latency: a remote invocation is usually several order of magnitude greater than that of a local one
  - availability: remote invocation is more likely to fail
  - errors/exceptions: failure of the network? server? hard to tell
    - syntax might need to be different to handle different local vs remote errors/exceptions (e.g. Argus)
  - consistency on the remote machine:
    - Argus: incomplete transactions, abort, restore states [as if the call was never made]

5.2.5 Implementation of RMI

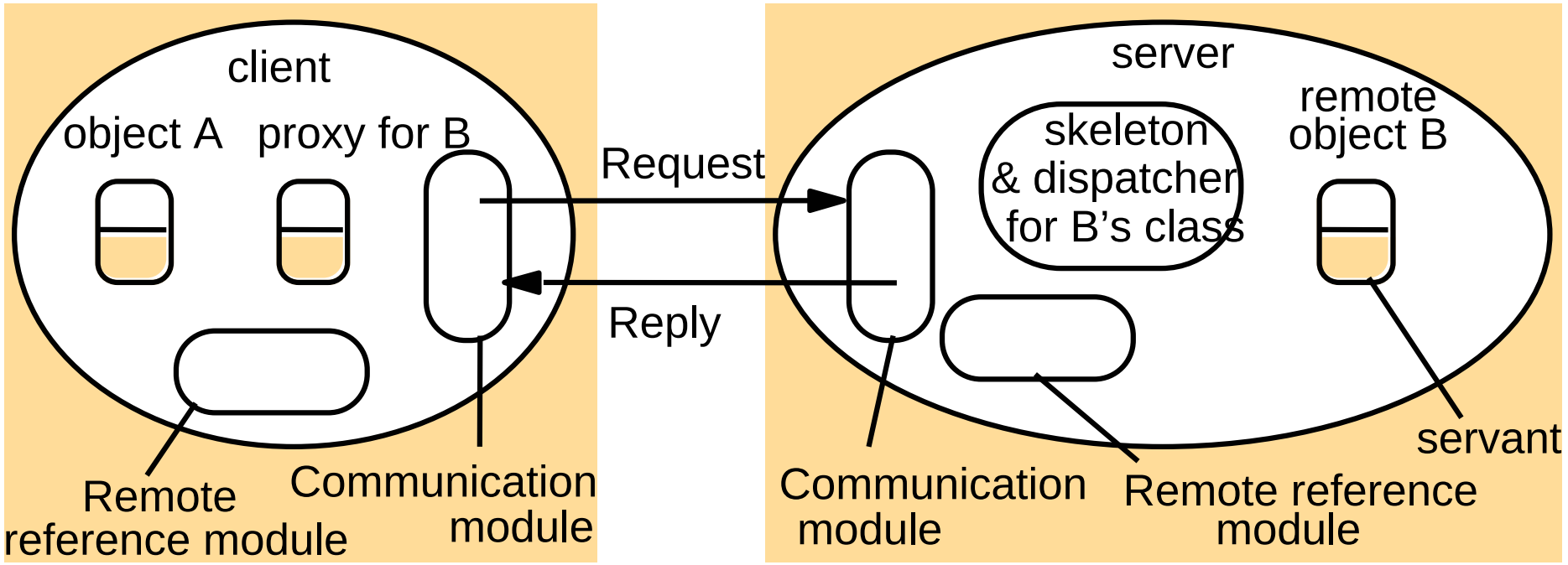


Figure 5.7 Role of proxy and skeleton in RMI

- Communication module
  - Two cooperating communication modules carry out the request-reply protocols: message type, requestID, remote object reference
    - Transmit request and reply messages between client and server
    - Implement specific invocation semantics
  - The communication module in the server
    - selects the dispatcher for the class of the object to be invoked,
    - passes on local reference from remote reference module,
    - returns request

Implementation of RMI (cont.)

- Remote reference module
  - Responsible for translating between local and remote object references and for creating remote object references
  - remote object table: records the correspondence between local and remote object references
    - remote objects held by the process (B on server)
    - local proxy (B on client)
  - When a remote object is to be passed for the first time, the module is asked to create a remote object reference, which it adds to its table
- Servant
  - An instance of a class which provides the body of a remote object
  - handles the remote requests
- RMI software
  - Proxy: behaves like a local object, but represents the remote object
  - Dispatcher: look at the methodID and call the corresponding method in the skeleton
  - Skeleton: implements the method

  Generated automatically by an interface compiler

Implementation Alternatives of RMI

- Dynamic invocation
  - Proxies are static—interface complied into client code
  - Dynamic—interface available during run time
    - Generic invocation; more info in “Interface Repository” (COBRA)
    - Dynamic loading of classes (Java RMI)
- Binder
  - A separate service to locate service/object by name through table mapping for names and remote object references
- Activation of remote objects
  - Motivation: many server objects not necessarily in use all of the time
    - Servers can be started whenever they are needed by clients, similar to inetd
  - Object status: active or passive
    - active: available for invocation in a running process
    - passive: not running, state is stored and methods are pending
  - Activation of objects:
    - creating an active object from the corresponding passive object by creating a new instance of its class
    - initializing its instance variables from the stored state
  - Responsibilities of ***activator***
    - Register passive objects that are available for activation
    - Start named server processes and activate remote objects in them
    - Keep track of the locations of the servers for remote objects that it has already activated

Implementation Alternatives of RMI (cont.)

- Persistent object stores
  - An object that is guaranteed to live between activations of processes is called a ***persistent object***
  - Persistent object store: managing the persistent objects
    - stored in marshaled from on disk for retrieval
    - saved those that were modified
  - Deciding whether an object is persistent or not:
    - persistent root: any descendent objects are persistent (persistent Java, PerDiS)
    - some classes are declared persistent (Arjuna system)
- Object location
  - specifying a location: ip address, port #, …
  - location service for migratable objects
    - Map remote object references to their probable current locations
    - Cache/broadcast scheme (similar to ARP)
      - Cache locations
      - If not in cache, broadcast to find it
    - Improvement: forwarding (similar to mobile IP)

5.2.6 Distributed Garbage Collection

- Aim: ensure that an object
  - continues to exist if a local or remote reference to it is still held anywhere
  - be collected as soon as no object any longer holds a reference to it
- General approach: reference count
- Java's approach
  - the server of an object (B) keeps track of proxies
  - when a proxy is created for a remote object
    - addRef(B) tells the server to add an entry
  - when the local host's garbage collector removes the proxy
    - removeRef(B) tells the server to remove the entry
  - when no entries for object B, the object on server is deallocated

5.3 Remote Procedure Call

- client: "stub" instead of "proxy" (same function, different names)
  - local call, marshal arguments, communicate the request
- server:
  - dispatcher
  - "stub": unmarshal arguments, communicate the results back

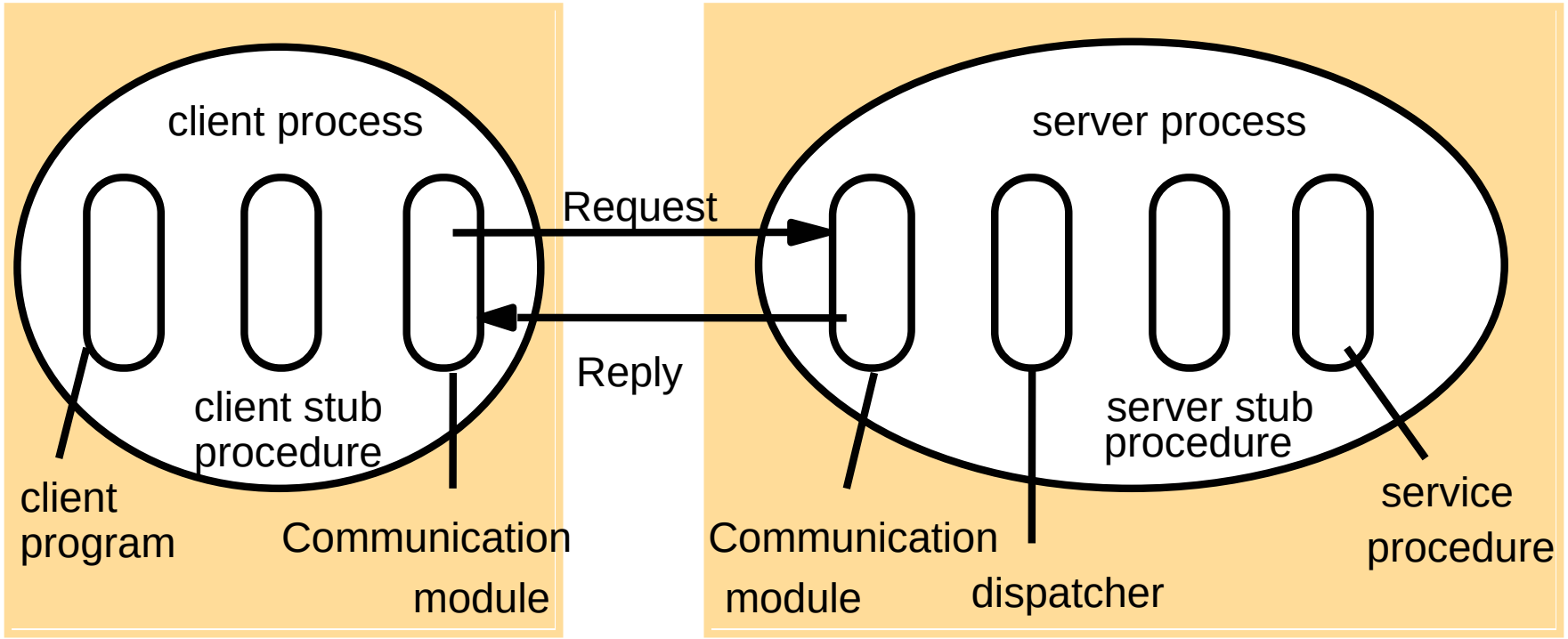


Figure 5.8 Role of client and server stub procedures in RPC in the context of a procedural language

5.3.1 Case Study: Sun RPC

- Sun RPC: client-server in the SUN NFS (network file system)
  - UDP or TCP; in other unix OS as well
  - Also called ONC (Open Network Computing) RPC
- Interface Definition Language (IDL)
  - initially XDR is for data representation, extended to be IDL
  - less modern than CORBA IDL and Java
    - program numbers instead of interface names
    - procedure numbers instead of procedure names
    - single input parameter (structs)
  - rpcgen: compiler for XDR
    - client stub; server main procedure, dispatcher, and server stub
    - XDR marshalling, unmarshaling
- Binding (registry) via a local binder - portmapper
  - Server registers its program/version/port numbers with portmapper
  - Client contacts the portmapper at a fixed port with program/version numbers to get the server port
  - Different instances of the same service can be run on different computers at different ports
- Authentication
  - request and reply have additional fields
  - unix style (uid, gid), shared key for signing, Kerberos

Figure 5.9 Files interface in Sun XDR

```
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;        1
        Data READ(readargs)=2;          2
    }=2;
} = 9999;
```

5.4 Events and Notifications

- Idea behind the use of events
  - One object can react to a change occurring in another object
- Events
  - Notifications of events: objects that represent events
    - asynchronous and determined by receivers what events are interested
  - event types
    - each type has attributes (information in it)
    - subscription filtering: focus on certain values in the attributes (e.g. "buy" events, but only "buy car" events)
- Publish-subscribe paradigm
  - publish events to send
  - subscribe events to receive
- Main characteristics in distributed event-based systems
  - Heterogeneous: a way to standardize communication in heterogeneous systems
    - not designed to communicate directly
  - Asynchronous: notifications are sent asynchronously
    - no need for a publisher to wait for each subscriber--subscribers come and go

Example: Communication of Notifications

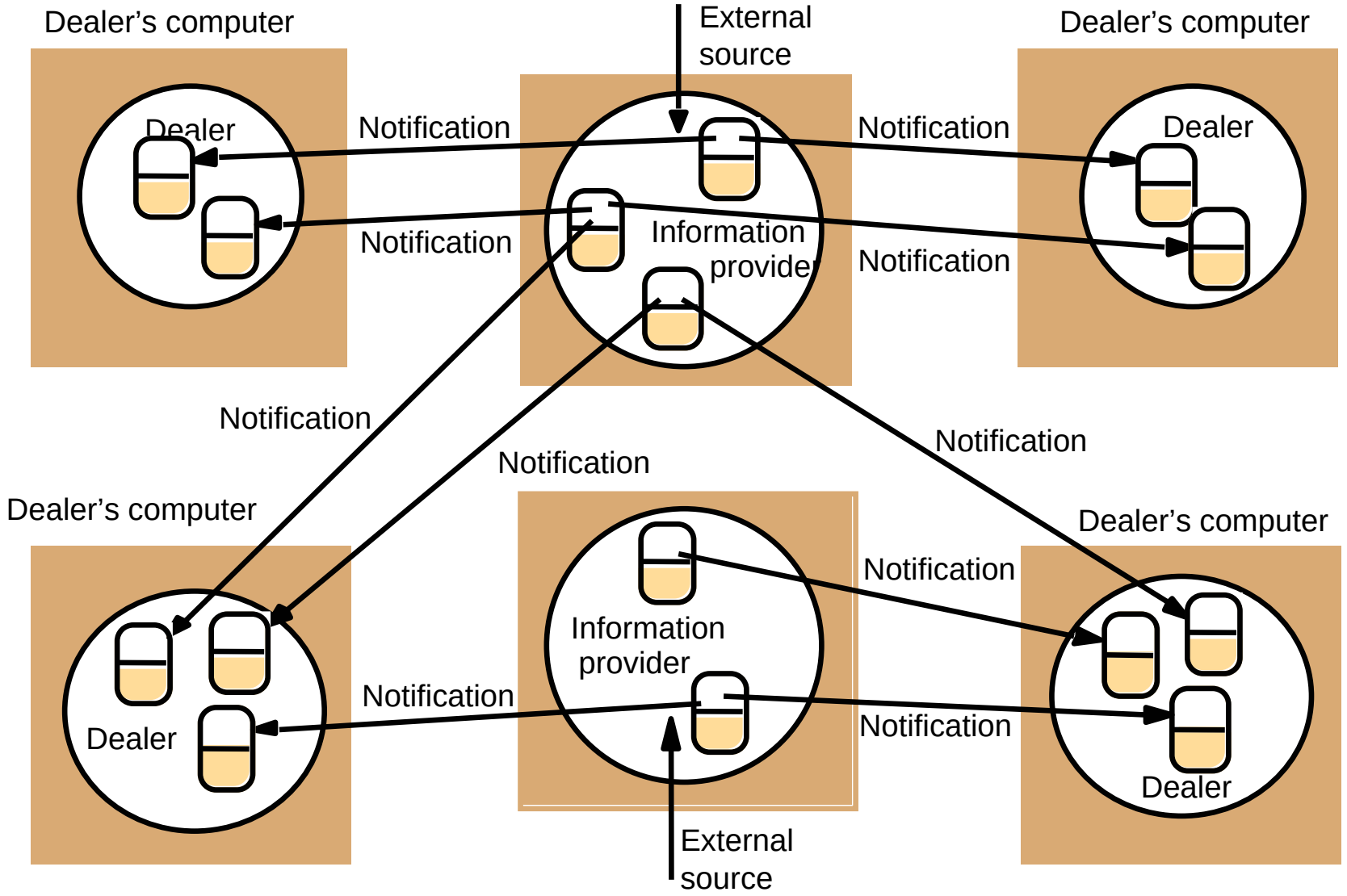


Figure 5.10 Dealing room system: allow dealers using computers to see the latest information about the market prices of the stocks they deal in

Distributed Event Notification

- Distributed event notification
  - decouple publishers from subscribers via an event service (manager)
- Architecture: roles of participating objects
  - object of interest (usually changes in states are interesting)
  - event
  - notification
  - subscriber
  - observer object (proxy) [reduce work on the object of interest]
    - forwarding
    - filtering of events types and content/attributes
    - patterns of events (occurrence of multiple events, not just one)
    - mailboxes (notifications in batches, subscriber might not be ready)
  - publisher (object of interest or observer object)
    - generates event notifications

Example: Distributed Event Notification

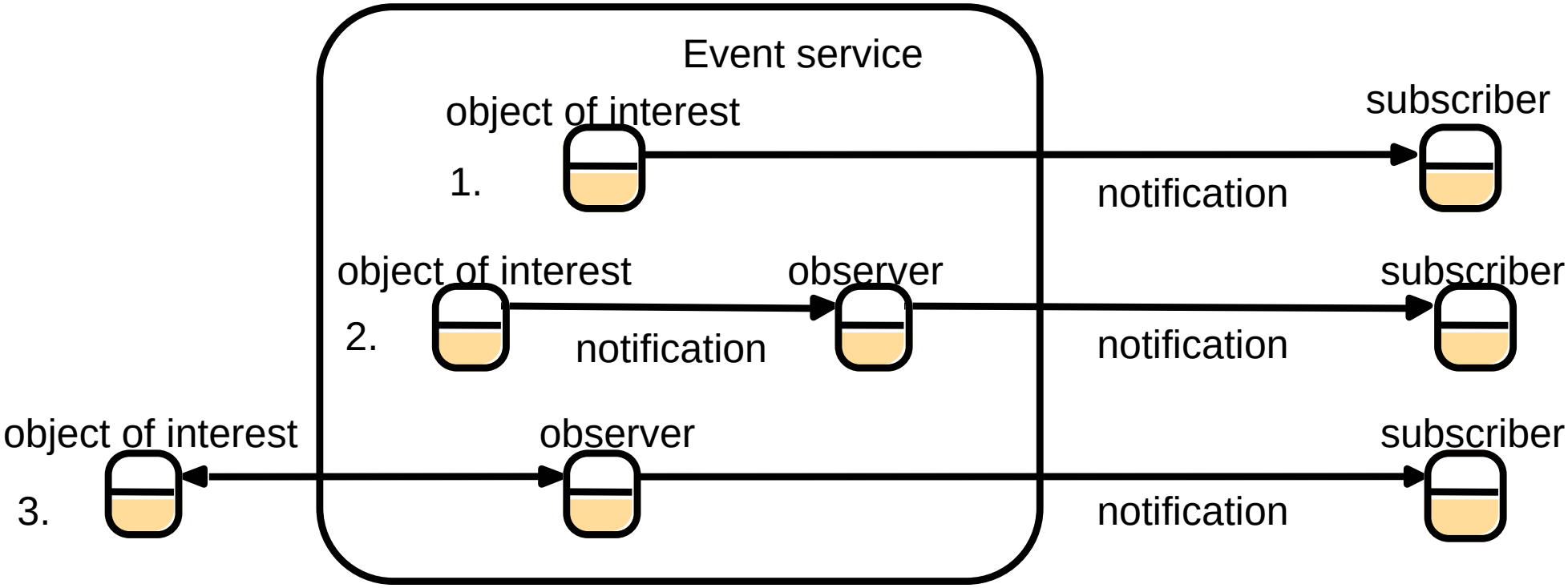


Figure 5.11 Architecture for distributed event notification

- Three cases
  - Inside object without an observer: send notifications directly to the subscribers
  - Inside object with an observer: send notification via the observer to the subscribers
  - Outside object (with an observer)
    1. an observer queries the object of interest in order to discover when events occur
    2. The observer sends notifications to the subscribers

Case Study: Jini Distributed Event Specification

- Jini
  - Allow a potential subscriber in one Java Virtual Machine (JVM) to subscribe to and receive notifications of events in an object of interest in another JVM
  - Main objects
    - event generators (publishers)
    - remote event listeners (subscribers)
    - remote events (events)
    - third-party agents (observers)
  - An object subscribes to events by informing the event generator about the type of event and specifying a remote event listener as the target for notification

5.5 Case Study: Java RMI

```
import java.rmi.*;
import java.util.Vector;
public interface Shape extends Remote {
  int getVersion() throws RemoteException;
  GraphicalObject  getAllState() throws RemoteException;   1
}
public interface ShapeList extends Remote {
  Shape newShape(GraphicalObject g) throws RemoteException;2
  Vector allShapes() throws RemoteException;
  int getVersion() throws RemoteException;
}
```

Figure 5.12 Java Remote interfaces: Shape and ShapreList

Figure 5.13 Java class ShapeListServant implements interface ShapeList

```
import java.rmi.*;
import java.rmi.server.UnicastRemoteObject;
import java.util.Vector;
public class ShapeListServant extends UnicastRemoteObject implements ShapeList {
   private Vector theList; // contains the list of Shapes              1
       private int version;
   public ShapeListServant()throws RemoteException{...}
   public Shape newShape(GraphicalObject g) throws RemoteException {  2
     version++;
         Shape s = new ShapeServant(g, version);                      3
       theList.addElement(s);
       return s;
   }
   public Vector allShapes()throws RemoteException{...}
   public int getVersion() throws RemoteException { ... }
}
```

Figure 5.14 Java class ShapeListServer with main method

Figure 5.16 Java client of ShapreList

```
import java.rmi.*;
public class ShapeListServer{
    public static void main(String args[]){
        System.setSecurityManager(new RMISecurityManager());
         try{
            ShapeList aShapeList = new ShapeListServant();            1
          Naming.rebind("Shape List", aShapeList );                   2
            System.out.println("ShapeList server ready");
        }catch(Exception e) {
           System.out.println("ShapeList server main " + e.getMessage());}
    }
}
```

```
import java.rmi.*;
import java.rmi.server.*;
import java.util.Vector;
public class ShapeListClient{
  public static void main(String args[]){
  System.setSecurityManager(new RMISecurityManager());
  ShapeList aShapeList = null;
  try{
     aShapeList  = (ShapeList) Naming.lookup("//bruno.ShapeList");        1
     Vector sList = aShapeList.allShapes();                               2
  } catch(RemoteException e) {System.out.println(e.getMessage());
  }catch(Exception e) {System.out.println("Client: " + e.getMessage());}
     }
}
```

Figure 5.13 Naming class of Java RMIregistry

void rebind (String name, Remote obj)

This method is used by a server to register the identifier of a remote object by name, as shown in Figure 15.13, line 3.

void bind (String name, Remote obj)

This method can alternatively be used by a server to register a remote object by name, but if the name is already bound to a remote object reference an exception is thrown.

void unbind (String name, Remote obj)

This method removes a binding.

Remote lookup(String name)

This method is used by clients to look up a remote object by name, as shown in Figure 15.15 line 1. A remote object reference is returned.

String [ ] list( )

This method returns an array of Strings containing the names bound in the registry.

Java RMI Callbacks

- Callbacks
  - server notifying the clients of events
  - why?
    - polling from clients increases overhead on server
    - not up-to-date for clients to inform users
  - how
    - remote object (callback object) on client for server to call
    - client tells the server about the callback object, server put the client on a list
    - server call methods on the callback object when events occur
  - client might forget to remove itself from the list
    - lease--client expire