

Unit 2:

Distributed Objects and Remote Communication

References:

1. G. Coulouris, J. Dollimore and T. Kindberg; **Distributed Systems Concepts and Design**, 4th Edition.
2. Andrew S. Tanenbaum and Maarten van Steen; **Distributed Systems: Principles and Paradigms**, 2nd Edition.

2. Communication in Distributed Systems

2.1 Inter-process Communication

2.1.1 Message passing

2.1.2 Remote Procedure Calls (RPC)

2.1.3 Communication between distributed objects

2.1.4 Remote Method Invocation (RMI)

2.1.5 Events and Notifications

2.2 Group Communication

2.2.1 Multicast

2.2.2 Publish/subscribe systems

2.2.3 Consistency models

Communication between distributed objects

What are issues in distributing objects?

- How can we identify objects?
- What is involved in invoking a method implemented by the class?
 - What methods are available?
 - How can we pass parameters and get results?
- Can we track events in a distributed system?

Communication between distributed objects

- Objects that can receive remote method invocations are called **remote objects** and they implement a **remote interface**.
- Programming models for distributed applications are:
 - Remote Procedure Call (RPC)
 - ❖ Client calls a procedure implemented and executing on a remote computer
 - ❖ Call as if it was a local procedure

Communication between distributed objects

➤ Remote Method Invocation (RMI)

- ❖ Local object invokes methods of an object residing on a remote computer
- ❖ Invocation as if it was a local method call

➤ Event-based Distributed Programming

- ❖ Objects receive asynchronous notifications of events happening on remote computers/processes

Communication between distributed objects

■ Middleware

- Software that provides a programming model above the basic building blocks of processes and message passing is called middleware.
- The middleware layer uses protocols based on messages between processes to provide its higher-level abstractions such as remote invocation and events.

(Figure 1)

Communication between distributed objects

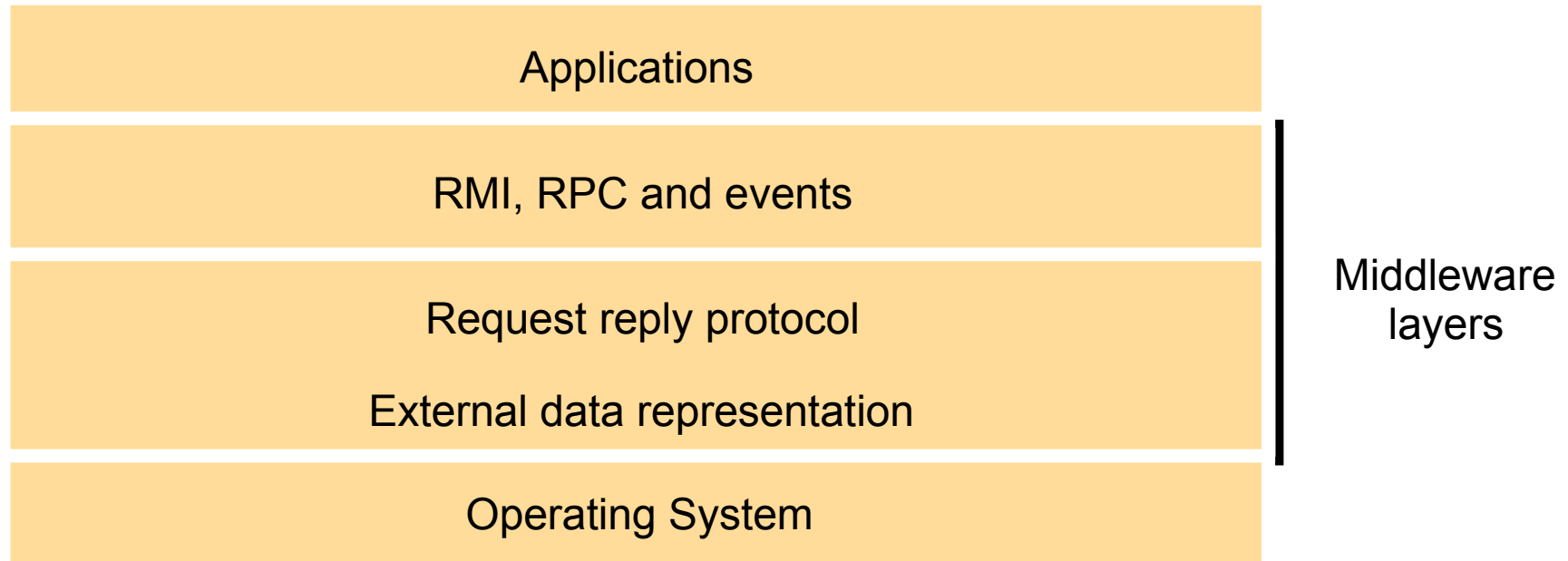


Figure 1. Middleware layers

Role of Middleware

▪ **Transparency Features of Middleware**

➤ **Location** transparency:

- ❖ In RMI and RPCs, the client calls a procedure/method without knowledge of the location of invoked method/procedure.

➤ **Transport protocol** transparency:

- ❖ E.g., request/reply protocol used to implement RPC can use either UDP or TCP.

Role of Middleware

- Transparency of computer **hardware**
 - ❖ They hide differences due to hardware architectures, such as byte ordering.
- Transparency of **operating system**
 - ❖ It provides independency of the underlying operating system.
- Transparency of **programming language** used
 - ❖ E.g., by use of programming language independent Interface Definition Languages (IDL), such as CORBA IDL.

Interface

- **Interfaces for RMI and RPC**
 - An explicit interface is defined for each module.
 - An Interface hides all implementation details.
 - Accesses the variables in a module can only occur through methods specified in interface.

Interface

➤ Interface in distributed system

- ❖ No direct access to remote variables is possible
 - Using message passing mechanism to transmit data objects and variables
 - » Request-reply protocols
 - » Local parameter passing mechanisms (by value, by reference) is not applicable to remote invocations
 - Specify input, output as attribute to parameters
 - » Input: transmitted with request message
 - » Output: transmitted with reply message

Interface

- ❖ Pointers are not valid in remote address spaces
 - Cannot be passed as argument along interface
- RPC and RMI interfaces are often seen as a client/server system
 - ❖ **Service interface** (in client server model)
 - Specification of procedures and methods offered by a server
 - ❖ **Remote interface** (in RMI model)
 - Specification of methods of an object that can be invoked by objects in other processes

Interface Definition Languages (IDL)

- ❖ Impossible to specify direct access to variables in remote classes
- ❖ Hence, access only through specified interface
- ❖ Desirable to have language-independent IDL that compiles into access methods in application programming language
- ❖ Example: CORBA IDL
(Figure 2)

Interface

```
// In file Person.idl  
struct Person {  
  string name;  
  string place;  
  long year;  
};  
interface PersonList {  
  readonly attribute string listname;  
  void addPerson(in Person p) ;  
  void getPerson(in string name, out Person p);  
  long number();  
};
```

Figure 2. CORBA IDL example

Service Interface (RPC)

❖ Service interface

- ❖ Specifies set of procedures available to client
- ❖ Input and output parameters
- ❖ Remote Procedure Call
 - ❖ arguments are marshaled
 - ❖ marshaled packet sent to server
 - ❖ server unmarshals packet, performs procedure, and sends marshaled return packet to client
 - ❖ client unmarshals the return
 - ❖ all the details are transparent

Remote Interface (RMI)

- **Remote interface**

- specifies methods of an object available for remote invocation
- input and output parameters may be objects
- Remote Method Invocation
 - communication actual arguments marshaled and sent to server
 - server unmarshals packet, performs procedure, and sends marshaled return packet to caller
 - client unmarshals return packet
 - common format definition for how to pass objects (e.g., CORBA IDL or Java RMI)

Interface Definition Language

- **Interface Definition Language**
 - notation for language independent interfaces
 - specify type and
 - kind of parameters
 - examples
 - CORBA IDL for RMI
 - Sun XDR for RPC
 - DCOM IDL
 - IDL compiler allows interoperability

CORBA IDL Example

```
struct Person {  
    string name;  
    string place;  
    long year;  
};  
interface PersonList {  
    readonly attribute string listname;  
    void addPerson(in Person p) ;  
    void getPerson(in string name, out Person  
p);  
    long number();  
};
```

Figure 5.2

CORBA has a struct

remote interface

remote interface defines
methods for RMI

parameters are *in*, *out* or *inout*

- Remote interface:
 - specifies the **methods** of an object available for remote invocation
 - an interface definition language (or IDL) is used to specify remote interfaces. E.g. the above in CORBA IDL.
 - Java RMI would have a class for *Person*, but CORBA has a *struct*

The Object Model

- ❖ An object encapsulates both data and methods.
- ❖ Objects can be accessed via **object references**.
- ❖ An **interface** provides a definition of the signatures of a set of methods.
- ❖ Actions are performed by **method invocations**.
 - ❖ The state of receiver may be changed.
 - ❖ Further invocations of methods on other objects may take place.

The Object Model

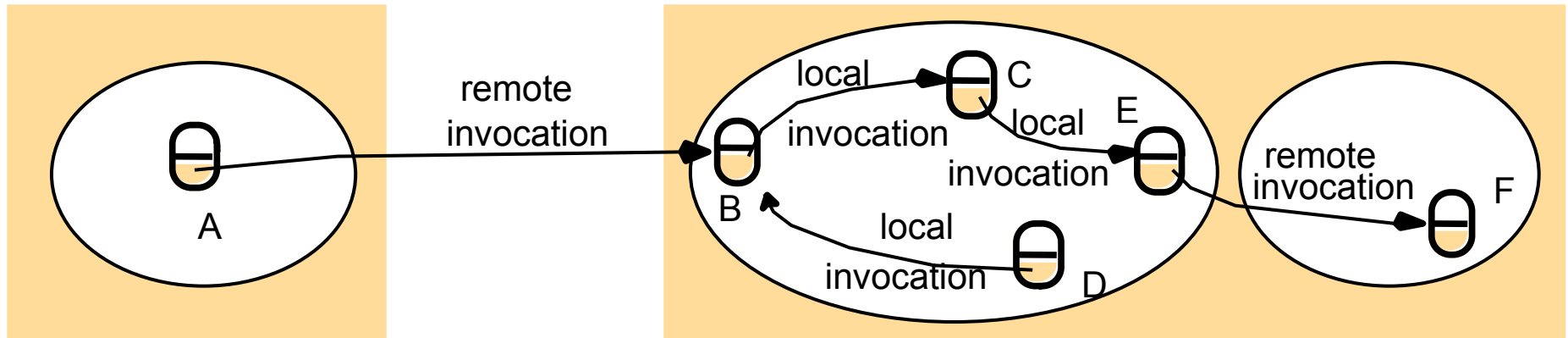
- ❖ **Exceptions** may be thrown to caller when an error occurs.
- ❖ **Garbage collection** frees the space occupied by objects when they are no longer needed.

The Distributed Objects Model

- ❖ **Remote method invocation** – Method invocations between objects in different processes, whether in the same computer or not.
- ❖ **Local method invocation** – Method invocations between objects in the same process.
- ❖ **Remote object** – Objects that can receive remote invocations.
- ❖ Remote and local method invocations are shown in Figure 5.3.

Distributed Object Model

Figure 5.3



- ❖ each process contains objects, some of which can receive remote invocations(B and F), others only local invocations(C,E and D)
- ❖ 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.
- ❖ the **remote interface** specifies which methods can be invoked remotely

Distributed Object Model

❖ Remote object reference

- ❖ An object must have the remote object reference of an object in order to do remote invocation of an object
- ❖ Remote object references may be passed as input arguments or returned as output arguments

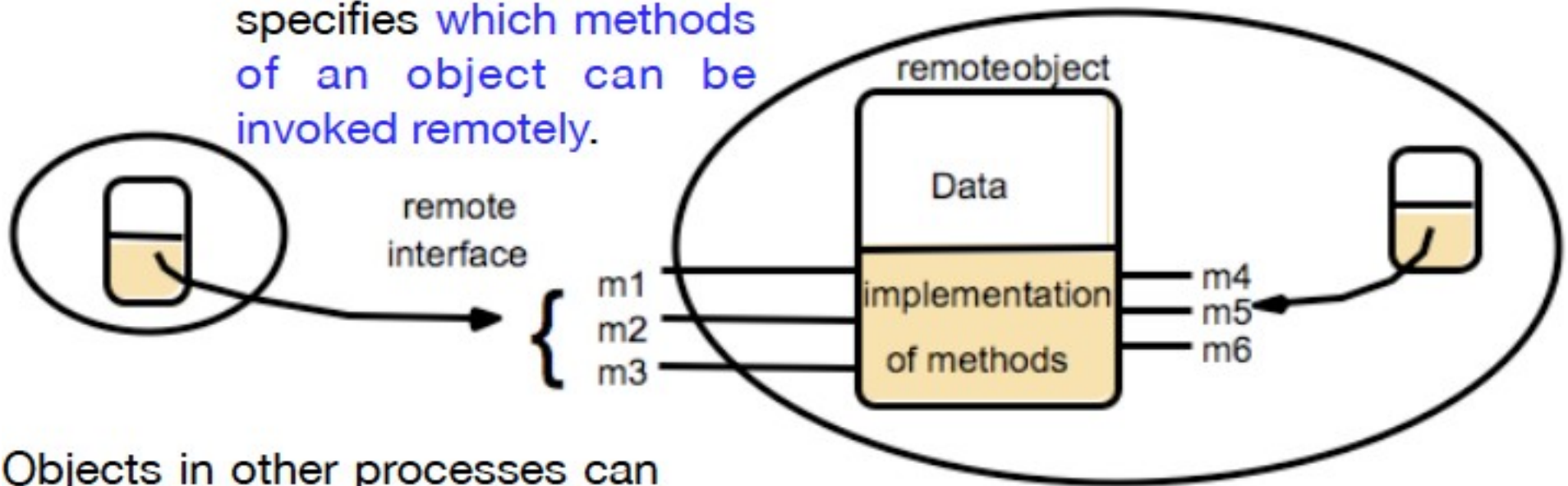
❖ Remote interface

- ❖ Objects in other processes can invoke only the methods that belong to its remote interface (Figure 5.4).
- ❖ CORBA – uses IDL to specify remote interface
- ❖ JAVA – extends interface by the **Remote** keyword.

Remote Interface

The **remote interface** specifies which methods of an object can be invoked remotely.

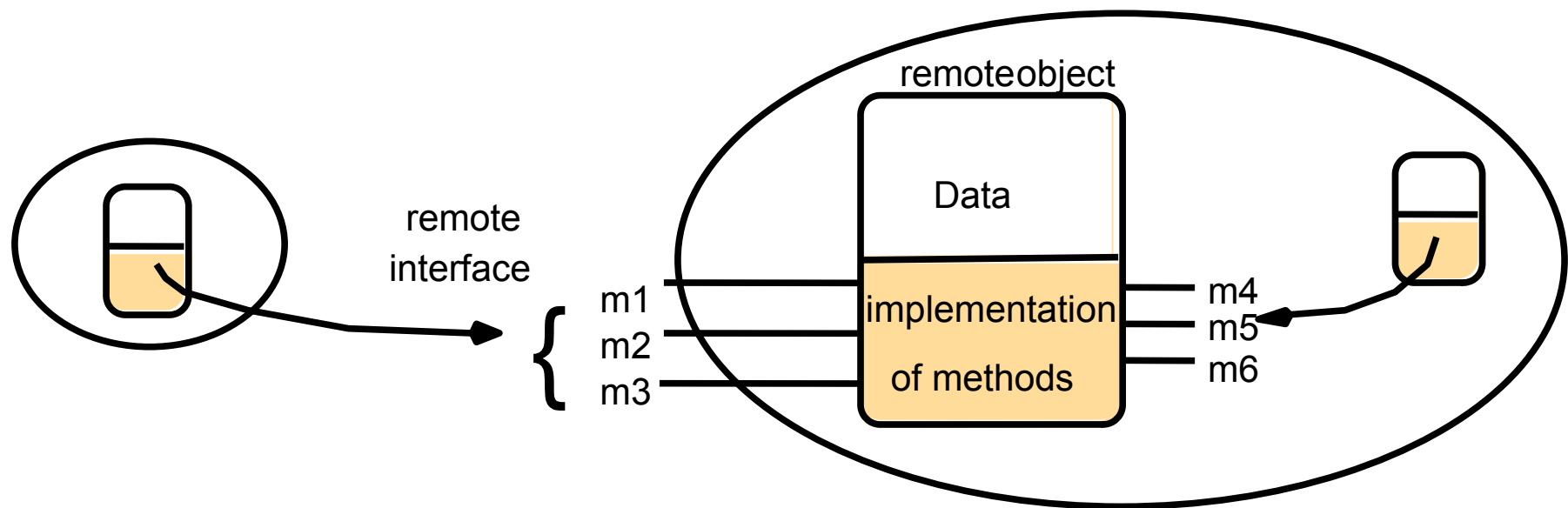
The class of a remote object implements the methods of its remote interface.



Objects in other processes can invoke *only* the methods that belong to the remote interface of a remote object.

Local objects can invoke the methods in the remote interface as well as other methods implemented by a remote object.

Figure 5.4
A remote object and its remote interface



Design Issues for RMI

- ❖ Two important issues in making RMI natural extension of local method: (These problems won't occur in the local invocation.)
 - ❖ **Number of times of invocations** are invoked in response to a single remote invocation
 - ❖ **Level of location transparency**
- ❖ **Exactly once invocation semantics** - Every method is executed exactly once. (Ideal situation)

Invocation Semantics Properties

❖ **Maybe invocation semantics:**

- ❖ The invoker can not determine whether or not the remote method has been executed.
- ❖ Types of failures:
 - ❖ **Omission failures** if the invocation or result message is lost.
 - ❖ **Crash failures** when the server containing the remote object fails.
- ❖ Useful for applications where occasional failed invocation are acceptable.

Invocation Semantics Properties

❖ **At-least-once invocation semantics:**

- ❖ The invoker either receives a result (in which case the user knows the method was executed **at least once**) or an exception.
- ❖ Types of failures:
 - ❖ Retransmitting request masks omission failures.
 - ❖ Crash failures when the server containing the remote object fails.
 - ❖ Arbitrary failure when the remote method is invoked more than once, wrong values are stored or returned. An **idempotent operation** that can be performed repeatedly with the same effect can be a solution.
- ❖ Useful if the objects in a server can be designed to have idempotent operations.

Invocation Semantics Properties

❖ **At-most-once invocation semantics:**

- ❖ The invoker either receives a result (and the user knows the the method was executed exactly **at most once**) or an exception.
- ❖ All fault tolerance methods executed.
- ❖ **Omission failures** can be eliminated by retransmitting request.
- ❖ **Arbitrary failures** can be prevented by ensuring that no method is executed more than once.
- ❖ JAVA RMI and CORBA use **at-most-once semantics**. CORBA also **maybe semantics** for methods that do not return results. SUNRPC provides **at-least-once semantics**.

Invocation Semantics

- ❖ **To provide a more reliable request-reply protocol, these fault-tolerant measures can be employed:**
 - ❖ **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 at the server.

Invocation Semantics

- ❖ Local invocations are executed exactly once
- ❖ Remote invocations cannot achieve this.
 - ❖ The Request-reply protocol can apply fault-tolerance measure.

Figure 5.5

Fault tolerance measures

Invocation semantics

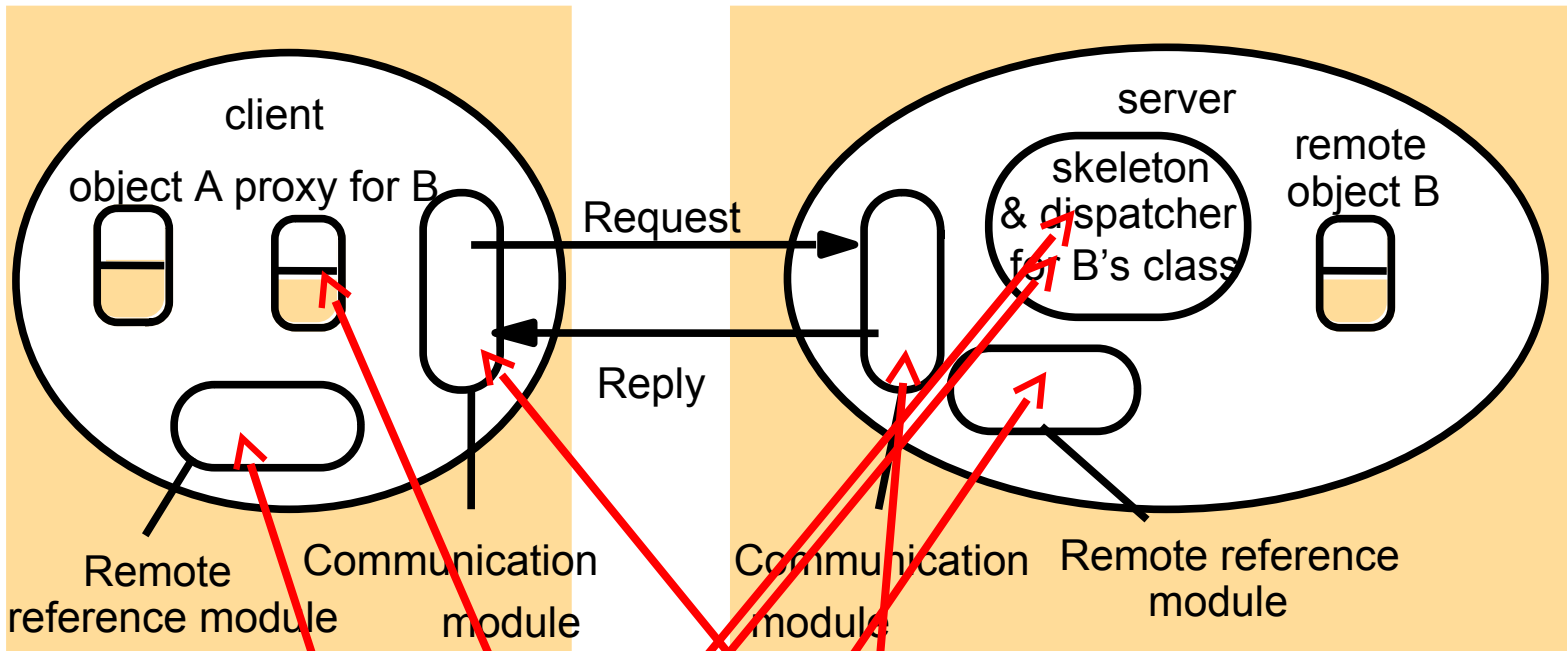
<i>Retransmit request message</i>	<i>Duplicate filtering</i>	<i>Re-execute procedure or retransmit reply</i>	
No	Not applicable	Not applicable	<i>Maybe</i>
Yes	No	Re-execute procedure	<i>At-least-once</i>
Yes	Yes	Retransmit reply	<i>At-most-once</i>

Implementation of RMI

- ❖ Figure 5.6 shows an object A invokes a method in a remote object B.
- ❖ Communication module:
 - ❖ Request-Reply Protocol
 - ❖ Responsible for providing selected invocation semantics
 - ❖ The server selects the dispatcher for the class of the object to be invoked.

The architecture of remote method invocation

Figure 5.6



Proxy - makes RMI transparent to client. Class implements remote interface. Marshals requests and unmarshals results. Forwards request.

Skeleton - implements methods in remote interface. Unmarshals requests and marshals results. Invokes method in remote object.

RMI software - between application level objects and communication and remote reference modules

Remote Reference Module

❖ Responsibilities

- ❖ translation between **local** and **remote** object references
 - ❖ remote object table
 - ❖ - entry for each remote object held by process
 - ❖ - entry for each local proxy
 - ❖ arriving remote object reference - creation of remote object references
- ❖ creation of remote object references
 - ❖ need to pass a remote object - look up in remote object table (create new remote object reference and add entry if necessary)

RMI Software

- ❖ **Proxy** - provides remote invocation transparency
 - ❖ marshal arguments, unmarshal results, send and receive messages
- ❖ **Dispatcher** - handles transfer of requests to correct method
 - ❖ receive requests, select correct method, and pass on request message
- ❖ **Skeleton** - implements methods of remote interface
 - ❖ unmarshal arguments from request, invoke the method of the remote object, and marshal the results

RMI Server and Client Programs

❖ Server

- ❖ classes for dispatchers, skeletons and remote objects
- ❖ initialization section for creating some remote objects
- ❖ registration of remote objects with the binder

❖ Client

- ❖ classes for proxies of all remote objects
- ❖ binder to look up remote object references
- ❖ cannot create remote objects by directly calling constructors - provide factory methods instead

RMI Binder and Server Threads

- ❖ A **binder** in a distributed system is a separate service that maintains a table containing mappings from textual names to remote object references.
- ❖ **Server threads**
 - ❖ sometimes implemented so that remote invocation causes a new thread to be created to handle the call
 - ❖ server with several remote objects might also allocate separate threads to handle each object

RMI Binder and Server Threads

❖ Activation of remote objects

- ❖ A remote object is described as **active** when it is a running process.
- ❖ A remote object is described as **passive** when it can be made active if requested.
- ❖ An object that can live between activations of processes is called a **persistent object**.
- ❖ A **location service** helps clients to locate remote objects from their remote references.

RMI Distributed Garbage Collection

- ❖ Aim - recover memory if no reference to an object exist. If there is a reference object should still exists.
- ❖ Java distributed algorithm - based on reference counting
- ❖ The distributed garbage collector works in cooperation with the local garbage collector.
 - ❖ Each server has a table (**B.holders**) that maintains list of references to an object.
 - ❖ When the client C first receives a reference to an object B, it invokes **addRef(B)** and then creates a proxy. The server adds C to the remote object holder **B.holders**.

RMI Distributed Garbage Collection

- ❖ The distributed garbage collection (continued):
 - ❖ When remote object B is no longer reachable, it deletes the proxy and invokes **removeRef(B)**.
 - ❖ When **B.holders** is empty, the server reclaim the space occupied by B.
- ❖ Leases in Jini
 - ❖ To avoid complicated protocols to discover whether a resource are still used, the resource is leased for use for a period of time.
 - ❖ An object representing a lease implements the **Lease** interface.

Remote Procedure Call (RPC) (Finished)

- A remote procedure call (RPC) is similar to a remote method invocation (RMI).
- A client program calls a procedure in another program running in a server process.
- RPC, like RMI, may be implemented to have one of the choices of invocation semantics - **at-least-once**, **at-most-once** are generally chosen.
- RPC is generally implemented over a request-reply protocol.
- The software that support RPC is shown in **Figure 3**.

Remote Procedure Call (RPC)

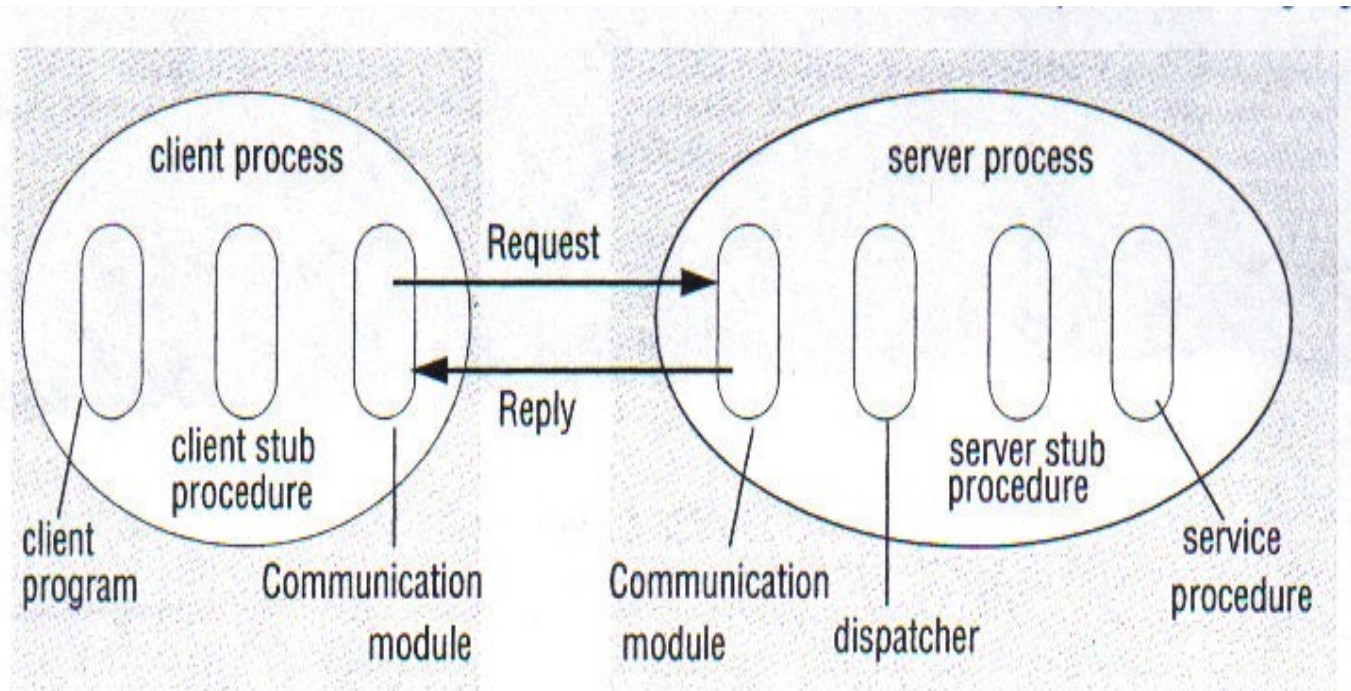


Figure 3. Role of client and server stub procedures in RPC in the context of a procedural language

Remote Procedure Call (RPC)

- RPC only addresses procedure calls.
- RPC is not concerned with objects and object references.
- A client that accesses a server includes one **stub procedure** for each procedure in the service interface.
- A client stub procedure is similar to a proxy method of RMI (discussed later).
- A server stub procedure is similar to a skeleton method of RMI (discussed later).

RPC Example 1: Local Program

- * A first program (hello.c) to test rpc. Use of this program is for
- * testing the logic of the rpc programs.

```
#include <stdio.h>
```

```
int
```

```
main (void) {
```

```
    static char * result;
```

```
    static char msg[256];
```

```
    printf("getting ready to return value\n");
```

```
    strcpy(msg, "Hello world");
```

```
    result= msg;
```

```
    printf("Returning %s\n", result);
```

```
    return (0);
```

```
} /
```

Protocol Definition Program

- The name of this program is hello.x
- The number at the end is version number and should be updated each time the service is updated to make sure the active old copies is not responding to the client program.

```
program HELLO {  
    version ONE{  
        string PRINT_HELLO() = 1;  
    } = 1 ;  
} = 0x2000059;
```

Client Program

- Now we are ready to use rpcgen (command for generating the required programs).
- Note that so far we have only hello.c and hello.x
- After running “rpcgen -a -C hello.x” the directory contain following files:

```
-rw-rw-r-- 1 user user 131 Oct 5 12:15 hello.c
-rw-rw-r-- 1 user user 688 Oct 5 12:19 hello.h
-rw-rw-r-- 1 user user 90 Oct 5 12:18 hello.x
-rw-rw-r-- 1 user user 776 Oct 5 12:19 hello_client.c
-rw-rw-r-- 1 user user 548 Oct 5 12:19 hello_clnt.c
-rw-rw-r-- 1 user user 316 Oct 5 12:19 hello_server.c
-rw-rw-r-- 1 user user 2076 Oct 5 12:19 hello_svc.c
```

- The two templates that we should modify for this example are **hello_client.c** and **hello_server.c**.

Template of hello_client Program

- * This is sample code generated by rpcgen.
- * These are only templates and you can use them as a guideline for developing your own functions.

```
#include "hello.h"  
void hello_1(char *host)  
{  
    CLIENT *clnt;  
    char * *result_1;  
    char *print_hello_1_arg;
```

Template of hello_client Program

```
#ifndef DEBUG
    clnt = clnt_create (host, HELLO, ONE, "udp");
    if (clnt == NULL) {
        clnt_pcreateerror (host);
        exit (1);
    }
#endif /* DEBUG */

    result_1 = print_hello_1((void*)&print_hello_1_arg, clnt);
    if (result_1 == (char **) NULL) {
        clnt_perror (clnt, "call failed");
    }
#endif /* DEBUG */

    clnt_destroy (clnt);
#endif /* DEBUG */
}
```

Template of hello_client Program

```
Int main (int argc, char *argv[])
{
    char *host;

    if (argc < 2) {
        printf ("usage: %s server_host\n", argv[0]);
        exit (1);
    }
    host = argv[1];
    hello_1 (host);
    exit (0);
}
```

hello_client Program

- We have to modified hello_client template program.
- The modifications for our first example are very simple.
- Next slides show the modified program of hello_client that needs only few lines.

hello_client Program

- * This is sample code generated by rpcgen.
- * These are only templates and you can use them as a guideline for developing your own functions.

```
#include "hello.h"  
#include <stdlib.h>  
#include <stdio.h>  
  
void  
hello_1(char *host)  
{  
    CLIENT *clnt;  
    char * *result_1;  
    char *print_hello_1_arg;
```

hello_client Program

```
#ifndef DEBUG
clnt = clnt_create (host, HELLO, ONE, "udp");
    if (clnt == NULL) {
        clnt_pcreateerror (host);
        exit (1);
    }
#endif /* DEBUG */
    result_1 = print_hello_1((void*)&print_hello_1_arg, clnt);
    if (result_1 == (char **) NULL)
        clnt_perror (clnt, "call failed");
    else printf("from server: %s\n", *result_1);

#ifndef DEBUG
    clnt_destroy (clnt);
#endif /* DEBUG */
}
```

hello_client Program

```
int
main (int argc, char *argv[])
{
    char *host;

    if (argc < 2) {
        printf ("usage: %s server_host\n", argv[0]);
        exit (1);
    }
    host = argv[1];
    hello_1 (host);
    exit (0);
} //end clinet_server.c
```

Template of hello-server Program

- * This is sample code generated by rpcgen.
- * These are only templates and you can use them as a guideline for developing your own functions.

```
#include "hello.h"
```

```
char **  
print_hello_1_svc(void *argp, struct svc_req *rqstp)  
{  
    static char * result;  
  
    /* insert server code here */  
  
    return &result;  
}
```

hello-server Program

- * This is sample code generated by rpcgen.
- * These are only templates and you can use them as a guideline for developing your own functions.

```
#include "hello.h"  
char **  
print_hello_1_svc(void *argp, struct svc_req *rqstp)  
{  
    static char * result;  
    static char msg[256];  
    printf("getting ready to return value\n");  
    strcpy(msg, "Hello world");  
    result= msg;  
    printf("Returning\n");  
    return &result;  
}
```

Making Client and Server Program

- To compile the client
`leda% gcc hello_client.c hello_clnt.c -o client -lnsl`
- To compile the server
`leda% gcc hello_server.c hello_svc.c -o server -lnsl`
- To run the server use
`leda% ./server`
- To run the client use
`elara% ./client leda`

RPC

- rpcgen facilitates the generation of client and server stubs from the IDL program.
- It even generates client and server template programs.
- The option **-a** is passed to rpcgen and also all the generation of all support files including the make files.
- The **-a** option causes the rpcgen to halt with warnings if template files with default names exist.
- Consider turning a factorial program into a client-server program using RPC.

Events and Notifications

- ❖ The idea behind the use of events is that one object can react to a change occurring in another object.
- ❖ The actions done by the user are seen as **events** that cause state changes in objects.
- ❖ The objects are **notified** whenever the state changes.
- ❖ Local event model can be extended to distributed event-based systems by using the **publish-subscribe** paradigm.

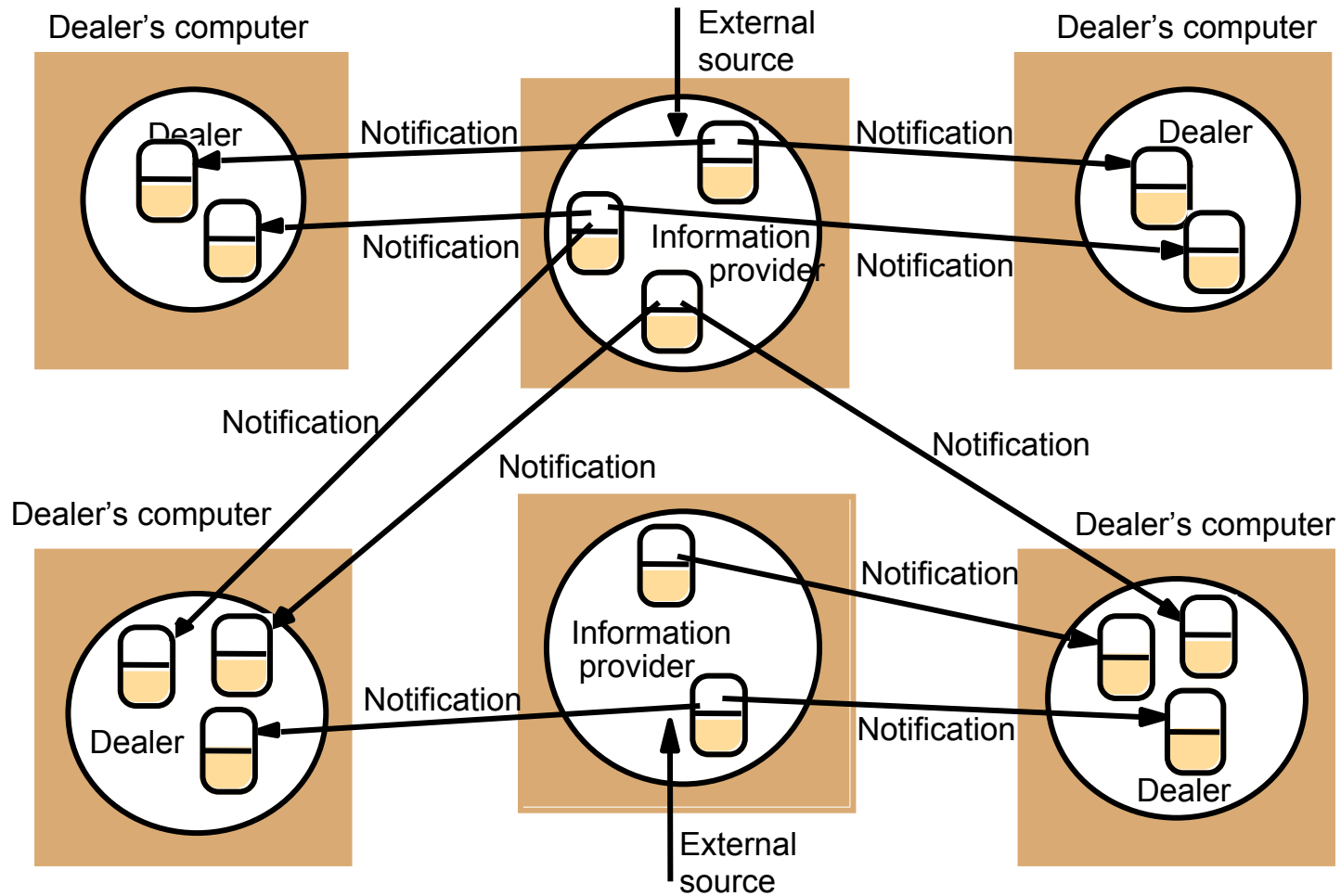
Events and Notifications

- ❖ In **publish-subscribe** paradigm
 - ❖ An object that has event publishes.
 - ❖ Those that have interest subscribe.
- ❖ Objects that represent events are called **notifications**.
- ❖ Distributed event-based systems have two main characteristics:
 - ❖ **Heterogeneous** – Event-based systems can be used to connect heterogeneous components in the Internet.
 - ❖ **Asynchronous** – Notification are sent asynchronously by event-generating objects to those subscribers.

Events and Notifications

- ❖ A dealing room system could be modeled by processes with two different tasks (Figure 5.9):
 - ❖ An **information provider process** continuously receives new trading information from a single external source and applies to the appropriate stock objects.
 - ❖ A **dealer process** creates an object to represent each named stock that the user asks to have displayed.
- ❖ An event source can generate events of one more different **types**. Each event has **attributes** that specify information about that event.

Figure 5.9
Dealing room system



Events and Notifications

- ❖ The architecture of distributed event notification specifies the roles of participants as in Fig. 5.10:
 - ❖ It is designed in a way that publishers work independently from subscribers.
 - ❖ Event service maintains a database of published events and of subscribers' interests.
- ❖ The **roles of the participants** are:
 - ❖ **Object of Interest** – This is an object experiences changes of state, as a result of its operations being invoked.

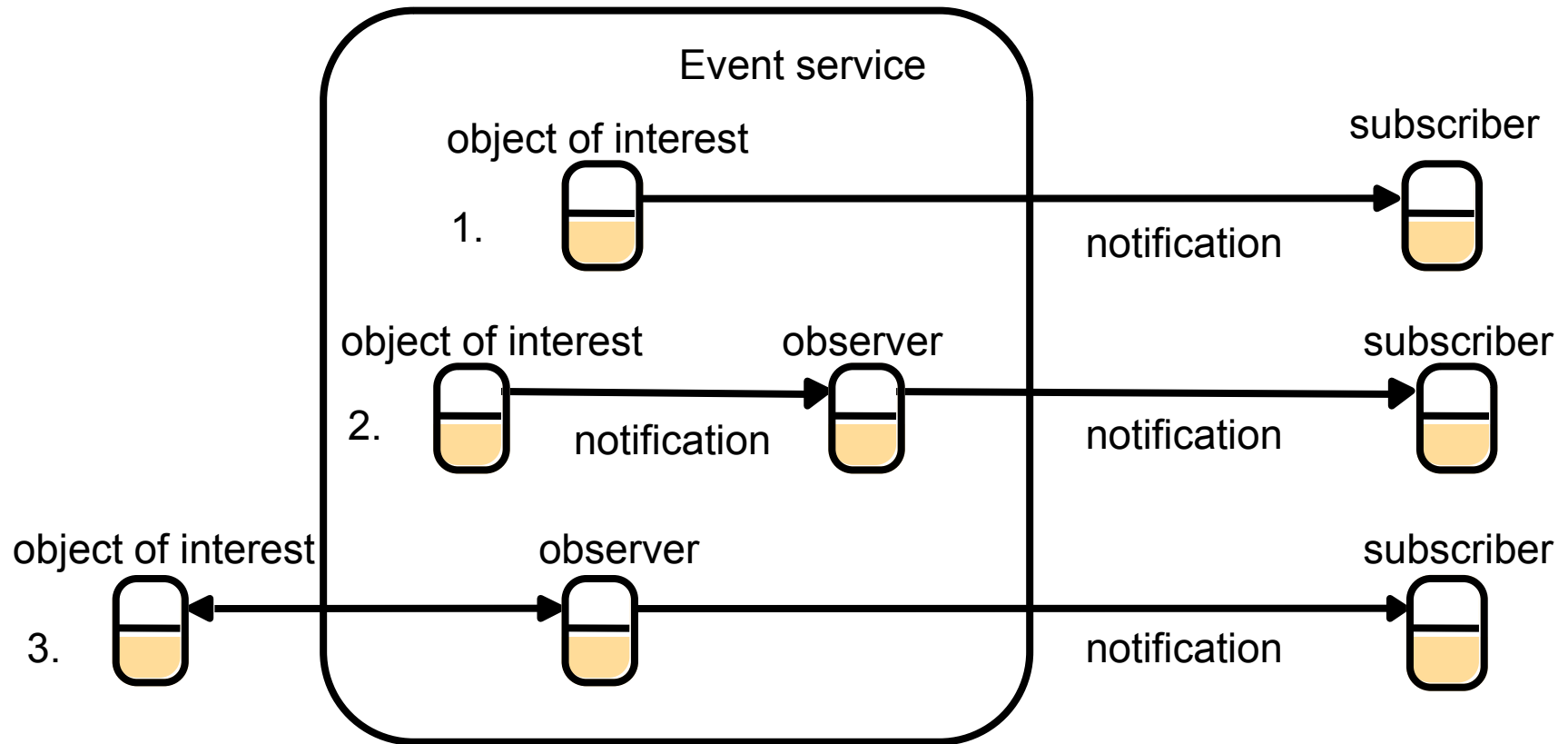
Events and Notifications

- ❖ The **roles of the participants** are (continued):
 - ❖ **Event** – An event occurs at an object of interest as the result of the completion of a method invocation.
 - ❖ **Notification** – A notification is an object that contains information about an event.
 - ❖ **Subscriber** – A subscriber is an object that has subscribed to some type of events in another object.
 - ❖ **Observer objects** – The main purpose of an observer is to separate an object of interest from its subscribers.
 - ❖ **Publisher** – This is an object that declares that it will generate notifications of particular types of event.

Events and Notifications

- Figure 5.10 shows three cases:
 1. An object of interest inside the event service sends notification directly to the subscribers.
 2. An object of interest inside the event service sends notification via the observer to the subscribers.
 3. The observer queries the object of interest outside the event service and sends notifications to the subscribers.

Figure 5.10
Architecture for distributed event notification



Events and Notifications

- ❖ A variety of **delivery semantics** can be employed:
 - ❖ **IP multicast protocol** – information delivery on the latest state of a player in an Internet game
 - ❖ **Reliable multicast protocol** – information provider / dealer
 - ❖ **Totally ordered multicast** - Computer Supported Cooperative Working (CSCW) environment
 - ❖ **Real-time** – nuclear power station / hospital patient monitor

Java RMI

- ❖ Java RMI extends the Java object model to provide support for distributed objects in the Java language.
 - ❖ It allows objects to invoke methods on remote objects using the same syntax as for local invocations.
 - ❖ Type checking applies equally to remote invocations as to local ones.
 - ❖ The remote invocation is known because **RemoteExceptions** has been handled and the remote object is implemented using the **Remote** interface.
 - ❖ The semantics of parameter passing differ because invoker and target are remote from one another.

Java RMI

- ❖ Programming distributed applications in Java RMI is simple.
 - ❖ It is a single-language system.
 - ❖ The programmer of a remote object must consider its behavior in a concurrent environment.
- ❖ The files needed for creating a Java RMI application are:
 - ❖ A **remote interface** defines the remote interface provided by the service. Usually, it is a single line statement specifies the service function (**HelloInterface.java**). (An interface is the skeleton for a public class.)

Java RMI

- ❖ The files needed for creating a Java RMI application are (continued):
 - ❖ A **remote object** implements the remote service. It contains a constructor and required functions. (**Hello.java**)
 - ❖ A **client** that invokes the remote method. (**HelloClient.java**)
 - ❖ The **server** offers the remote service, installs a security manager and contacts rmiregistry with an instance of the service under the name of the remote object. (**HelloServer.java**)

HelloInterface.java

```
import java.rmi.*;
```

```
public interface HelloInterface extends Remote {  
    public String say(String msg) throws  
        RemoteException;  
}
```

Hello.java

```
import java.rmi.*;
import java.rmi.server.*;
public class Hello extends
    UnicastRemoteObject implements HelloInterface {
    private String message;

    public Hello(String msg) throws RemoteException {
        message = msg;
    }
}
```

Hello.java (continued)

```
public String say(String m) throws RemoteException {  
    // return input message - reversing input and suffixing  
    // our standard message  
    return new StringBuffer(m).reverse().toString() + "\n" +  
        message;  
}  
}
```

HelloClient.java

```
import java.rmi.*;
```

```
public class HelloClient {  
    public static void main(String args[]) {  
        String path = "//localhost/Hello";  
        try {  
            if (args.length < 1) {  
                System.out.println("usage: java HelloClient  
<host:port> <string> ... \n");  
            } else path = "/" + args[0] + "/Hello";  
        }  
    }  
}
```

HelloClient.java

```
HelloInterface hello =  
    (HelloInterface) Naming.lookup(path);  
for (int i = 0; i < args.length; ++i)  
    System.out.println(hello.say(args[i]));  
} catch (Exception e) {  
    System.out.println("HelloClient exception: " + e);  
}  
}  
}
```

HelloServer.java

```
import java.rmi.*;
import java.rmi.server.*;

public class HelloServer {
    public static void main(String args[]) {
        // Create and install a security manager
        if (System.getSecurityManager() == null)
            System.setSecurityManager(new RMISecurityManager());
        try {
            Naming.rebind("Hello", new Hello("Hello, world!"));
            System.out.println("server is running...");
        }
    }
}
```

HelloServer.java

```
    catch (Exception e) {  
        System.out.println("Hello server failed:" + e.getMessage());  
    }  
}  
}
```

Java RMI

- **Compile the code**

```
javac Hello.java HelloClient.java HelloInterface.java  
HelloServer.java
```

- **Generate stubs for the remote service**

(make sure that your classpath contains your current directory)

```
rmic Hello
```

- **Start the registry** (in a separate window or in the background)

```
rmiregistry
```

(be sure to kill this process when you're done)

Java RMI

- **Start the server** in one window or in the background with the security policy

java -Djava.security.policy=policy HelloServer

or without the security policy

java HelloServer

- **Run the client** in another window

java HelloClient testing

RMI Summary

- Each object has a (global) remote object reference and a remote interface that specifies which of its operations can be invoked remotely.
- Local method invocations provide exactly-once semantics; the best RMI can guarantee is at-most-once.
- Middleware components (proxies, skeletons and dispatchers) hide details of marshalling, message passing and object location from programmers.

2. Communication in Distributed Systems

2.1 Inter-process Communication

2.1.1 Message passing

2.1.2 Remote Procedure Calls (RPC)

2.1.3 Communication between distributed objects

2.1.4 Remote Method Invocation (RMI)

2.1.5 Events and Notifications

2.2 Group Communication

2.2.1 Multicast

2.2.2 Publish/subscribe systems

2.2.3 Consistency models