Distributed Systems

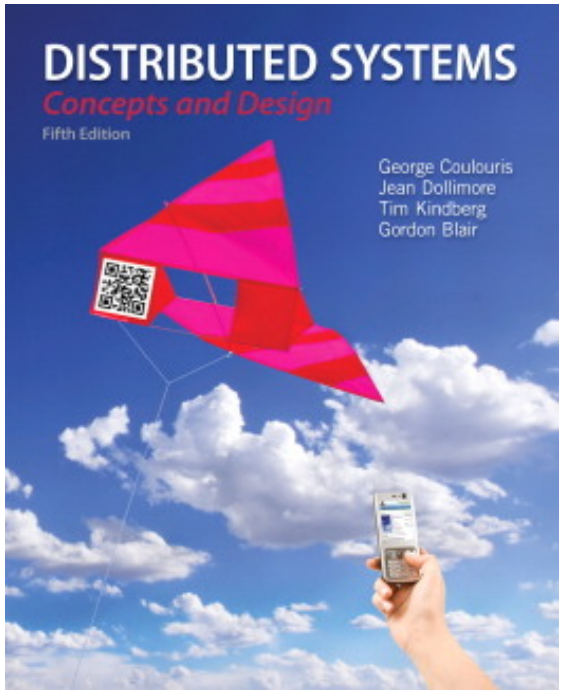
COMP90015 2018 SM2

Remote Invocation

Lectures by Aaron Harwood

Remote Invocation

From Coulouris, Dollimore and Kindberg, *Distributed Systems: Concepts and Design*, Edition 5, © Addison-Wesley 2012.



Lectures prepared by: Shanika Karunasekera and Aaron Harwood.

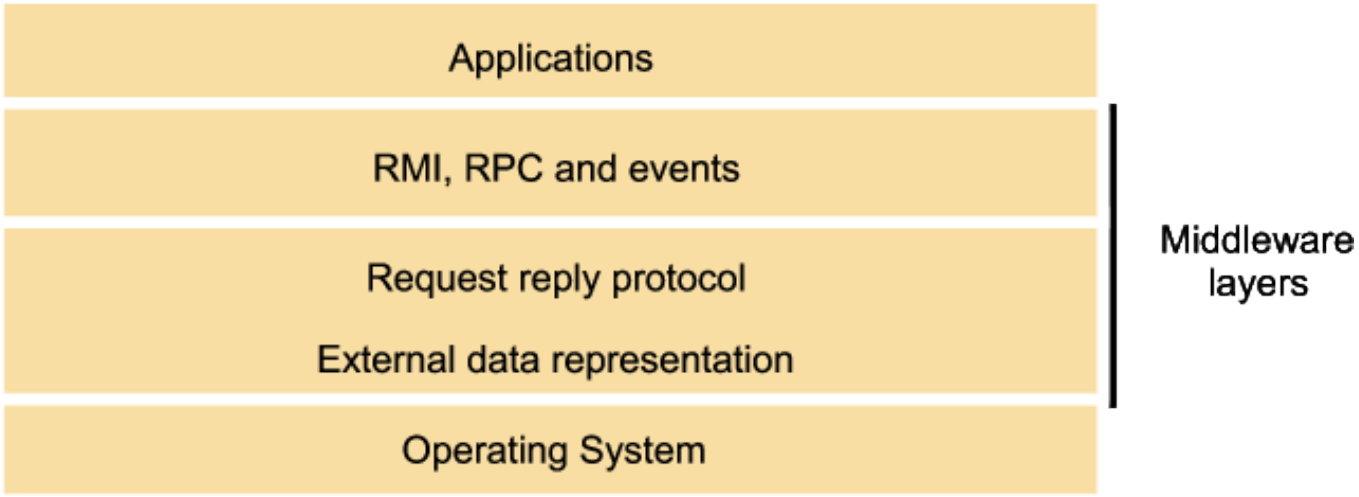
Overview

- Exchange protocols
- Remote Procedure call
- Remote Method Invocation

Introduction

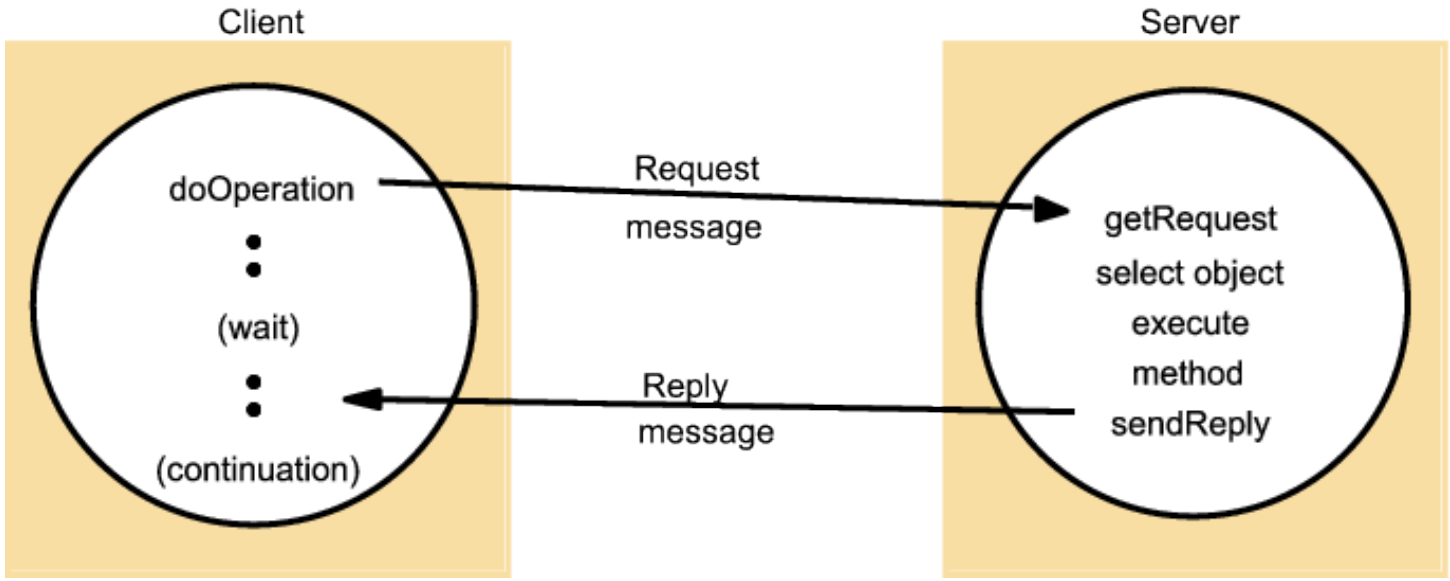
This section covers the high level programming models for distributed systems. Three widely used models are:

- *Remote Procedure Call model* - an extension of the conventional procedure call model.
- *Remote Method Invocation model* - an extension of the object-oriented programming model.



The Request-Reply protocol

The request-reply protocol is perhaps the most common exchange protocol for implementation of remote invocation in a distributed system. We discuss the protocol based on three abstract operations: `doOperation`, `getRequest` and `sendReply`.



Request-Reply Operations

Operations used in the request-reply protocol:

public byte[] doOperation (RemoteObjectRef o, int methodId, byte[] arguments)

sends a request message to the remote object and returns the reply.
The arguments specify the remote object, the method to be invoked and the arguments of that method.

public byte[] getRequest ();

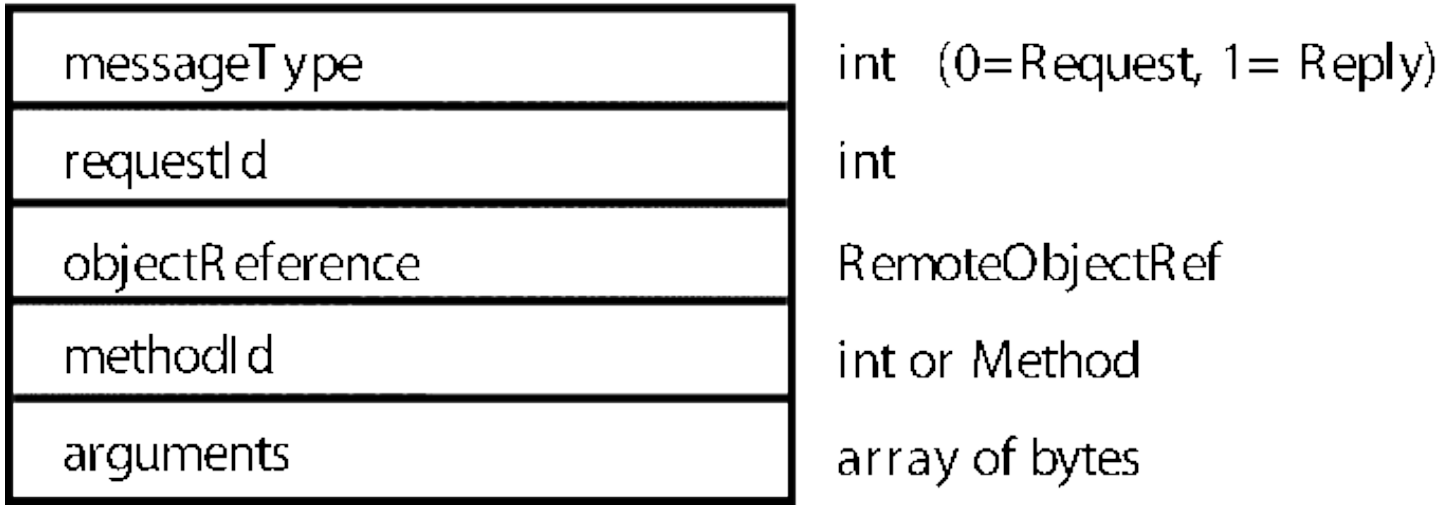
acquires a client request via the server port.

public void sendReply (byte[] reply, InetAddress clientHost, int clientPort);

sends the reply message reply to the client at its Internet address and port.

Typical Message Content

A message in a request-reply protocol typically contains a number of fields as shown below.



Design Issues

- Failure model can consider:
  - Handling timeouts
  - Discarding duplicate messages
  - Handling lost reply messages - strategy depends on whether the server operations are *idemponent* (an operation that can be performed repeatedly)
  - History - if servers have to send replies without re-execution, a history has to be maintained

Three main design decisions related to implementations of the request/reply protocols are:

- Strategy to retry request message
- Mechanism to filter duplicates
- Strategy for results retransmission

Exchange protocols

- Three different types of protocols are typically used that address the design issues to a varying degree:
  - the request (R) protocol
  - the request-reply (RR) protocol
  - the request-reply-acknowledge reply (RRA) protocol
- Messages passed in these protocols:

Name	Messages sent by		
	Client	Server	Client
R	Request		
RR	Request	Reply	
RRA	Request	Reply	Acknowledge reply

Invocation Semantics

Middleware that implements remote invocation generally provides a certain level of semantics:

- **Maybe invocation semantics**: The remote procedure call may be excecuted once or not at all. Unless the caller receives a result, it is unknown as to whether the remote procedure was called.
- **At-least-once invocation semantics**: Either the remote procedure was executed at least once, and the caller received a response, or the caller received an exception to indicate the remote procedure was not executed at all.
- **At-most-once**: The remote procedure call was either executed exactly once, in which case the caller received a response, or it was not executed at all and the caller receives an exception.

The level of transparency provided for remote invocation depends on the design choices and objectives. Java Remote Method Invocation supports at-most-once invocation semantics. Sun Remote Procedure Call supports at-least-once semantics.

Fault Tolerance Measures

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

Transparency

Although location and access transparency are goals for remote invocation, in some cases complete transparency is not desirable due to:

- remote invocations being more prone to failure due to network and remote machines
- latency of remote invocations is significantly higher than that of local invocations

Therefore, many implementations provide access transparency but not complete location transparency. This enables the programmer to make optimisation decisions based on location.

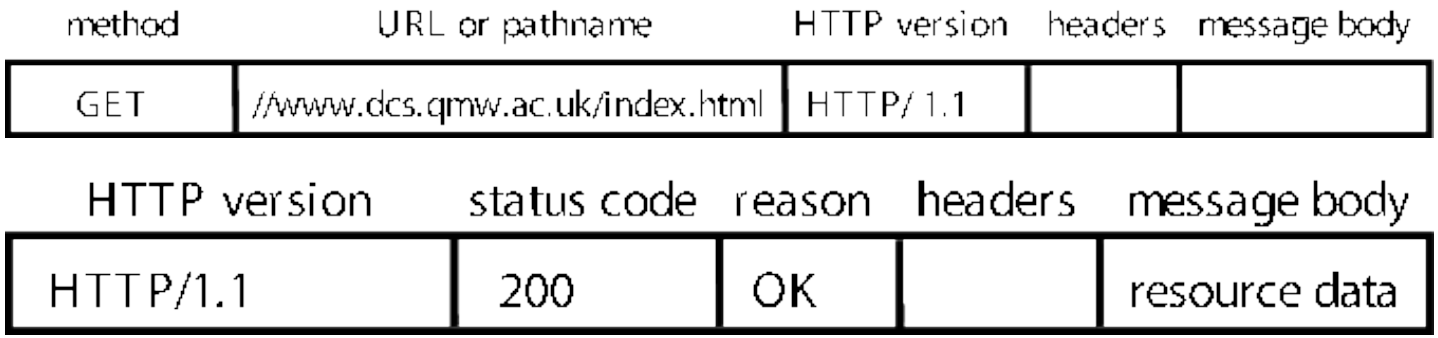
Client-server communication

- Client-server communication normally uses the synchronous request-reply communication paradigm
- Involves *send* and *receive* operations
- TCP or UDP can be used - TCP involves additional overheads:
  - redundant acknowledgements
  - needs two additional messages for establishing connection
  - flow control is not needed since the number of arguments and results are limited

HTTP: an example of a RR protocol

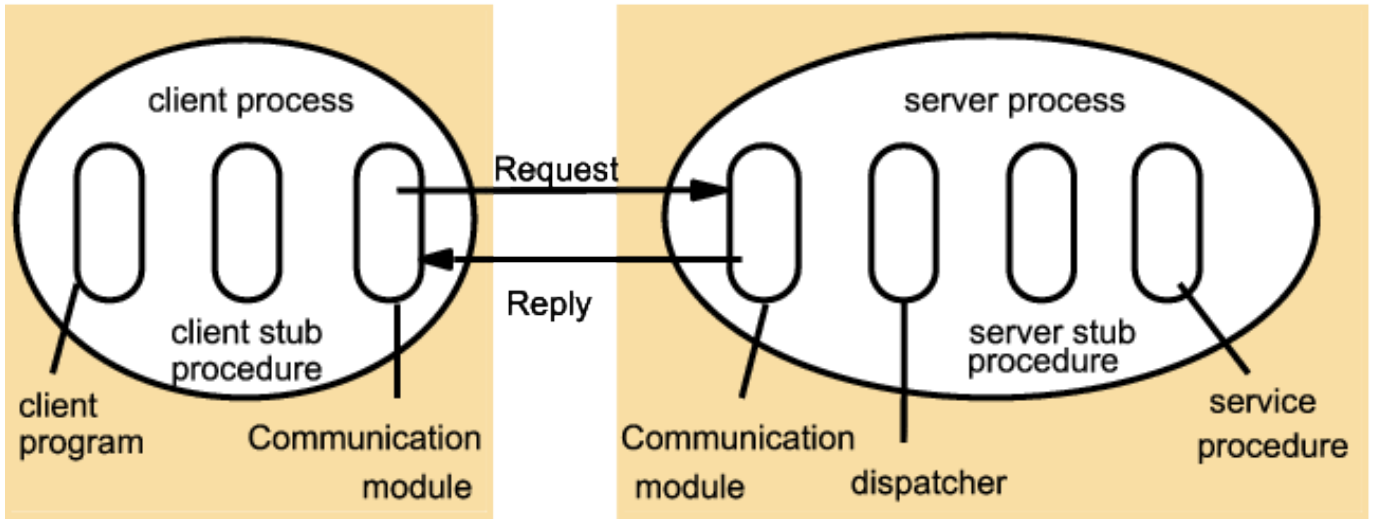
- HTTP protocol specifies the:
  - the messages involved in the protocol
  - the methods, arguments and results
  - the rules for marshalling messages
- Allows content negotiation - client specify the data format they can accept
- Allows authentication - based on credentials and challenges
- Original version of the protocol did not persist connections resulting in overloading the server and the network
- HTTP 1.1 uses persistent connections

- HTTP methods
  - `GET` - Request resources from a URL
  - `HEAD` - Identical to `GET` but does not return data
  - `POST` - Supplies data to the resources
  - `PUT` - Requests the data to be stored with the given URL
  - `DELETE` - Requests the server to delete the resource indentified with the given URL
  - `OPTIONS` - Server supplies the available options
  - `TRACE` - Server sends back the request message
- Requests and replies are marshalled into messages as ASCII text strings


method
URL or pathname
HTTP version
headers
message body
GET
//www.dcs.qmw.ac.uk/index.html
HTTP/ 1.1
HTTP version
status code
reason
headers
message body
HTTP/1.1
200
OK
resource data

Remote Procedure Call (RPC)

RPCs enable clients to execute procedures in server processes based on a defined service interface.



Communication Module Implements the desired design choices in terms of retransmission of requests, dealing with duplicates and retransmission of results.

Client Stub Procedure Behaves like a local procedure to the client. Marshals the procedure identifiers and arguments which is handed to the communication module. Unmarshalls the results in the reply.

Dispatcher Selects the server stub based on the procedure identifier and forwards the request to the server stub.

Server stub procedure Unmarshalls the arguments in the request message and forwards it to the service procedure. Marshalls the arguments in the result message and returns it to the client.

Case study: Sun RPC

Reference: UNIX Network programming, W. Richard Stevens, Prentice Hall, Inc.

Sun RPC includes:

- A standard way to define the remote interface using the interface definition language XDR (eXternal Data Representation)
- A compiler `rpcgen` for compiling the remote interface
- A run-time library

Case study: Sun RPC

Step 1: Define the interface using XDR (date.x)

```
 1 /*
 2  * date.x - Specification of remote date and time service.
 3  */
 4
 5 /*
 6  * Define 2 procedures:
 7  *  bin_date_1() returns the binary time and date (no arguments).
 8  *  str_date_1() takes a binary time and returns a human-readable string.
 9  */
10
11 program DATE_PROG {
12   version DATE_VERS {
13     long BIN_DATE(void) = 1;        /* procedure number = 1 */
14     string STR_DATE(long) = 2;      /* procedure number = 2 */
15   } = 1; /* version number = 1 */
16 } = 0x31234567;     /* program number = 0x31234567 */
```

Case study: Sun RPC

Step 2: Compile the interface

```
rpcgen date.x
```

This generates the header (date.h), server stub (date_svc.c) and client stub (date_clnt.c).

Case study: Sun RPC

Step 3: Implement the client (rdate.c)

```
 1 /*
 2  * rdate.c - client program for remote date service.
 3  */
 4 #include<stdio.h>
 5 #include<rpc/rpc.h>  /* standard RPC include file */
 6 #include"date.h"  /* this file is generated by rpcgen */
 7
 8 main(argc, argv)
 9 int argc;
10 char *argv[];
11 {
12   CLIENT  *cl;    /* RPC handle */
13   char  *server;
14   long  *lresult;  /* return value from bin_date_1() */
15   char  **sresult;  /* return value from str_date_1() */
16   if (argc != 2) {
17      fprintf(stderr, "usage: %s hostname\n", argv[0]);
18      exit(1);
19   }
20   server = argv[1];
```

```
 1   /*
 2    * Create the client "handle."
 3   */
 4   if ( (cl = clnt_create(server, DATE_PROG, DATE_VERS, "udp")) == NULL) {
 5    /*
 6    * Couldn't establish connection with server.
 7    */
 8   clnt_pcreateerror(server);
 9   exit(2);
10   }
11  /*
12  * First call the remote procedure "bin_date".
13  */
14   if ( (lresult = bin_date_1(NULL, cl)) == NULL) {
15     clnt_perror(cl, server);
16     exit(3);
17   }
18
19   printf("time on host %s = %ld\n", server, *lresult);
```

```
 1   /*
 2    * Now call the remote procedure "str_date".
 3   */
 4   if ( (sresult = str_date_1(lresult, cl)) == NULL) {
 5     clnt_perror(cl, server);
 6     exit(4);
 7   }
 8   printf("time on host %s = %s", server, *sresult);
 9   clnt_destroy(cl);    /* done with the handle */
10   exit(0);
11 }
```

Case study: Sun RPC

Step 4: Implement the server functions (date_proc.c)

```
 1 /*
 2  * dateproc.c - remote procedures; called by server stub.
 3  */
 4 #include  <rpc/rpc.h>  /* standard RPC include file */
 5 #include  "date.h"  /* this file is generated by rpcgen */
 6 /*
 7  * Return the binary date and time.
 8  */
 9 long *
10 bin_date_1()
11 {
12   static long  timeval;  /* must be static */
13   long    time();    /* Unix function */
14   timeval = time((long *) 0);
15   return(&timeval);
16 }
```

```
 1 /*
 2  * Convert a binary time and return a human readable string.
 3  */
 4 char **
 5 str_date_1(bintime)
 6 long  *bintime;
 7 {
 8   static char  *ptr;    /* must be static */
 9   char    *ctime();  /* Unix function */
10   ptr = ctime(bintime);    /* convert to local time */
11   return(&ptr);    /* return the address of pointer */
12 }
```

Case study: Sun RPC

Step 5: Compile the client and the server

```
gcc -o rdate rdate.c date_clnt.c -lrpcsvc -lnsl

gcc -o date_svc date_proc.c date_svc.c -lrpcsvc -lnsl
```

Case study: Sun RPC

Step 6: Run the server and client

```
./date_svc &

./rdate [hostname]
```

Object-Oriented Concepts

Objects consists of attributes and methods. Objects communicate with other objects by invoking methods, passing arguments and receiving results.

Object References can be used to access objects. Object references can be assigned to variables, passed as arguments and returned as results.

Interfaces define the methods that are available to external objects to invoke. Each method signature specifies the arguments and return values.

Actions - objects performing a particular task on a target object. An action could result in:

- The state of the object changed or queried
- A new object created
- Delegation of tasks to other objects

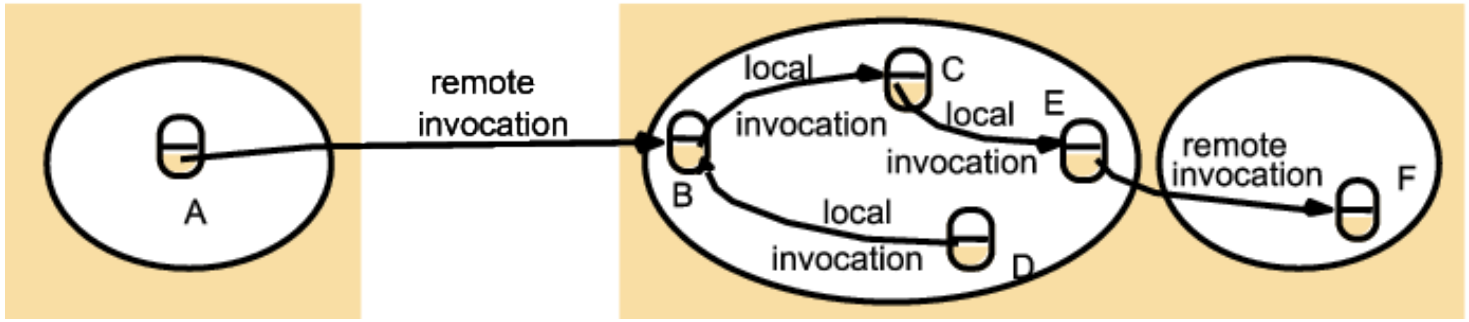
Exceptions are thrown when an error occurs. The calling program catches the exception.

Garbage collection is the process of releasing memory used by objects that are no longer in use. Can be automatic or explicitly done by the program.

Distributed Object Concepts

Remote Objects

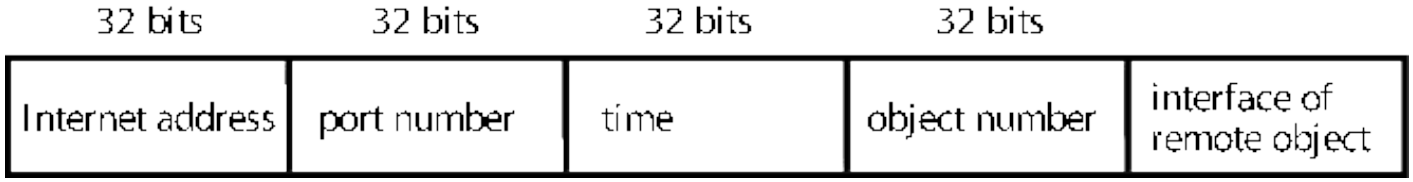
An object that can receive remote invocations is called a remote object. A remote object can receive remote invocations as well as local invocations. Remote objects can invoke methods in local objects as well as other remote objects.



Distributed Object Concepts

Remote Object Reference

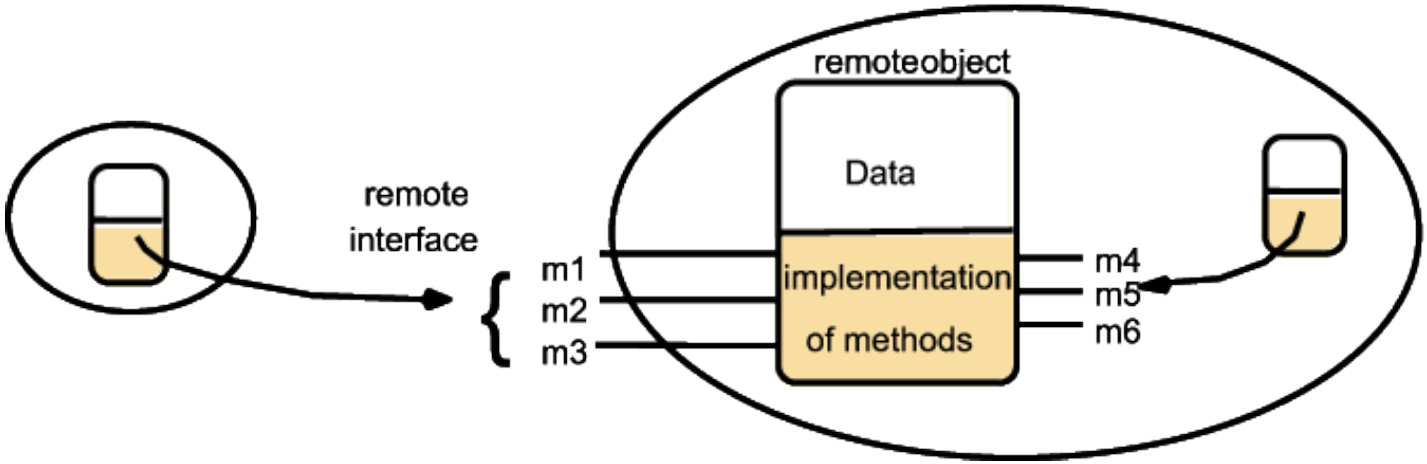
A remote object reference is a unique identifier that can be used throughout the distributed system for identifying an object. This is used for invoking methods in a remote object and can be passed as arguments or returned as results of a remote method invocation.



Distributed Object Concepts

Remote Interface

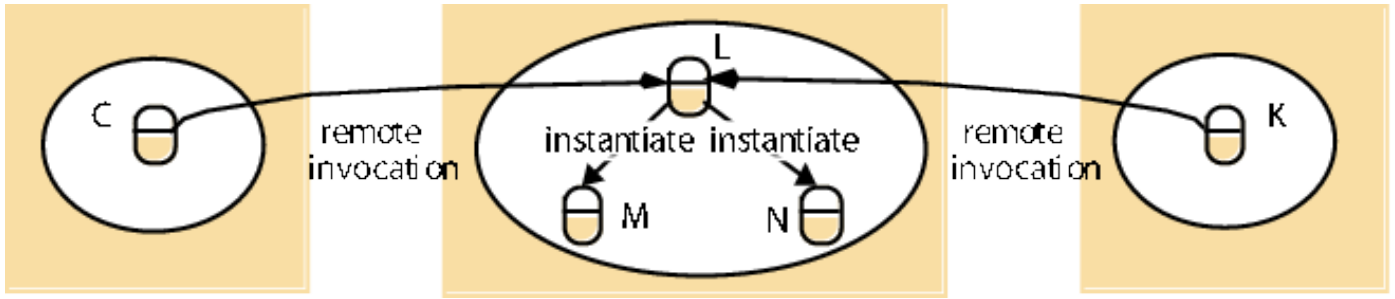
A remote interface defines the methods that can be invoked by external processes. Remote objects implement the remote interface.



Distributed Object Concepts

Actions in a distributed system

Actions can be performed on remote objects (objects in other processes of computers). An action could be executing a remote method defined in the remote interface or creating a new object in the target process. Actions are invoked using Remote Method Invocation (RMI).



Distributed Object Concepts

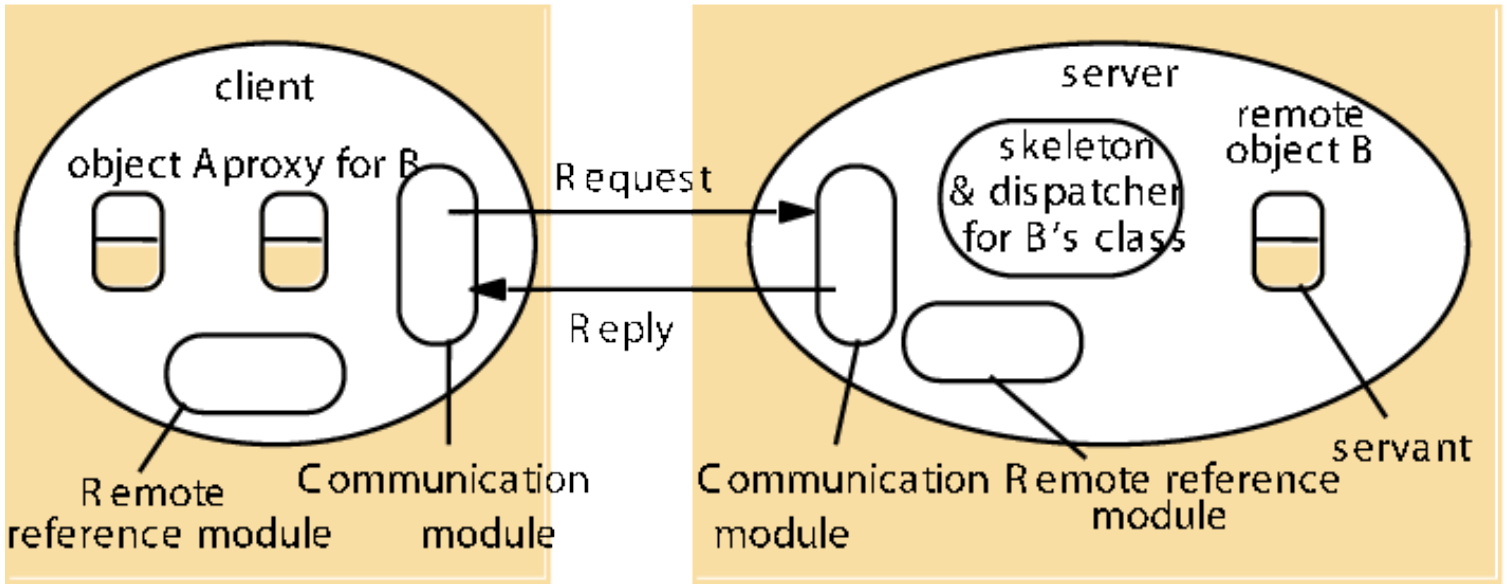
Garbage collection in a distributed system

Is achieved through reference counting.

Exceptions

Similar to local invocations, but special exceptions related to remote invocations are available (e.g. timeouts).

Implementation of RMI



The **Communication Module** is responsible for communicating messages (requests and replies) between the client and the server. It uses three fields from the message:

- message type
- request ID
- remote object reference

It is responsible for implementing the invocation semantics. The communication module queries the remote reference module to obtain the local reference of the object and passes the local reference to the dispatcher for the class.

The **Remote reference module** is responsible for:

- Creating remote object references
- Maintaining the remote object table which is used for translating between local and remote object references

The remote object table contains an entry for each:

- Remote object reference held by the process
- Local proxy

Entries are added to the remote object table when:

- A remote object reference is passed for the first time
- When a remote object reference is received and an entry is not present in the table

Servants are the objects in the process that receive the remote invocation.

The RMI software: This is a software layer that lies between the application and the communication and object reference modules. Following are the three main components.

- **Proxy**: Plays the role of a local object to the invoking object. There is a proxy for each remote object which is responsible for:
    - Marshalling the reference of the target object, its own method id and the arguments and forwarding them to the communication module.
    - Unmarshalling the results and forwarding them to the invoking object
- **Dispatcher**: There is one dispatcher for each remote object class. Is responsible for mapping to an appropriate method in the skeleton based on the method ID.
- **Skeleton**: Is responsible for:
    - Unmarshalling the arguments in the request and forwarding them to the servant.
    - Marshalling the results from the servant to be returned to the client.

Developing RMI Programs

Developing a RMI client-server program involves the following steps:

1. *Defining the interface for remote objects* - Interface is defined using the interface definition mechanism supported by the particular RMI software.

2. *Compiling the interface* - Compiling the interface generates the proxy, dispatcher and skeleton classes.

3. *Writing the server program* - The remote object classes are implemented and compiled with the classes for the dispatchers and skeletons. The server is also responsible for creating and initializing the objects and registering them with the binder.

4. *Writing client programs* - Client programs implement invoking code and contain proxies for all remote classes. Uses a binder to lookup for remote objects.

Dynamic invocation

Proxies are precompiled to the program and hence do not allow invocation of remote interfaces not known during compilation.

Dynamic invocation allows the invocation of a generic interface using a `doOperation` method.

Server and Client programs

A server program contains:

- classes for dispatchers and skeletons
- an *initialization* section for creating and initializing at least one of the servants
- code for registering some of the servants with the binder

A client program will contain the classes for all the proxies of remote objects.

Factory methods

- Servants cannot be created by remote invocation on constructors
- Servants are created during initialization or methods in a remote interface designed for this purpose
- **Factory method** is a method used to create servants and a **factory object** is an object with factory patterns

The binder

Client programs require a way to obtain the remote object reference of the remote objects in the server.

A **binder** is a service in a distributed system that supports this functionality.

A binder maintains a table containing mappings from textual names to object references.

Servers register their remote objects (by name) with the binder. Clients look them up by name.

Activation of remote objects

A remote object is *active* if it is available for invocation in the process.

A remote object is *passive* if it is not currently active but can be made active. A passive object contains:

- the implementation of the methods
- its state in marshalled form

Object Location

- Remote object references are used for addressing objects
- The object reference contains the Internet address and the port number of the process that created the remote object
- This restricts the object to reside within the same process
- A **location server** allows clients to locate objects based on the remote object reference

Case Study: Java RMI

Steps to develop a java RMI server:

1. Specify the Remote Interface
2. Implement the Servant Class
3. Compile the Interface and Servant classes
4. Generate skeleton and stub classes
5. Implement the server
6. Compile the server

Steps to develop a java RMI client:

1. Implement the client program
2. Compile the client program

Java RMI Server Example

Specify the Remote Interface

```
1 import java.util.*;
2 import java.rmi.*;
3
4 public interface RemoteMath extends Remote {
5   double add( double i, double j ) throws RemoteException;
6   double subtract( double i, double j ) throws RemoteException;
7 }
```

Java RMI Server Example

Implement the Servant Class

```
 1 import java.rmi.*;
 2 import java.rmi.server.UnicastRemoteObject;
 3 public class RemoteMathServant
 4        extends UnicastRemoteObject
 5        implements RemoteMath {
 6   int _numberOfComputations;
 7
 8   public RemoteMathServant( ) throws RemoteException {
 9      _numberOfComputations=0;
10   }
11   public double add ( double i, double j ) {
12     _numberOfComputations++;
13     System.out.println("Number of computations performed so far = "
14         + _numberOfComputations);
15     return (i+j);
16   }
17   public double subtract ( double i, double j ) {
18     _numberOfComputations++;
19     System.out.println("Number of computations performed so far = "
20         + _numberOfComputations);
21     return (i-j);
22   }
23 }
```

Java RMI Server Example

Compile the Interface and Servant classes

```
javac RemoteMath.java
javac RemoteMathServant.java
```

Java RMI Server Example

Generate skeleton and stub classes

```
rmic RemoteMathServant
```

Java RMI Server Example

Implement the server

```
 1 import java.rmi.*;
 2 public class MathServer {
 3   public static void main(String args[]){
 4     System.setSecurityManager(new RMISecurityManager());
 5     System.out.println("Main OK");
 6     try{
 7       RemoteMath aMath = new RemoteMathServant();
 8       System.out.println("After create");
 9       Naming.rebind("Compute", aMath);
10       System.out.println("Math server ready");
11     }catch(Exception e) {
12       System.out.println("MathServer server main " +
13                                             e.getMessage());
14     }
15   }
16 }
```

Java RMI Server Example

Compile the server

```
javac MathServer.java
```

Class path should include the classes for the skeleton classes generated by the `rmic` command.

Java RMI Client Example

Implement the client program

```
 1 import java.rmi.*;
 2 import java.rmi.server.*;
 3
 4 public class MathClient {
 5
 6   public static void main(String[] args) {
 7     try {
 8       if(System.getSecurityManager() == null) {
 9         System.setSecurityManager( new RMISecurityManager() );
10       }
11       RemoteMath math = (RemoteMath)Naming.lookup("rmi://localhost/Compute");
12       System.out.println( "1.7 + 2.8 =  "
13                           + math.add(1.7, 2.8) );
14       System.out.println( "6.7 - 2.3 =  "
15                           + math.subtract(6.7, 2.3) );
16     }
17     catch( Exception e ) {
18       System.out.println( e );
19     }
20   }
21 }
```

Compile the client program

```
javac MathClient.java
```

Class path should include the classes for the stub classes generated by the `rmic` command.

Running the Server and Client

1) Develop a security policy file

```
grant {
 permission java.security.AllPermission "", "";
};
```

2) Start RMI registry

```
rmiregistry &
```

3) Start server

```
java -Djava.security.policy=policyfile MathServer
```

4) Start client

```
java -Djava.security.policy=policyfile MathClient
```

Distributed garbage collection

A distributed garbage collector ensures that a remote object continues to exist as long as there are local or remote object references to the object. If no references exist then the object will be removed and the memory will be released.

Java's distributed garbage collection mechanism:

- The server maintains a list of names that hold remote object references to the object `B.holders`
- When a client `A` receives a reference to a remote object `B` it makes a `addRef(B)` invocation to the server which adds `A` to `B.holders`
- When `A`'s garbage collector notices that no references exist for the proxy of remote object `B` it makes a `removeRef(B)` invocation to the server which removes `A` from `B.holders`
- When `B.holders` is empty, the server's local garbage collector releases the space

Note: Race conditions have to be taken care of.