Lecture 6
From RPC to RMI (Remote Method Invocation)

(June-23,27-2020. Chapter 5)

Lecture Outline

```
1. Introduction
2. Request-reply protocols
3. Remote procedure calls
4. Distributed objects and RMI (remote method invocation)
```

1. Introduction

(1) Client-server communications are typical in distributed/cloud computing

a. Client-server communications usually employ the *request-reply* protocol.

b. Request-reply communications can be synchronous or asynchronous. But most of time it is synchronous – a client normally suspends its operations while waiting for reply from server.

c. Request-reply communications are independent of underlying transport layer protocols. Both UDP and TCP can be used.

(2) From RPC to RMI

a. RPC

b. Object oriented paradigm

c. Distributed objects and RMI

2. Request-reply protocols

(1) Typical to support the roles and message exchanges in typical client-server interactions

a. Illustration: Fig.5.2, p.187

(a) The presented protocol is generic. However most RPC and RMI implementations support similar request-reply protocols.

(b) Three comm. primitives (Fig.5.3, p.188):

i. *doOperation* method: performs the action of invoking a remote method (i.e. sending a request).

ii. *getRequest* method: is used by a remote server process to receive service requests.

iii. *sendReply* method: sends the reply message to the client.

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.`

https://www.ibm.com/support/knowledgecenter/en/ssw_aix_71/network/sctp_intro.html

b. Can be both synchonous and asynchronous

c. Transport layer can be either UDP or TCP (or SCTP)

d. UDP implemented RR protocol: less comm. overhead

   (a) No explicit ack is needed, as replies can serve as acks.

   (b) TCP connection and termination involves seven message exchanges.

   (c) There is no need of flow control (available in TCP) as most remote invocations are short in duration.

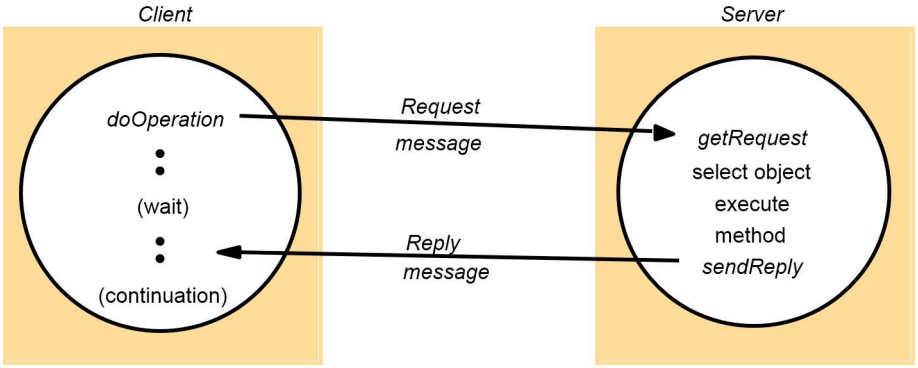


Figure 34: Request-reply communications (Fig.5.2, p.187)

(2) Structure of request-reply messages: Fig.5.4, p.188

messageType	*int (0=Request, 1=Reply)*
requestID	*int*
remoteReference	*RemoteRef*
operationID	*int or Operation*
arguments	*//array of bytes*

Figure 5.4 Request-reply message structure

a. *requestId*: (Note: the book showed a *requestId* in Fig.5.4. It then said that a *message identifier* contains a *requestId*) it consists of two parts: a *requestId* and an identifier of the sender process (its IP address and port number). *requestId* is taken from an increasing sequences of integers. A retransmitted request will carry the same requestId as in the original request.

b. *remoteReference*: this is remote object reference. It may consist of remote server's IP, port number, together with remote object ID.

c. Possible errors. For each request, the following events might happen:

   (a) Request message is not received by the server:

      i. The request message is lost during comm.

      ii. The server process is not running.

      iii. The computer hosting the server process is not running.

   (b) The reply message does not reach the client

      i. The reply message is lost during comm.

   (c) The server process crashes and restarts

   (d) The client process crashes and restarts

   (e) The request message is delayed

   (f) The request message is corrupted

   (g) The reply message is corrupted

   * If (a), or (c), or (f) occurs, the request has not been served.
   * If (b) or (g) occurs, the request has been served.
   * If (c) or (d), the request may or may not have been served.

   The critical observation is that, from a client's perspective, it has no way to know what exactly has happened that has caused it not to be able to receive the reply.

d. *Idempotent operations* and semantics of request-reply protocols

   (a) An operation is *idempotent* if the effect of performing the operation repeatedly is same as that of performing it exactly once. Example idempotent operations:

      i. Access a WEB page;

      ii. The operation of adding an element to a set. The property of sets guarantee idempotency.

   (b) *At-least-once, at-most-once, maybe*, and *exactly-once* semantics of request-reply protocol

i. *At-lease-once*: a request is guaranteed to be served by a server *at least once*. Example application: client may request a remote email server to send a email message to a specific user to notify approval of the user's application. This semantics is suitable for idempotent operations.

ii. *At-most-once*: a request is guaranteed to be served by a server at most once. But it may not be served at all. Not very common.

iii. *Maybe*: a request may or may not be served. Not very common. It arises when no any fault tolerance mechanism is taken.

iv. *Exactly-once*: a request is guaranteed to be served by a server *exactly once*. Too many applications require this semantics.

(c) Comments:

i. Exactly-once semantics is from local procedure calls. Combination of at-least-once and at-most-once semantics provides exactly-once semantics.

ii. Most IPC implementations do not support exactly-once semantics due to its overhead. It's left to applications to apply additional fault tolerance mechanisms to achieve exactly-once semantics.

This topic will be re-examined in Section 6 "Distributed objects and RMI" in this lecture.

e. Failure model of the request-reply protocol.

(a) A request-reply protocol based on UDP that does not add any mechanism to enhance reliability by itself suffers from two well-know failures:

i. Omission failures;

ii. Non-delivery of messages (violation of *validity* in our failure model).

Such a protocol can only support *at-most-once* or *maybe* semantics. It cannot provide *at-least-once* or *exactly-once* semantics.

(b) *Timeout* and *retransmission*.

i. Timeout and retransmission are essential for a request-reply protocol to provide at-least-once or exactly-once semantics (although for the latter, additional mechanism is needed). The *doOperation* uses timeout to detect loss of requests or replies.

ii. For at-most-once and maybe semantics, timeout alone is sufficient.

(c) *Detecting and discarding duplicate request messages*.

i. For non-idempotent operations, a server in a client-server comm. must be able to detect a duplicate request message and discard it. Otherwise inconsistency may arise.

ii. A server should also re-transmit the reply when it detects a duplicate request message. This will keep the client from performing pair of time-out/retransmit operations repeatedly.

(d) *History*.

i. In order for a server to able to detect duplicate requests and retransmit replies, it must maintain, a record of executions of requests. Such a record is called *history*.

ii. When a request from a specific client arrives, the server will check the newly arrived request against the history. If that request is new, it will be served. If that request is a duplicate, a stored copy of reply will be fetched from the history and retransmitted to the client.

iii. If the client-server comm is synchronous, the server only has to keep one copy of completed request for each client. A new request from a client signals that that client has received reply to its previous request. Hence the server can purge the history to delete reply for that client.

(3) RPC exchange protocols (Fig.5.5, p.191)

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

Figure 5.5 RPC exchange protocols

a. The three RPC exchange protocols are:

(a) The request (R) protocol

(b) The request/reply (RR) protocol

(c) The request/reply/acknowledge (RRA) protocol

b. The three RPC exchange protocols were special request-reply protocols originally designed for RPC. But its semantics also applies to request-reply protocols in general.

c. **R** protocol. Used when no value has to be returned in response to a request and the client does not require/care confirmation from the server. It implements the *maybe* and *at-most-once* semantics. This protocol is the least reliable and least expensive.

d. **RR** protocol. Used in most RPC systems. This protocols implements the *exactly-once* semantics.

(a) In order to filter out duplicate requests a server must keep three separate histories (also called *pools*) containing information of all outstanding requests:

i. *Completion pool*: which contains those requests, together with their replies, that have been *recently* completed;

ii. *Execution pool*: which contains those requests that are currently being served;

iii. *Waiting pool*: which contains those requests that were received but are waiting their turn to be served.

(b) A newly arrived RPC request is checked against the three pools as follows:

i. If there is a request in the waiting pool or execution pool with the same *messageId* as the newly arrived request, the newly arrived request message is simply discarded. This is a duplicate request whose original request has been received but has not been completed.

ii. If the messageId of the newly received request matches that of a request in the completion pool, this newly received request is also a duplicate, hence discarded. However, the reply for that request in the completion pool will be fetched and retransmitted to the client. This is a duplicate request whose original request has been completed.

iii. If the newly arrived request came from some process with some *SourceAddress* $= x$ such that the *messageId* is not in the completion pool, then the newly arrived request is either put into the waiting pool or given the CPU for execution. In addition, entry in the completion pool with the *SourceAddress* equal to x is purged, because the client must have received the reply to its previous request.

e. **RRA** protocol.

(a) Problems with RR protocols: the waiting pool may grow very large, consuming system resources (a completed request is kept in the completion pool until the initiating client makes a new request. However, a client may never send new requests after initial interactions). RRA is a protocol that can remedy this problem.

(b) The RRA protocol:

- An acknowledgment message is required from the client to which a reply message has been sent (by the server). The ack message contains messageId of the original request message.
- After receiving the ack message the server can free all memory associated with that particular client's request.

(c) Alternatively, the server can adopt the strategy of holding completed requests in its histories for a certain period τ of time and deletes them after τ time. Deciding the value of τ is critical to the correctness and performance of a

request-reply protocol. Too short: may not ensure at-most-once semantics. Too long: large waiting pool.

(4) Request-reply protocols implementation using TCP streams

a. Problems of datagram communications: (besides unreliability) the buffers sizes at the sender and receiver are difficult to decide.

(a) Certain applications require large buffers that are difficult to support by clients or servers.

(b) If a client supports small buffers, an application request that is significantly larger than size of the buffers will result in multiple datagram packets. An UDP based request-reply protocol has to deal with the issue of disassembling and reassembling these scrambled packets.

b. Benefits of TCP based request-reply protocols

(a) TCP is a stream protocol, hence buffer sizes no longer present a problem.

(b) TCP is reliable. Therefore retransmission is not an issue here.

(c) TCP significantly simplifies the implementation of request-reply protocols. If a client intends to communicate with a server for a relatively long time period, the overhead of TCP connection establishment and termination is negligible.

c. The request-reply protocol in Java RMI is TCP based. A server *extends* the *UnicastRemoteObject* class, which provides support for point-to-point active object references (invocations, parameters, and results) using TCP stream.

(5) Example: the HTTP (hypertext transfer protocol) protocol

a. A Web server manages different resources in different ways:

(a) as data such as texts and images of an HTML page.

(b) as program such *cgi* scripts/programs and *servlets*, which run on the web server.

b. HTTP is a request-reply protocol. It specifies formats of messages involved in a request-reply exchange, the supported methods on Web servers, arguments and results and rules for representing (i.e. marshalling) them in messages.

(a) HTTP supports a fixed set of methods that are applicable to all resources managed by a Web server. This is different from other protocols where different resources (objects) have different methods.

(b) HTTP supports content negotiations (what data format/representation a client can accept) and password based authentication mechanism.

(c) HTTP is based on TCP.

i. HTTP 1.0 would open/close a TCP session for every pair of request/reply exchange between a client and server (high overhead).

ii. HTTP 1.1 (and newer) uses *persistent connections* which allows a connection to be up through a sequence of request/reply exchanges. A persistent connection can be explicitly closed by either a client or server.

c. *Marshalling* in HTTP

(a) Requests and replies are marshalled into ASCII text strings unless specified otherwise.

(b) Requests and replies can also be communicated as byte sequences and compressed.

(c) Multimedia data are sent and received as MIME formed structures, which has a prefix about its MIME data type and subtype (such as *text/html, image/gif*). This information allows the receiving process to use appropriate applications to handle the data.

d. HTTP *messages*

i. Request messages (Fig.5.6, p.193)

method	*URL*	*HTTP version*	*headers*	*message body*
GET	//www.dcs.qmul.ac.uk/index.html	HTTP/1.1		

Figure 5.6 HTTP *request* message

ii. Reply messages (Fig.5.7, p.195)

HTTP version	*status code*	*reason*	*headers*	*message body*
HTTP/1.1	200	OK		

Figure 5.7 HTTP *reply* message

iii. *header* field: contains request modifiers and client information: for reload, a condition (date) of retrieve a web page; for password based authentication, a encrypted password or other credentials.

iv. *message body* field: itself may also be structured and contain its own headers so that the client may invoke different applications to handle different types of data in it.

e. HTTP *methods*

(a) *GET*: requests all resources specified by the given URL. An HTML page located by an URL may refer to programs (CGI or Servlet), which will be invoked and results of the invocation will be returned.

(b) *HEAD*: Similar to GET. But it will not retrieve message body. It can be used by a proxy server or cache server to check freshness of a resource.

(c) *POST*: Places a document to a Web server. The URL that will be used to process the document is specified. Example applications that uses this method include:

i. Posting a message to a discussion group or message board.

ii. Submitting an online application, an online electronic payment, or an online order request.

A POST request normally will invoke a CGI script or Servlet on the Web server.

(d) *PUT*: specifies that the data supplied in the request should be stored with the given URL as identifier.

(e) *DELETE*: requests the remote Web server to delete the resource identified by the specified URL. Potentially insecure and disabled on many Web server.

(f) *OPTIONS*: mainly used by a server to declare to a client the list of methods that are allowed to be applied to the resources with the specified URL.

(g) *TRACE*: the server will echo the request sent by a client. For debugging purpose.

3. Remote procedure calls

(1) Motivations

a. A process that wants to access to a service hosted on a remote computer should be able to make a service request that specifies the *service name* and possibly some needed parameters, just like making a local procedure call (LPC). The calling process may not and does not want to know the physical location of the service.

b. Advantages of RPC.

The concept of RPC and its implementation are important steps toward *access transparency*. RPC relieves applications from the details of locations of available services, communications, transmission errors, and various failures.

(2) Syntax and semantics of local procedure calls

a. Before a procedure is actually invoked, a new environment is created, which includes the actual parameters passed, local variables, and variables in the program enclosing the called procedure.

b. The caller is suspended upon the call.

c. When the called procedure completes, it returns the results to the calling procedure and terminates.

d. The calling procedure resumes its execution.

(3) Syntax and semantics of RPCs:

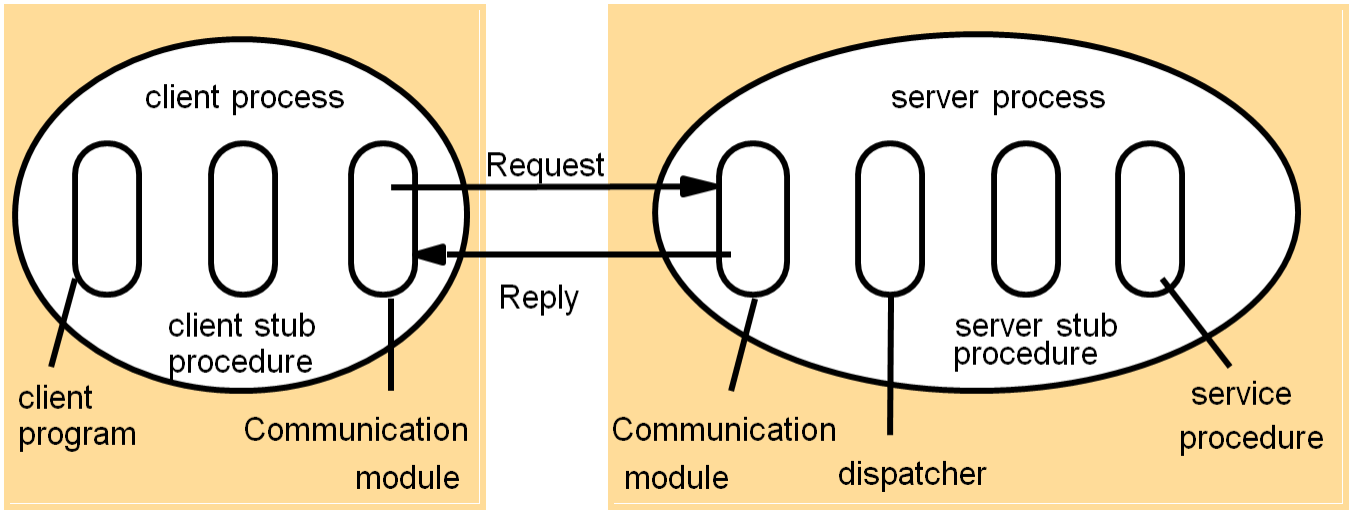


Figure 35: Role of client and server stub procedures in RPC (Fig.5.10, p.201)

a. The local process makes a RPC, providing the name of a remote procedure, and possibly a list of parameters.

b. The RPC will be caught by a communication daemon, which calls the *client stub* procedure. The client stub prepares a *request message* and sends it to the remote host.

c. A corresponding server stub procedure at the remote host accepts the request message, and prepares a local procedure call that implements the requested service. It then initiates the local procedure. The client and server stubs are illustrated in Fig.5.8, p.198.

d. The LPC at the remote host completes, and the results are given to the server stub at the remote host.

e. The server stub at the remote host prepares a *reply message* and sends it back to the client stub at the local host.

f. Upon receiving the reply message, the client stub will unpack the message and send the results back to the local process.

(4) Major issues of RPCs

a. *Uniform call semantics* – transparency problem. Should communication failures and long delays be hidden?

b. *Type checking* – strong type checking available in local procedure call implementation should also be used for RPCs.这是什么?

c. *Full parameter functionality* – all primitive data types should be allowed. Supports to structured and application defined data types must also be considered.

d. *Concurrency control and exception handling* – not fundamental, but should be considered and implemented if possible.

e. *Binding* – a remote procedure name called by a client will be eventually associated with a local procedure at a remote host. When will this association occur?

(a) *Static binding*: bind it at compilation time. when a client program is compiled, every remote procedure name is bound for each RPC.

i. Advantages: simple and easy to implement.

ii. Problems: programs need to be recompiled when the service locations change.

(b) *Dynamic binding*: delay the binding until run-time

i. Advantages: location transparency is maintained because no recompilation is necessary. 什么是location transparency???

ii. Problems: each site must maintain a table that maps every remote procedure name to a list of possible servers. The table must consulted for each RPC at run-time and updated frequently to reflect the real configuration.

(c) Binding based on name servers

i. A server name of a remote service must be registered

ii. Remote procedure name is resolved through name servers 什么意思?

iii. Flexible. Could be dynamic or static.

(5) Interfaces in distributed computing

a. Interface provides abstraction and supports transparency

b. Interface in RPC: an interface must be defined for a RPC system. Each remote procedure must be assigned a unique procedure id and have an entry in the interface which must contain the following information:

(a) The types of parameters and results, for type checking. This is just like in LPC.

(b) Different languages have different notions of data types. The interface must contain adequate information so that stubs can perform appropriate type conversions if necessary (real in Pascal, float and double in C).

(c) CORBA IDL interface: Fig. 5.8, p.197

```
// 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();
};
```

Fig. 5.8 CORBA IDL example

Fault tolerance measures			*Call 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 36: Call semantic s(Fig.5.9, p.198)

(6) RPC call semantics: Fig.5.9, p.198

a. Retry request semantics: whether to retransmit the request message (and how many tries?)

b. Duplicate filtering: whether to filter out duplicates (in the case of retry semantics)

c. Retransmission of results: whether to keep track of reply results

(7) Transparency

a. RPC is a high level client-server IPC facility. As such it faces all possible errors a request-reply protocol has to deal with (cf. RPC protocols in Section 4 of this lecture)

b. Many factors can affect transparency support in RPC:

(a) Comm. failures, which cause omission failures (loss of request and result messages).

(b) Crash failures of server hosts.

(c) Delay of message delivery.

Supporting high degree of transparency is challenging and is impossible in most cases.

(8) Concurrency

a. Blocking calls and their effects

(a) By definition, a client is blocked after a RPC request until a reply message is received.

(b) Problems: degree of parallelism is reduced and hence system performance may degrade. A process may be both a client and server. Blocking a client may prevent it from accepting incoming service requests.

b. *Threads*: a collection of processes created by a single parent that share the same address space as the parent. Benefits of using threads:

(a) Unlike conventional processes, threads have little context switching overhead. 什么意思并且， 为什么?

(b) A client may create a collection of threads. Some them are responsible to accept incoming requests and others are responsible for making outgoing requests to other servers.

(c) Similarly, a server may also create multiple threads to serve requests concurrently.

(9) Implementation of RPC: Fig. 5.10, p.201

a. External data representation

b. Methods of passing parameters and results in RPCs

a. *Input* parameters are passed by value

b. *Output* parameters are passed by reference

c. Marshalling. The client and server stub processes will perform marshalling/unmarshalling for applications.

d. Interface declaration

f. Stubs hide the remote nature of RPC. A stub has the following responsibilities:

(a) For each RPC creates an appropriate message that contains the arguments, message id, client id, message type, request id, and procedure id. A typical message is of the form. Note: each retransmitted RPC request message will carry the same messageId but different requestId fields.

```
type message record
        MessageType: {request, reply};
        MessageId: integer; (* one per message request*)
        RequestId: integer; (* one per RPC request *)
        SourceAddress: port; (* network address of client process *)
        ProcedureId: integer; (* unique server id in interface *)
        Argument: flattenedList;
    end;
```

(b) Perform type checking and conversions according to the interface specification and declaration.
Example. The remote procedure has as its parameters an array of integers. The client calls the stub with an array name. Then the stub must get the values of the array and pass them as individual arguments.

(c) Initiate a message exchange request. Usually the message exchange will be done by a system *message module* on behalf of the stub.

(d) At the server end, the server stub will unpack the messages. Each host has a *dispatcher*, which, for each incoming request message, will look at the message for a *procedureId*, and *dispatches* the request to the intended server.

(e) When the remote procedure completes, the results are submitted to the server stub. The results are marshalled and a reply message is prepared by the server stub. It then sends the reply message to the client host through the message module at the remote host.

(f) The client stub at the client host receives and unpack the reply message and delivers the results to the client.

Stub hides and encapsulates all the details of type conversions and communications, making RPCs just like LPCs.

g. Special data structures: pointers, arrays, linked lists and etc.

(a) Prohibited by many RPC systems.

(b) Even if supported by some RPC systems, their semantics are much more restrictive than in LPCs.

什么意思?

Example. Linked lists: copy it and pass the whole; created at remote node; send back as results.

4. Distributed objects and RMI (remote method invocation)

(1) The object model

a. **Objects**. An object contains a collection of of data items and a set of exported (also called public) methods for accessing and manipulating the data items.

(a) Data items in an object can be internal (private) or external (public). Private data items can only be accessed through exported methods. Therefore private data items and ways to access them are *protected*.

(b) Similarly methods can be private or public and public objects can be invoked by applications.

(c) Arguments and return values of each exported method is clearly defined.

(d) The name, argument types and their positions, and type of return value of a method together is termed as *signature* of that method. *Method overloading* allows the same method name to be associated with several different signatures to provide maximum convenience.

(e) The selection of data structures for data items and implementation of methods are also encapsulated. They can be changed if necessary without affecting object users as long as declarations of exported methods haven't been changed.

b. **Object references**. Objects are accessed through object references.

(a) Informally speaking, an object reference (which is given as the name of a class instance variable in C++ and Java) is a pointer that points to a storage location that contains specification of the object, values of each data item in the object, and code for each method (private or public).

(b) An object reference is internally represented an *object number* within each process (like file descriptors).

(c) To invoke a method in a given object, an application supplies an object reference to the object and the method name (plus appropriate arguments as specified in the method declaration). An object whose method is being invoked is called the *target* or *receiver*.

(d) Object references are *first-class* values – they may be assigned to variables, passed as arguments and as return values.

c. **Interfaces**.

(a) An interface contains definitions of signatures of a set of methods and possibly some data types that are used by arguments and return values of some methods . An interface does not contain implementation code for each method in it.

(b) The information in an interface is adequate for programmers to write applications that will invoke methods contained in the interface.

(c) Some OO programming languages like Java allow a class to have multiple interfaces, one for each specific type of applications.

(d) Interfaces do not contain constructors.

d. **Actions**. An action is an invocation of a method of an object by another object.

(a) Besides object reference, an invocation contains all the information in the signature of the invoked method.

(b) Effects of an action.

i. The target will execute the method and send return values back to the invoker.

ii. The execution of the specified method may cause changes in the state of the storage associated with the invoked object. Other objects may be invoked in the course of execution.

e. **Exceptions**. Exceptions are unexpected conditions that are raised during object invoking.

(a) The concept of exceptions is similar to the concept of *signals* in UNIX systems. For each exception, an implementation has its default ways to handle it. Applications can specify their own methods, which override the default ways, to handle exceptions.

(b) Use of exceptions enhances debugging and free programmers from details of dealing with each possible error.

f. **Garbage collection**. When an object is not referenced by any process, it becomes garbage. Garbage should be collected by reclaiming all physical resources (storage) allocated to it. Some OO languages like Java can automatically detect and collect garbages, while others like C++ do not have this support. In the latter case, applications have to free resources explicitly.

(2) Distributed objects and remote interfaces

a. Besides all the characteristics and issues of normal objects as discussed above (object interfaces and object references), distributed objects have its own unique properties and issues.

(a) Distributed objects are physically distributed on different computers and managed by different processes. This further enhances encapsulation.

(b) An invocation of a method of a distributed object involves an IPC message exchange.

b. Distributed processes can access distributed objects through client-server models using request-reply protocols.

(3) The distributed object model

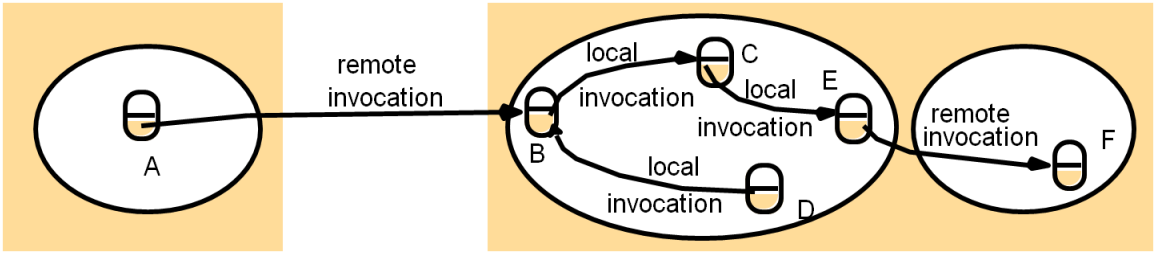


Figure 37: Remote and local method invocations (Fig.5.12, p.207)

a. The model (Fig.5.12, p.207)
   (a) A DS consists of a collection processes, each of which manages a collection of objects. Some of these objects, termed as *remote objects*, are remotely invokable, while the rest can only receive local invocations.
   (b) Method invocations between objects in different processes are called *remote method invocations* (RMI).
   (c) Two fundamental concepts of distributed object model: *remote object reference* and *remote interface*.

b. Remote object references: Fig.4.13,p.169
   (a) A remote object reference is an extension of local object reference. It is an identifier that can be used throughout a DS to refer and identify a particular unique remote object.
   (b) Fig.4.13 illustrates elements of a remote object reference.
      i. The *IP address* determines the remote host.
      ii. The *port number* and *time* components uniquely identify a process on the remote host.
      iii. A process will assign an *object number* (unique within the process) for every object created in the process.
      iv. The *interface of remote object* is provided so that any process, which receives a remote object reference as an argument or result of a remote invocation, can use the specified remote object interface to find out the methods exported by the remote object. 这是什么

32 bits	*32 bits*	*32 bits*	*32 bits*	
Internet address	port number	time	object number	interface of remote object

Figure 4.13 Representation of a remote object reference

c. Remote interfaces: Fig.5.13, p.208

(a) The concept of remote interfaces extends that of normal interfaces.

(b) Each remote object supports two groups of methods: remote invocable methods and those that can only be invoked locally. A remote interface specify all those remote invocable methods.

(c) In Java RMI, a remote interface is declared the same as a normal interface except that it must *Extends* the interface java.rmi.Remote to *acquire* remote invocable ability for all methods listed in the interface.

(d) CORBA IDL, Java IDL, and DCE IDL all support interface definitions. All objects (or procedures in DCE) can be invoked remotely.

d. Actions in distributed object system: Fig.5.14, p.209

(a) A method invocation may cause cascading remote object invocations by different processes hosted on different computers.

(b) The objects involved in such a chain of RMI may reside on different processes (elements).

(c) When an object is *instantiated* by a RMI, the new object usually resides on within the element where the instantiation is requested.

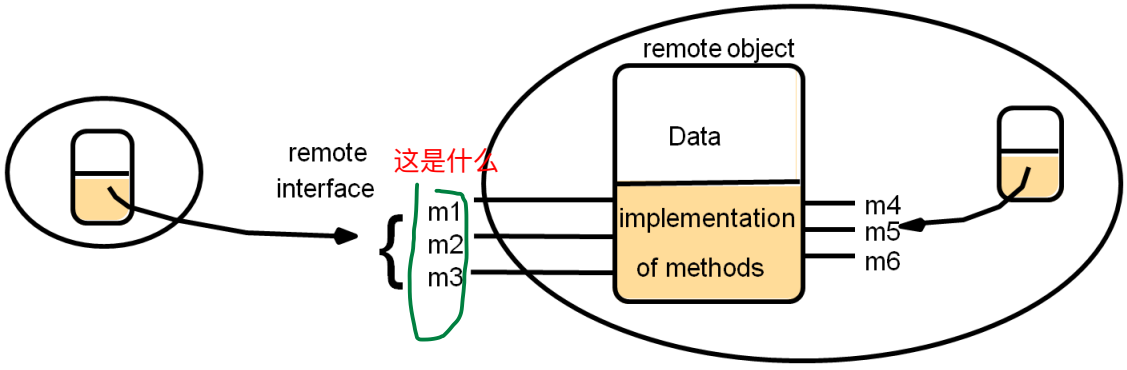


Figure 38: A remote object and its remote interface (Fig.5.13, p.208)

(4) Design issues of RMI

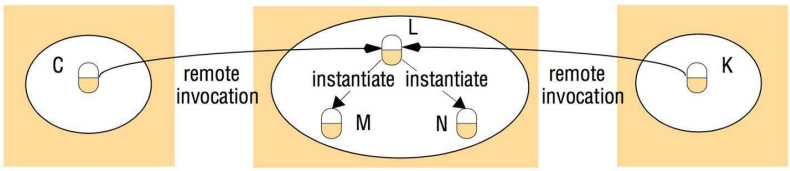


Figure 39: Instantiation of remote objects (Fig.5.14, p.209)

a. RMI invocation semantics. RMI can be viewed as a special form of client-server communications. Therefore the *at-least-once, at-most-once, maybe* and *exactly* semantics of request-reply protocols are also applicable to RMI (Cf. Section 4 of this lecture). Fig.5.9, p.198 summarizes RMI invocation semantics.

   (a) *Maybe invocation semantics*. The method in an RMI request may or may have not been executed. The invoker cannot be sure either way.

      i. This is the least reliable semantics. An RMI design that does not take any fault tolerance mechanisms (timeout, retransmissions, histories) will lead to this semantics. The server does not keep track if a RMI has been successfully completed and the client has no way to know one way or another.

      ii. It suffers: (i) omission failures if the invocation or result message is lost; (ii) crash failures when the server hosting the remote object fails. A crash failure may occur either before or after the method is executed.

      iii. Although this semantics is the least expensive to implement, it also has the least applicability. An example application using this semantics is sending a fun email message to a group of friends.

   (b) *At-least-once invocation semantics*.

      i. With this semantics, after an RMI the invoker will either receive a result, or it will receive an exception (such as a timeout) informing it that no result is received. In the former case, the invoker knows that the method has been executed at least once (possibly more than once!), and in the latter case, it knows that the method has been either executed once or not at all. In the latter case, retransmission will be used until a result is received.

      网页就是一个好例子

      ii. This semantics is suitable for *idempotent* RMI methods – methods whose multiple executions have the same effects as a single execution

      iii. It suffers: (ii) crash failures when the server hosting the remote object fails; (ii) arbitrary fails – multiple executions of non-idempotent RMI methods will cause inconsistency.

   (c) *At-most-once invocation semantics*. With this semantics, after an RMI the invoker will either receive a result, or it will receive an exception (such as

a timeout) informing it that no result is received. In the former case, the invoker knows that the method has been executed *exactly once*, and in the latter case, it knows that the method has been either executed once or not at all. It is up to the application to decide what to do next in the second case.

b. Transparency. This issue is very similar to that in RPC (cf. Section 5 of this lecture)

   (a) A RMI design supporting IDL can allow programmers to have a choice of calling semantics (at-least-once and etc.) so that a suitable semantics can be selected according to requirements of applications.

   (b) RMI designs normally allow applications to place different exceptions handlers to handle different failures.

   (c) The consensus is that an RMI should provide transparent syntax, but should specify the remote nature in interfaces. Java RMI uses the Remote interface to achieve this.

(5) Implementation issues of RMI

a. The main objects and modules in a RMI implementations are shown in Fig.5.15, p.210.

b. **Communication module**. This is the same module as in RPC (Fig.5.10). This module is responsible for actual receiving and sending of RMI request and reply messages, whose structure is shown in Fig.5.4.

   (a) The comm. module only interprets the first three fields in an RMI message. The rest are left for RMI software.

   (b) The comm. modules collectively implement a specified invocation semantics such as at-most-once.

   (c) The comm. module in a server selects the dispatcher for the class of object to be invoked.

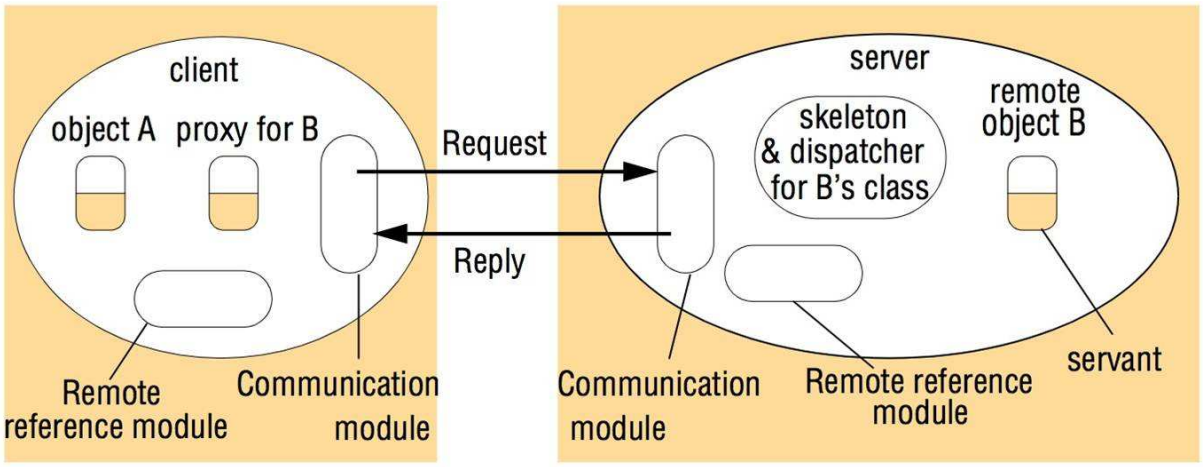
c. **Remote reference module** (RRM). Responsible for: a) translating between local and remote object references; b) creating remote object references.

   (b) The RRM in each process has a *remote object table* that records the correspondence between local object references in that process and remote object references (system wide). That is, the table maps between local object references to remote object references. The table contains:

      i. An entry for all the remote objects held by the process. That is one entry for each remote object managed by the process.

      ii. An entry for each local proxy.

   (b) This module is called components of RMI software when they marshall or unmarshall remote object references. When a request message arrives, the



这是什么?

Figure 40: The role of proxy and skeleton in remote method invocation (Fig.5.15, p.210)

table is consulted to find out which local object is to be invoked. When a request message is to be sent out, the table will record the remote object reference and its local object reference.

(c) The actions of the RRM:

i. When a remote object is to be passed as argument or result for the first time, the RRM is asked to create a remote object reference that will be added to the table.

ii. 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 either to a proxy or to a remote object. If 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.

d. **The RMI software**. A layer of software between the application objects and the communication and remote reference modules. It plays a role similar to stub procedures in RPC. It consists a *proxy*, a *dispatcher*, and a *skeleton*.

(a) *Proxy*. Proxies are *client stubs* (in terms of RPC terminologies) and hence are only in client processes. There is one proxy for each remote object for which a process holds a remote object reference.
A proxy for a remote object acts as a client's local representative or proxy (hence the name) for the remote object. The caller invokes a method on the local proxy which is responsible for carrying out the method call on the remote object. In RMI, a proxy for a remote object implements the same set of remote interfaces that a remote object implements.
When a proxy's method is invoked, it does the following:

· Initiates a connection with the remote JVM containing the remote object;

- Marshals (converts, writes and transmits) the parameters to the remote JVM;
- Waits for the result of the method invocation;
- Unmarshall (reads and converts) the return value or exception returned, and returns the value to the caller.

(b) *Dispatcher*. A dispatcher and a skeleton collectively function as a *server stub*, hence they only reside on a server process for a remote object. A server has one dispatcher and one skeleton for each class representing a remote object. A dispatcher receives *request* messages from the communication module. For each request message, it uses *methodId* to select appropriate method in the skeleton, passing on the *request* message to the skeleton.

(c) *Skeleton*. A skeleton *implements* the methods in the remote interface. However, this implementation is not actual implementation of remote methods themselves. Rather, a skeleton implements how remote methods in the remote interface will be invoked on the server.

When a skeleton receives an incoming method invocation it does the following:

- Unmarshall (reads and converts) the parameters for the remote method;
- Invokes the method on the actual remote object implementation; and
- Marshals (converts, writes and transmits) the result (return value or exception) to the caller.

Comment: Java 1.2 no longer requires a skeleton class. Only a stub class is required and generated by rmic.

e. **Generation of the classes for proxies, dispatchers, and skeletons**. The classes for the RMI software are generated automatically by an interface compiler. In Java RMI, the set of methods available in a remote object is defined as a Java interface that is implemented within the class of the remote object. The Java RMI compiler (rmic.exe in Windows platform) generates the code for the proxy, dispatcher, and skeleton classes from the class of the remote object.

f. **Server and client programs**.

(a) The server program contains the classes for the dispatcher and skeleton, together with the codes for classes (called *servant classes*) of all the remote objects it supports.

(b) Because remote interfaces cannot include constructors, remote objects are created in either an *initialization section*, or in methods in a remote interface designed for that purpose. Such methods are called *factory methods*. Any remote object that intends to be able to create new remote objects on demand for clients must provide methods in its remote interface for this purpose.

(c) An *initialization section* is responsible for creating and initializing at least one of the remote objects to be hosted by the server. Additional remote objects

may be created in response to requests from clients. The initialization section may also register some of its remote objects with a *binder*. Normally it will register just one of the remote objects, which can be used to access the rest.

为啥连接一个就可以包括剩余的了

(d) The client program consists of classes for proxies for all of the remote objects that it will invoke. A binder can be used to look up remote object references.

g. **The binder**. Before invoking any remote method, a client has to obtain a reference to a remote object that hosts the remote method.

(a) A *binder* in a DS is a separate service that maintains a table containing mappings from textual names of remote objects to remote object references.

(b) A server can use a binder to register its remote objects by name. A client can use a binder to obtain remote object references.

(c) Java RMI supports a binder RMIregistry. DCE RPC supports DCE name services. CORBA has its own Naming Service.

h. **Object location**.

(a) In a distributed object system, if remote objects are static, i.e. they stay under management of the same process for the rest of their life after their creation, the remote object references as defined in Section 4.4 of the 1st textbook are adequate for clients to find remote object servers.

(b) If remote objects can migrate from process to process and from computer to computer, then the previously presented representation of remote object references is not sufficient for clients to locate remote object servers.

· In this case a so called *location service* is needed. Such a service will map remote object references to their probable current locations.

· A client will then have to present both a remote object reference and an a current location of the remote object for each RMI.

i. **Server threads**.

(a) An RMI executes a remote object in a server. During the execution, other local and remote objects may be invoked. Hence an RMI may take an arbitrary amount of time to return. In a single thread server, the server will be blocked before the return.

(b) A server can allocate a separate thread to executing each remote invocation. This demands the designers of a remote object to deal with potential problems with concurrent threads. (cf. the *Thread* class in Section 2 about Java TCP API of this lecture).

j. **Activation of remote objects**.

(a) Most servers that provide RMI services are not running as physical processes until explicitly invoked by a remote invocation. This is similar to an FTP

or TELNET server, which is not actually *forked* as a separate server process (by the so called *Internet super daemon* process *inetd*) until a client request an FTP or TELNET session.

(b) *Active* and *passive* remote objects.

i. A remote object is termed as *active* when it is available for invocation within a running process. It is called *passive* if it is not currently active but can be made active.

ii. A passive object contains two ports: (i) the implementation of the methods; and (ii) its state in the marshalled form.

(c) *Activators*. A process that starts server processes (more precisely it activates passive processes) to host remote objects are called a activator. An activator is responsible for:

· Registering passive objects that are available for activation, which involves recording the names of servers against the URLs or file names of the corresponding passive objects.

· Starting named server processes and activating remote objects in them.

· Keeping track of the locations of the servers for remote objects that it has already activated.

In Java RMI each host must run an activator as a system daemon process. The activator is responsible for activating objects on that host.

k. **Persistent object stores**.

(a) *Persistent objects*. An object that lives between activations is called persistent object (like *static* variables can survive between procedure calls).

(b) Persistent objects are generally managed by persistent object stores in a marshalled form on disk. They will be activated when methods in them are invoked by other object.

(c) The *Activatable* class Java and *Entity beans* in JavaBeans are examples of persistent objects.