Script generated by TTT

Title: Distributed_Applications (14.05.2013)

Date: Tue May 14 14:30:33 CEST 2013

Duration: 89:42 min

Pages: 31

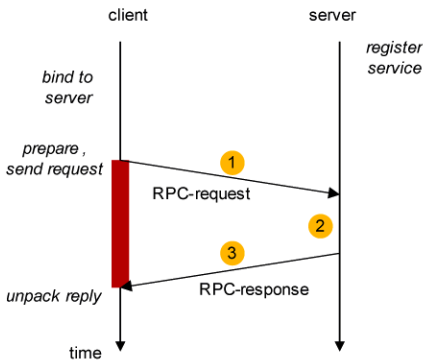
Remote Invocation (RPC/RMI)

Issues
Introduction
Distributed applications based on RPC
Remote Method Invocation (RMI)
Servlets

RPC properties

Neither the client nor the server assume that the procedure call is performed over a network.

Control flow for RPC calls



Differences between RPC and local procedure call
Basic RPC characteristics
RPC and OSI
RPC vs message exchange

RPC and OSI

Integration of the RPC into ISO/OSI protocol stack

layer 7 application layer	client-server model	
layer 6 presentation layer	RPC	hides communication details
layer 5 session layer	message exchange , e.g. request -response protocol	Operating system interface to underlying communication protocols
layer 4 transport layer	transport protocols e.g. TCP/UDP or OSI TP4	transfer of data packets

transport protocols: UDP (User Datagram Protocol) transports data packets without guarantees; TCP (Transmission Control Protocol) verifies correct delivery of data streams.

message exchange: socket interface to the underlying communication protocols.

RPC: hides communication details behind a procedure call and helps bridge heterogeneous platforms.

RPC vs message exchange

RPC	message exchange
synchronous (generally)	asynchronous
1 primitive operation (RPC call)	2 primitive operation (send, receive)
messages are configured by RPC system	message specification by programmer
one open RPC	several parallel messages possible

The RPC protocol defines only the structure of the request/answer messages; it does not supply a mechanism for secure data transfer.

RPC exchange protocols

RPC exchange protocols

There are different types of RPC exchange protocols

- the request (R) protocol
- the request-reply (RR) protocol
- the request-reply-acknowledge (RRA) protocol.

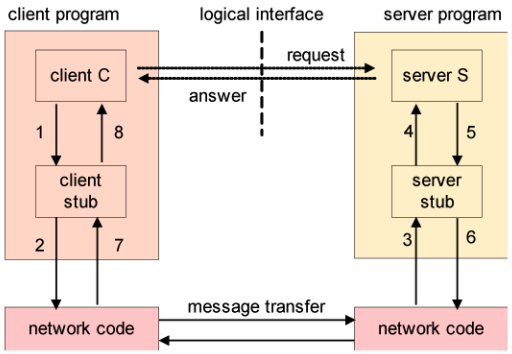
Remote Invocation (RPC/RMI)

Issues
Introduction
Distributed applications based on RPC
Remote Method Invocation (RMI)
Servlets

Stubs

Integration of software handling the communication between components of a distributed application.

- Stubs encapsulate the distribution specific aspects.
- Stubs represent interfaces.
- ***Client Stub*** : contains the proxy definition of the remote procedure P.
- ***Server Stub*** : contains the proxy call for the procedure P.



Stub functionality

Client and server stubs have the following tasks during client - server interaction.

1. **Client stub**

   specification of the remote service operation; assigning the call to the correct server; representation of the parameters in the transmission format.

   decoding the results and propagating them to the client application.

   unblocking of the client application.

2. **Server stub**

   decoding the parameter values; determining the address of the service operation (e.g. a table lookup).

   invoking the service operation.

   prepare the result values in the transmission format and propagate them to the client.

Distributed applications based on RPC

How to implement distributed applications based on remote procedure calls?

Distributed application

In order to isolate the communication idiosyncrasy of RPCs and to make the network interfaces transparent to the application programmer, so-called ***stubs*** are introduced.

Stubs

Stub functionality

Implementing a distributed application

RPC language

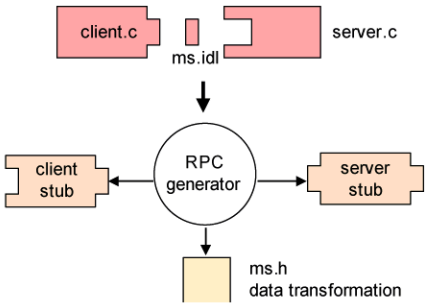
Phases of RPC based distributed applications

RPC generator

An RPC generator

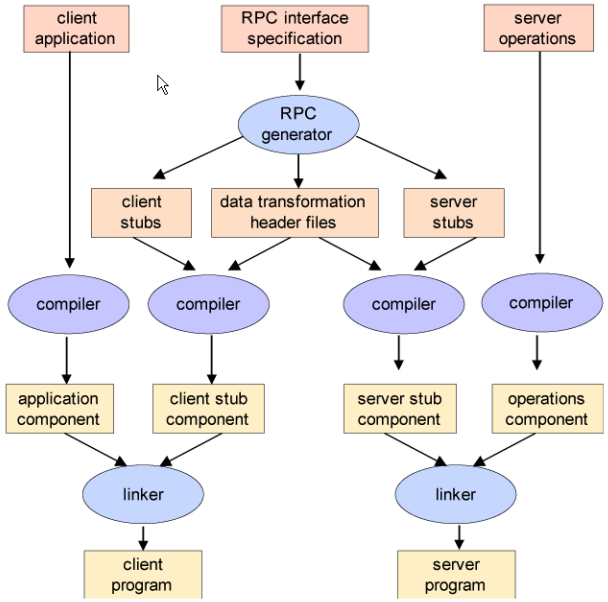
reduces the time necessary for implementation and management of the components of a distributed application.

a declarative interface description is easier to modify and therefore less error-prone.



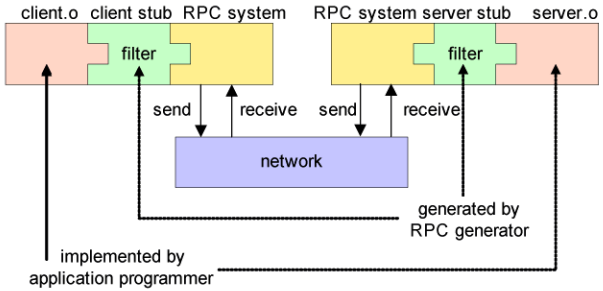
Applying the RPC generator

The individual steps for generating a distributed application are illustrated in the following figure.



Structure of a distributed application

The internal structure of a distributed application created using an RPC generator is as follows:



Implementing a distributed application

Manual implementation of stubs is error-prone ⇒ use of a ***RPC generator*** to generate stubs from a declarative specification.

RPC generator

Applying the RPC generator

Structure of a distributed application

Distributed applications based on RPC

How to implement distributed applications based on remote procedure calls?

Distributed application

In order to isolate the communication idiosyncrasy of RPCs and to make the network interfaces transparent to the application programmer, so-called ***stubs*** are introduced.

Stubs

Stub functionality

Implementing a distributed application

RPC language

Phases of RPC based distributed applications

Semistatic binding

The client determines the server address during the initialization of the client process. Server address remains unchanged for the whole life span of the client process.

Binding can take place via

entry in a database.

broadcast or multicast message.

name service.

mediation mechanism ("broker" or "trader"); a broker mediates between client and server.

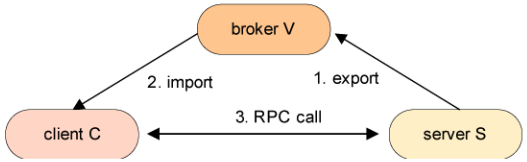
Possible terms for a mediation component are: registry, broker or trader; Corba uses the term object request broker.

Functionality of a broker

servers register their available service interfaces with the broker ("export interface").

the broker supplies the client with information in order to localize a suitable server and to determine the correct service interface ("import interface").

Client-to-server binding



Broker information

Handling client requests

Broker may either just provide the service interface to the client or act as a mediator between client and server.

direct communication between C and S.

indirect communication between C and S; communication between C and S is only possible via broker V (or several brokers).

A broker manages information about the available, exported interfaces.

server names ("white pages")

service types ("yellow pages")

behavioral or functional attributes

static attributes: functionality of the provided services, cost, required bandwidth.

dynamic attributes: current server state.

Mediation and brokering

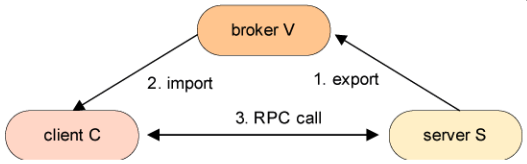
Possible terms for a mediation component are: registry, broker or trader; Corba uses the term object request broker.

Functionality of a broker

servers register their available service interfaces with the broker ("export interface").

the broker supplies the client with information in order to localize a suitable server and to determine the correct service interface ("import interface").

Client-to-server binding



Broker information

Handling client requests

Broker may either just provide the service interface to the client or act as a mediator between client and server.

direct communication between C and S.

indirect communication between C and S; communication between C and S is only possible via broker V (or several brokers).

Generated by Targeteam

We distinguish between 3 phases:

a) design and implementation

b) binding of components

c) invocation: a client invoking a server operation.

Component binding

Mediation and brokering

Definitions

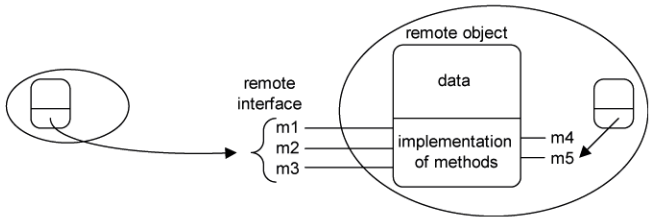
Definition: Remote object is an object whose method can be called by an object residing on another Java Virtual Machine (JVM), even on another computer.

Definition: Remote interface

is a Java interface specifying the methods of a remote object.

Definition: Remote method invocation (RMI) allows object-to-object communication between different Java Virtual Machines (JVM), i.e. it is the action of invoking a method of a remote interface on a remote object.

The method calls for local and remote objects have the same syntax.



RMI characteristics

Note in RMI: client and server are objects.

RMI supports location and access transparency.

Localization of remote objects.

Communication with remote objects (using method calls).

Automated class loading for objects passed as parameters or results.

Clients interact with remote interfaces, rather than with classes implementing these interfaces.

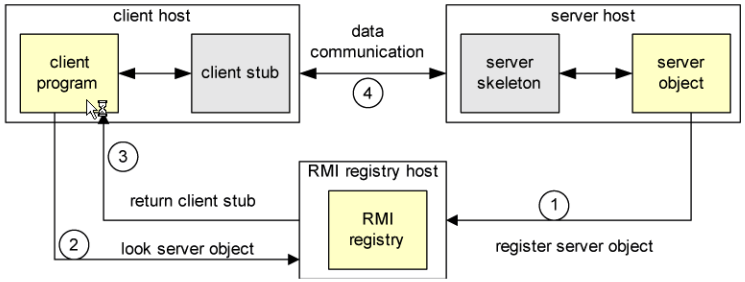
How does RMI work

How does RMI work

Java RMI uses

a registry to provide naming services for remote objects,

stub and skeleton to facilitate communications between client and server.

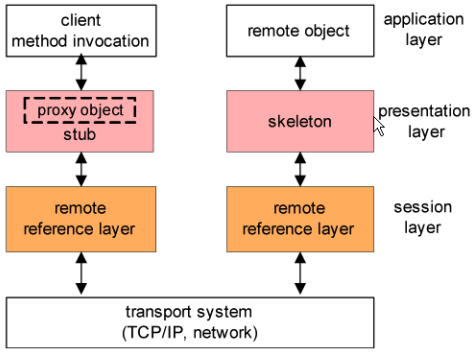


RMI works as follows

1. a server object is registered with the RMI registry
2. a client looks through the RMI registry for the remote object
3. once the remote object is located, its stub is returned to the client
4. the remote object can be used in the same way as a local object

communication between client and server is handled by stubs and skeletons.

RMI architecture



Stub/Skeleton layer

Layer intercepts method calls by the client and redirects these calls to the remote object.

Object serialization/deserialization; hidden from the application.

Remote Reference layer

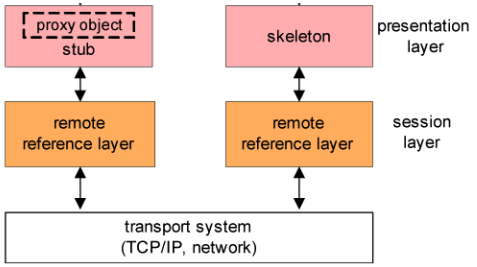
Connects client and remote objects exported by the server environment by a 1-to-1 connection link.

The layer provides JRMP (Java Remote Method Protocol) via TCP/IP.

Mapping of stub/skeleton operations to the transport protocol of the host; it interfaces the application code with the network communication.

The layer supports the method `invoke`.

RMI architecture



stub realises remote server object interface

Stub/Skeleton layer

Layer intercepts method calls by the client and redirects these calls to the remote object.

Object serialization/deserialization; hidden from the application.

Remote Reference layer

Connects client and remote objects exported by the server environment by a 1-to-1 connection link.

The layer provides JRMP (Java Remote Method Protocol) via TCP/IP.

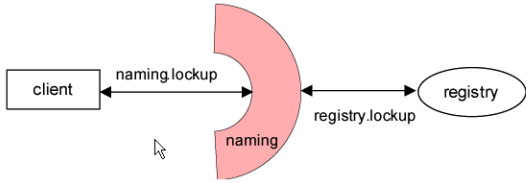
Mapping of stub/skeleton operations to the transport protocol of the host; it interfaces the application code with the network communication.

The layer supports the method `invoke`.

```
Object invoke (Remote obj, java.lang.reflect.Method method, Object
[ ] params, long opnum) throws Exception
```


Locating remote objects

How does the client find the remote object?



RMI supports a special name service, the ***RMI registry***

mapping of names to remote objects.

stand-alone Java application.

the RMI registry runs on all those machines hosting remote objects.

standard port for registry requests is 1099.

the RMI registry is itself a remote object.

access of the RMI registry via the java.rmi.Naming class.

Naming interface methods

Registry-Lookup

Naming interface methods

public static void `bind (String name, Remote obj)`

Throws AlreadyBoundException, java.net.MalformedURLException, RemoteException.

associates the remote object obj with name (in URL format).

example for name: rmi: //host[:service-port]/service-name

if name is already bound to an object, then AlreadyBoundException is triggered.

public static void `rebind (String name, Remote obj)`

Throws java.net.MalformatURLException, RemoteException.

associates always the remote object obj with name (in URL format).

public static `Remote lookup (String name)`

Throws NotBoundException, java.out.MalformedURLException, RemoteException.

returns as a result a reference (a stub) to the remote object.

if name is not bound to an object, then NotBoundException is triggered.

public static void `unbind (String name)`

Throws NotBoundException, RemoteException.

public static `String [ ] list (string name)`

Throws java.net.MalformedURLException, RemoteException.

Naming interface methods

example for name: rmi: //host[:service-port]/service-name

if name is already bound to an object, then AlreadyBoundException is triggered.

public static void `rebind (String name, Remote obj)`

Throws java.net.MalformatURLException, RemoteException.

associates always the remote object obj with name (in URL format).

public static `Remote lookup (String name)`

Throws NotBoundException, java.out.MalformedURLException, RemoteException.

returns as a result a reference (a stub) to the remote object.

if name is not bound to an object, then NotBoundException is triggered.

public static void `unbind (String name)`

Throws NotBoundException, RemoteException.

public static `String [ ] list (string name)`

Throws java.net.MalformedURLException, RemoteException.

as a result, it returns all names entered in the registry.

the name parameter specifies only the host and port information.

associates the remote object obj with name (in URL format).

example for name: rmi: //host[:service-port]/service-name

if name is already bound to an object, then AlreadyBoundException is triggered.

public static void `rebind (String name, Remote obj)`

Throws java.net.MalformatURLException, RemoteException.

associates always the remote object obj with name (in URL format).

public static `Remote lookup (String name)`

Throws NotBoundException, java.out.MalformedURLException, RemoteException.

returns as a result a reference (a stub) to the remote object.

if name is not bound to an object, then NotBoundException is triggered.

public static void `unbind (String name)`

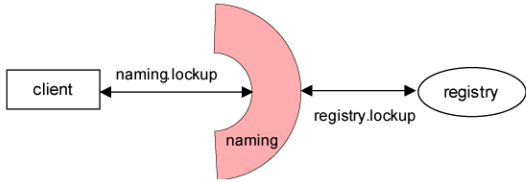
Throws NotBoundException, RemoteException.

public static `String [ ] list (string name)`

Throws java.net.MalformedURLException, RemoteException.

as a result, it returns all names entered in the registry.

How does the client find the remote object?



RMI supports a special name service, the ***RMI registry***

mapping of names to remote objects.

stand-alone Java application.

the RMI registry runs on all those machines hosting remote objects.

standard port for registry requests is 1099.

the RMI registry is itself a remote object.

access of the RMI registry via the java.rmi.Naming class.

Naming interface methods
Registry-Lookup

The client invokes a lookup for a particular URL, the name of the service (rmi://host:port/service). The following describes the steps:

1) a socket connection is opened with the host on the specified port.

2) a stub to the remote registry is returned.

3) the method Registry.lookup() on this stub is performed. The method returns a stub for the remote object.

4) the client interacts with the remote object through its stub.