Message Passing vs. Distributed Objects

Distributed Objects

M. L. Liu

Message Passing versus Distributed Objects

- The message-passing paradigm is a natural model for distributed computing, in the sense that it mimics **interhuman communications**. It is an appropriate paradigm for network services where processes interact with each other through the exchanges of messages.
- However, the abstraction provided by this paradigm does not meet the needs of the complexity of sophisticated network applications.

Message Passing versus Distributed Objects –2

- **Message passing requires the participating processes to be tightly-coupled: throughout their interaction, the processes must be in direct communication with each other. If communication is lost between the processes (due to failures in the communication link, in the systems, or in one of the processes), the collaboration fails.**
- **The message-passing paradigm is data-oriented. Each message contains data marshalled in a mutually agreed upon format, and is interpreted as a request or response according to the protocol. The receiving of each message triggers an action in the receiving process.** It is inadequate for complex applications involving a large mix of requests and responses. In such an application, the task of interpreting the messages can become overwhelming.

The distributed object paradigm

- The distributed object paradigm is a paradigm that provides abstractions beyond those of the message-passing model. As its name implies, the paradigm is based on objects that exist in a distributed system.
- In object-oriented programming, supported by an object-oriented programming language such as Java, objects are used to represent an entity significant to an application. Each object encapsulates:
  - the **state** or data of the entity: in Java, such data is contained in the instance variables of each object;
  - the **operations** of the entity, through which the state of the entity can be accessed or updated.

object-oriented programming

To illustrate, consider objects of the *DatagramMessage* class. Each object instantiated from this class contains three state data items: a message, the sender’s address, and the sender’s port number. In addition, each object contains three operations:

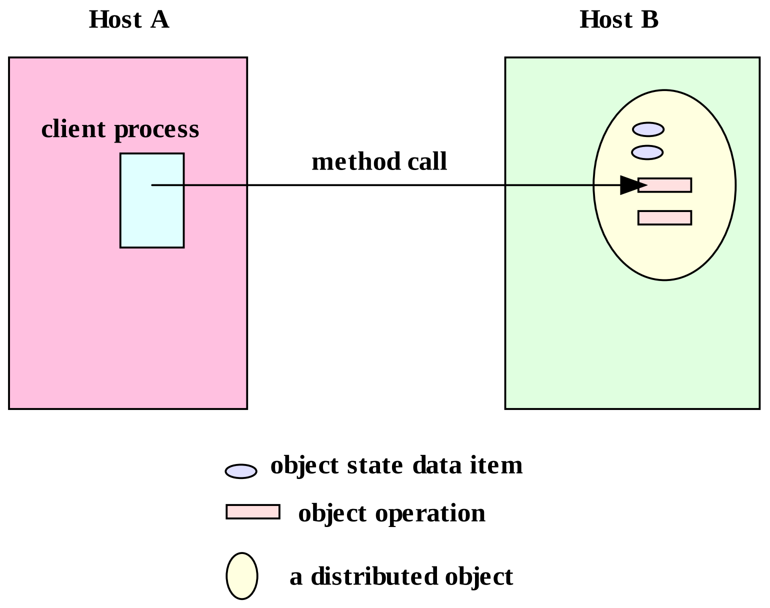
- a method *putVal*, which allows the values of these data items to be modified,
- a *getMessage* method, which allows the current value of the message to be retrieved, and
- a *getAddress* method, which allows the sender’s address to be retrieved.

Local Objects vs. Distributed Objects

- Local objects are those whose methods can only be invoked by a **local process**, a process that runs on the same computer on which the object exists.
- A distributed object is one whose methods can be invoked by a **remote process**, a process running on a computer connected via a network to the computer on which the object exists.

The Distributed Object Paradigm

In a distributed object paradigm, network resources are represented by distributed objects. To request service from a network resource, a process invokes one of its operations or methods, passing data as parameters to the method. The method is executed on the remote host, and the response is sent back to the requesting process as a return value.



Message Passing versus Distributed Objects - 3

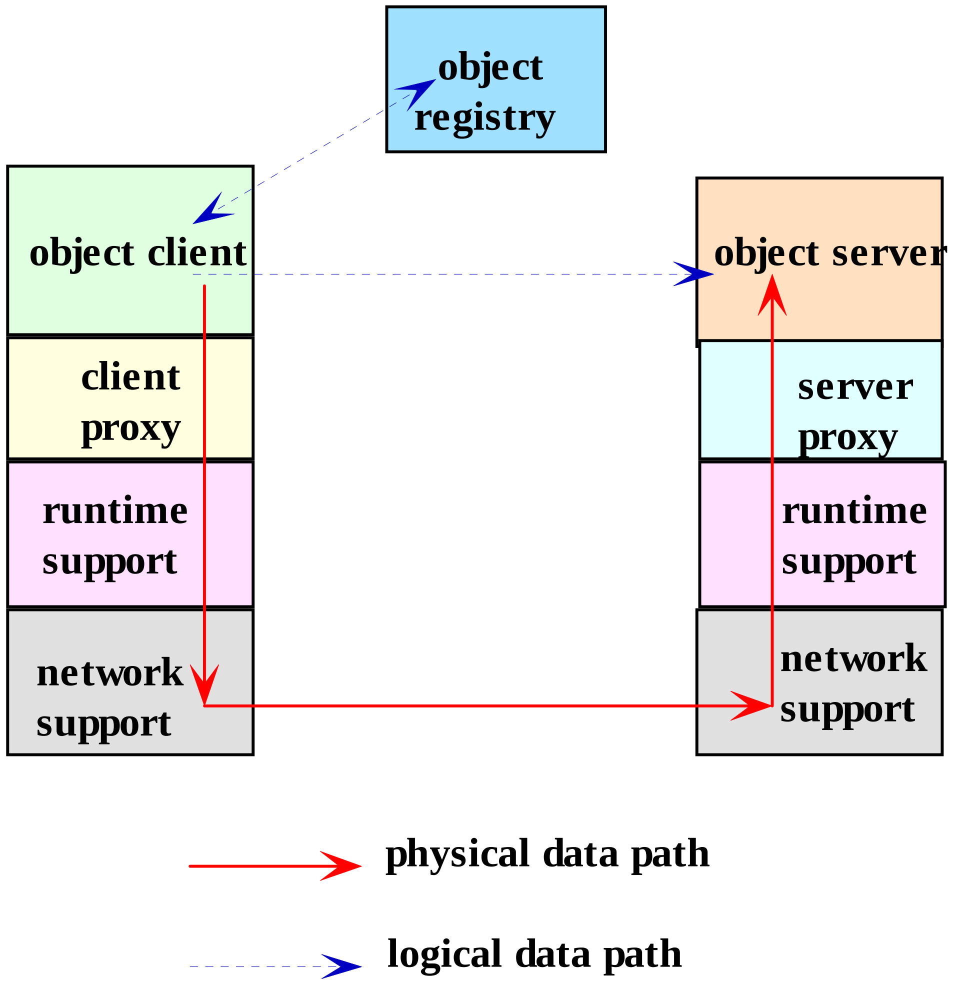
Compared to the message-passing paradigm, which is data-oriented, the distributed objects paradigm is **action-oriented**: the focus is on the invocation of the operations, while the data passed takes on a secondary role. Although less intuitive to human-beings, the distributed-object paradigm is more natural to object-oriented software development.

The Distributed Object Paradigm - 2

- A process running in host A makes a method call to a distributed object residing on host B, passing with the call data for the parameters, if any.
- The method call invokes an action performed by the method on host B, and a return value, if any, is passed from host B to host A.
- A process which makes use of a distributed object is said to be a client process of that object, and the methods of the object are called remote methods (as opposed to local methods, or methods belonging to a local object) to the client process.

The Distributed Objects Paradigm

An Archetypal Distributed Objects System



Distributed Object System

- A distributed object is provided, or exported, by a process, here called the object server. A facility, here called an object registry, must be present in the system architecture for the distributed object to be registered.
- To access a distributed object, a process –an **object client** – looks up the object registry for a **reference**[1] to the object. This reference is used by the object client to make calls to the methods.

[1] A reference is a "handle" for an object; it is a representation through which an object can be located in the computer where the object resides.

Distributed Object System - 2

- Logically, the object client makes a call directly to a remote method.
- In reality, the call is handled by a software component, called a *client proxy*, which interacts which the software on the client host that provides the runtime support for the distributed object system.
- The runtime support is responsible for the interprocess communication needed to transmit the call to the remote host, including the marshalling of the argument data that needs to be transmitted to the remote object.

Distributed Object System - 3

- A similar architecture is required on the server side, where the runtime support for the distributed object system handles the receiving of messages and the unmarshalling of data, and forwards the call to a software component called the server proxy.
- The server proxy interfaces with the distributed object to invoke the method call locally, passing in the unmarshalled data for the arguments.
- The method call results in the performance of some tasks on the server host. The outcome of the execution of the method, including the marshalled data for the return value, is forwarded by the server proxy to the client proxy, via the runtime support and network support on both sides.

Distributed Object Systems/Protocols

The distributed object paradigm has been widely adopted in distributed applications, for which a large number of mechanisms based on the paradigm are available. Among the most well known of such mechanisms are:

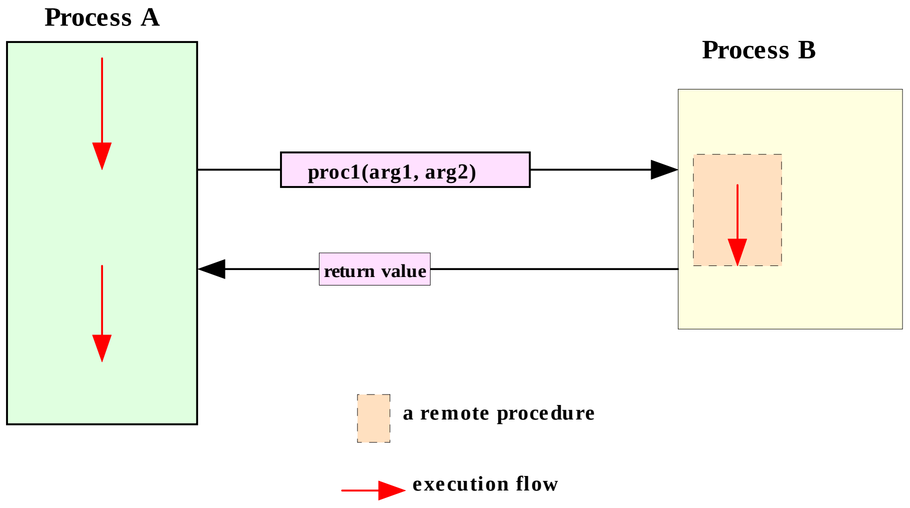
- ~ Java Remote Method Invocation (RMI),
- ~ the Common Object Request Broker Architecture (CORBA) systems,
- ~ the Distributed Component Object Model (DCOM),
- ~ mechanisms that support the Simple Object Access Protocol (SOAP).

Of these, the most straightforward is the Java RMI

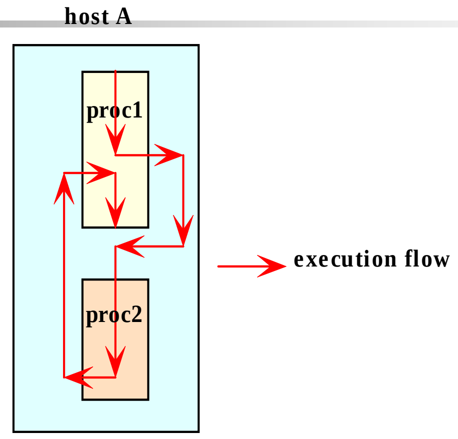
From Remote Procedure Call to Remote Method Invocation

Remote Procedure Calls (RPC)

- Remote Method Invocation has its origin in a paradigm called Remote Procedure Call
- In the remote procedure call model, a procedure call is made by one process to another, with data passed as arguments. Upon receiving a call, the actions encoded in the procedure are executed, the caller is notified of the completion of the call, and a return value, if any, is transmitted from the callee to the caller.



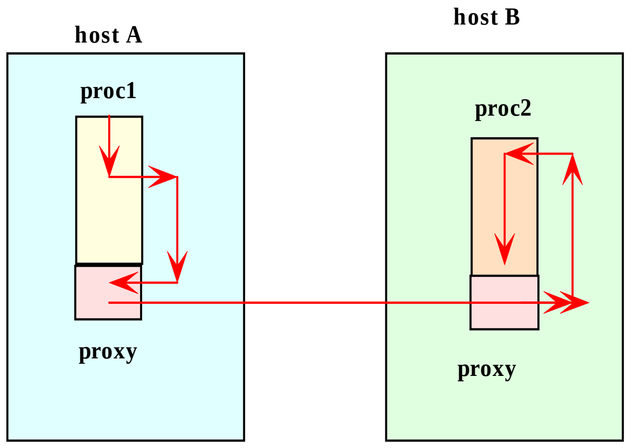
Local Procedure Call and Remote Procedure Call



A local procedure call

1. proc1 on host A makes a call to proc 2 on host B.
2. The runtime support maps the call to a call to the proxy on host A.
3. The proxy marshalls the data and makes an IPC call to a proxy on host B.

7. The proxy received the return value, unmarshalls the data, and forwards the return value to proc1, which resumes its execution flow.



4. The proxy on host B unmarshalls the data received and issues a call to proc2.
5. The code in proc2 is executed and returns to the proxy on host B.
6. The proxy marshalls the return value and makes an IPC call to the proxy on host A.

A remote procedure call
(the return execution path is not shown)

Remote Procedure Calls (RPC) - 2

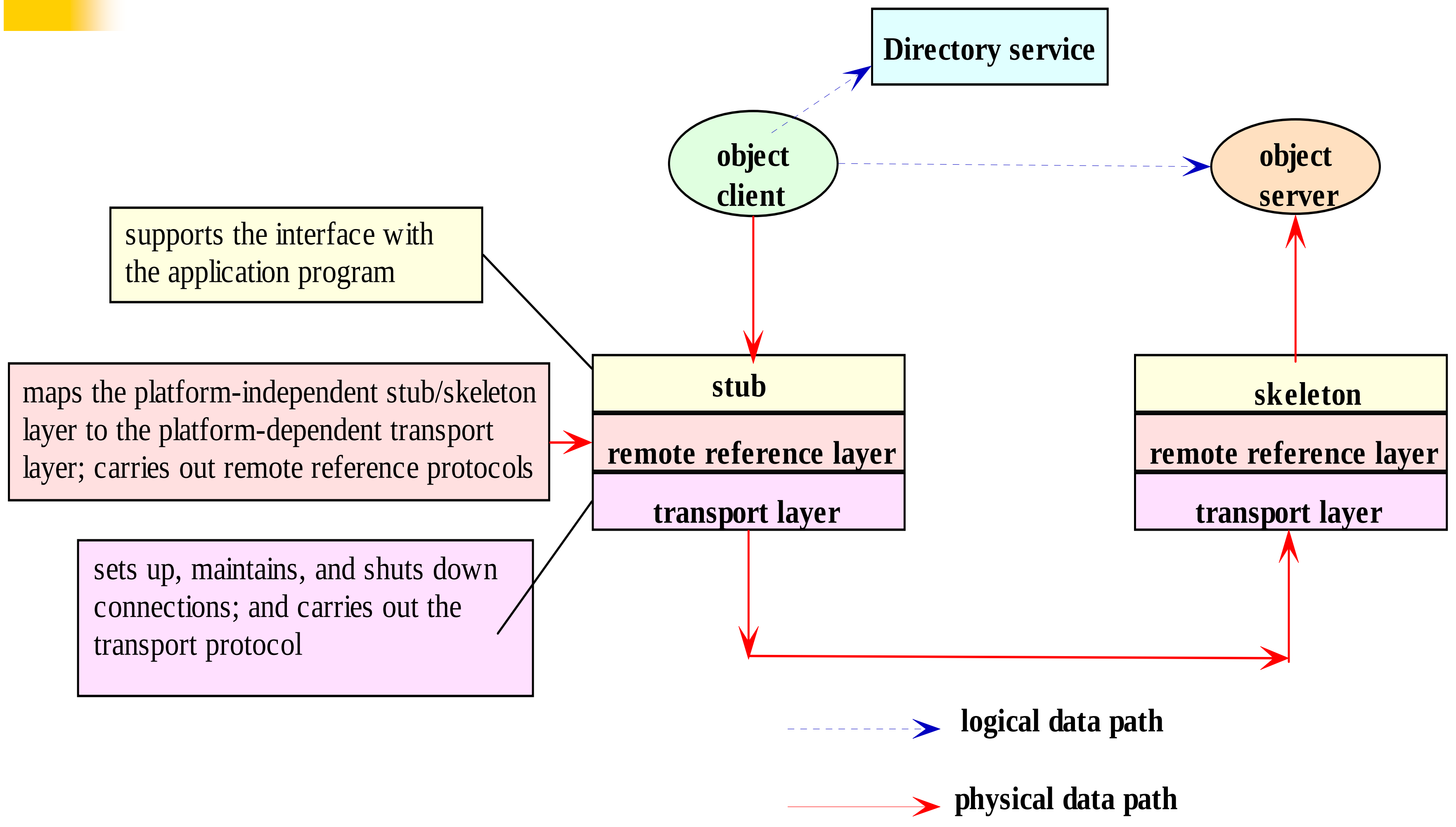
- Since its introduction in the early 1980s, the Remote Procedure Call model has been widely in use in network applications.
- There are two prevalent APIs for this paradigm.
  - the *Open Network Computing Remote Procedure Call*, evolved from the RPC API originated from Sun Microsystems in the early 1980s.
  - The other well-known API is the *Open Group Distributed Computing Environment* (DCE) RPC.
- Both APIs provide a tool, *rpcgen*, for transforming remote procedure calls to local procedure calls to the stub.

Java Remote Method Invocation

Remote Method Invocation

- Remote Method Invocation (RMI) is an object-oriented implementation of the Remote Procedure Call model. It is an API for Java programs only.
- Using RMI, an *object server* exports a *remote object* and registers it with a directory service. The object provides remote methods, which can be invoked in client programs.
- Syntactically:
  - A remote object is declared with a *remote interface*, an extension of the Java *interface*.
  - The remote interface is implemented by the object server.
  - An *object client* accesses the object by invoking the remote methods associated with the objects using syntax provided for remote method invocations.

The Java RMI Architecture


Directory service
object client
object server
supports the interface with the application program
maps the platform-independent stub/skeleton layer to the platform-dependent transport layer; carries out remote reference protocols
sets up, maintains, and shuts down connections; and carries out the transport protocol
stub
remote reference layer
transport layer
skeleton
remote reference layer
transport layer
logical data path
physical data path

Object Registry

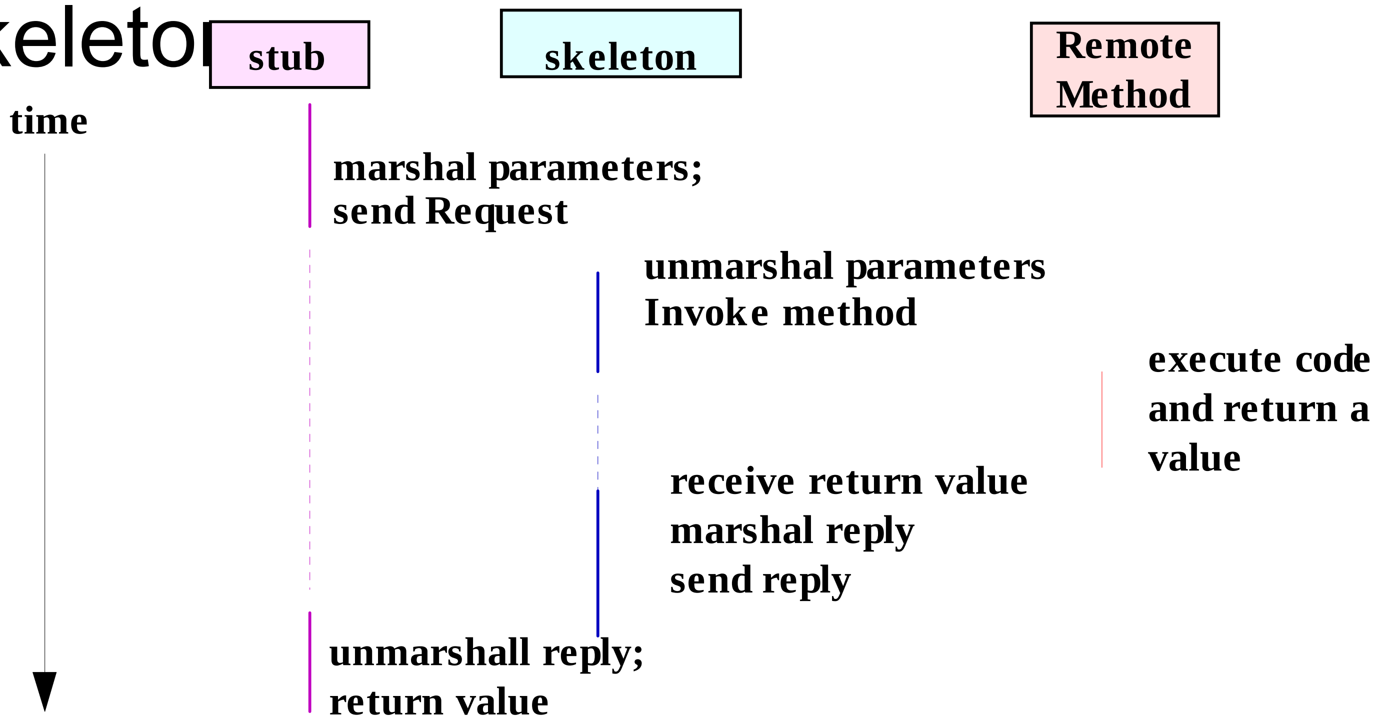
- The RMI API allows a number of directory services to be used[1] for registering a distributed object.
- We will use a simple directory service called the RMI registry, *rmiregistry*, which is provided with the Java Software Development Kit (SDK)[2]. The RMI Registry is a service whose server, when active, runs on the **object server's host machine**, by convention and by default on the TCP port 1099.

[1] One such service is the Java Naming and Directory Interface (JNDI), which is more general than the RMI registry, in the sense that it can be used by applications that do not use the RMI API.

[2] The Java SDK is what you download to your machine to obtain the use of the Java class libraries and tools such as the java compiler javac .

The interaction between the stub and the skeleton

A time-event diagram describing the interaction between the stub and the skeleton:


stub
skeleton
Remote Method
time
marshal parameters; send Request
unmarshal parameters Invoke method
execute code and return a value
receive return value marshal reply send reply
unmarshall reply; return value

(based on http://java.sun.com.marketing/collateral/javarim.html)

The API for the Java RMI

- The Remote Interface
- The Server-side Software
  - The Remote Interface Implementation
  - Stub and Skeleton Generations
  - The Object Server
- The Client-side Software

The Remote Interface

- A Java interface is a class that serves as a template for other classes: it contains declarations or signatures of methods[1] whose implementations are to be supplied by classes that implements the interface.
- A java remote interface is an interface that inherits from the Java *Remote* class, which allows the interface to be implemented using RMI syntax. Other than the Remote extension and the Remote exception that must be specified with each method signature, a remote interface has the same syntax as a regular or local Java interface.

[1] They are called abstract methods.

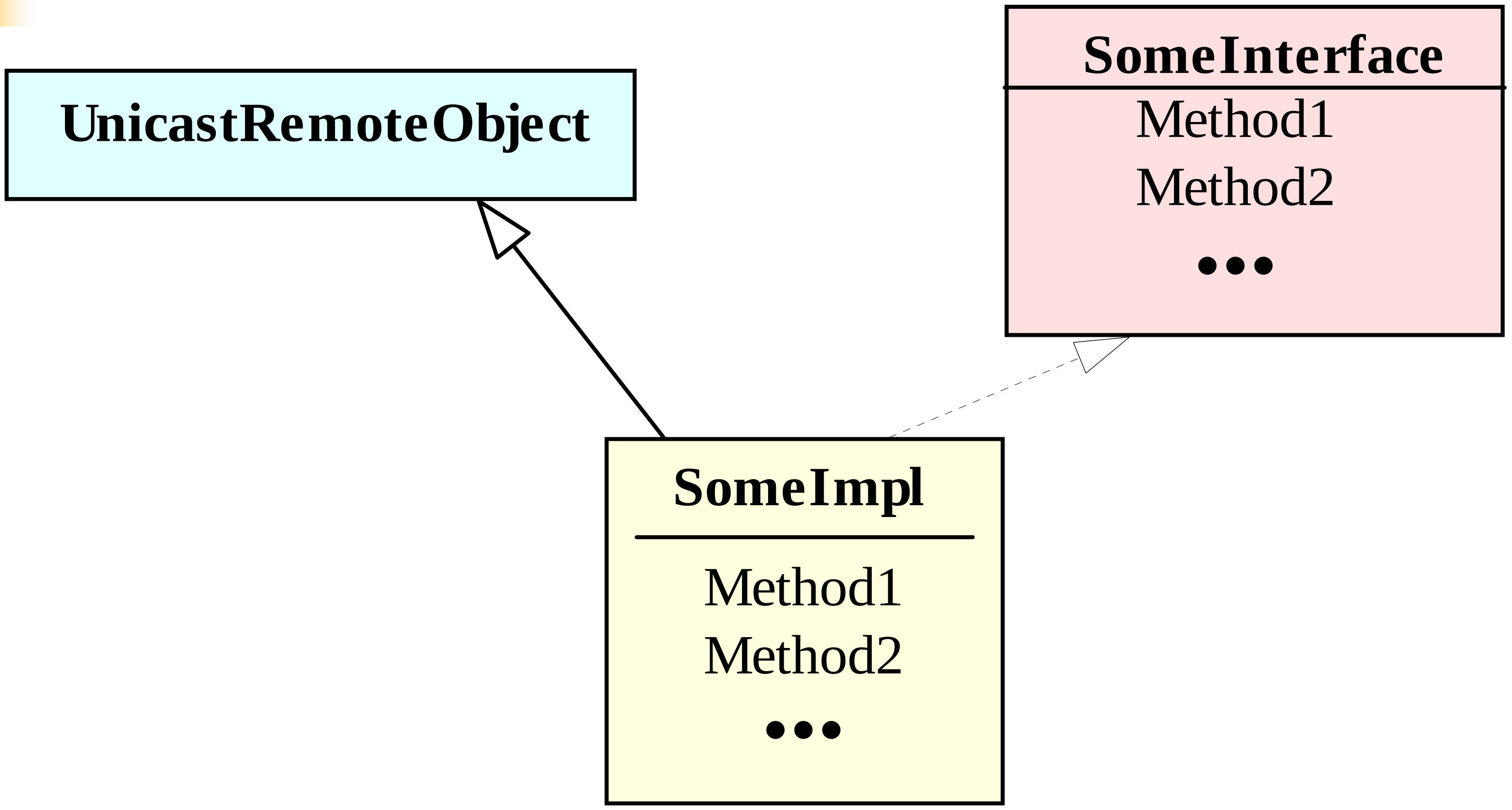
The Server-side Software

An object server is an object that provides the methods of and the interface to a distributed object. Each object server must

- implement each of the remote methods specified in the interface,
- register an object which contains the implementation with a directory service.

It is recommended that the two parts be provided as separate classes.

UML diagram for the SomeImpl class



UMLDiagram for SomeImpl

Stub and Skeleton Generations

In RMI, each distributed object requires a proxy each for the object server and the object client, knowns as the object's skeleton and stub respectively. These proxies are generated from the implementation of a remote interface using a tool provided with the Java SDK: the RMI compiler *rmic*.

```
rmic <class name of the remote interface
  implementation>
```

For example:

```
rmic SomeImpl
```

As a result of the compilation, two proxy files will be generated, each prefixed with the implementation class name:

SomeImpl_skel.class

SomeImpl_stub.class.

The stub file for the object

- The stub file for the object, as well as the remote interface file, must be shared with each object client – these file are required for the client program to compile.
- A copy of each file may be provided to the object client by hand. In addition, the Java RMI has a feature called “stub downloading” which allows a stub file to be obtained by a client dynamically.

The Object Server

- When an object server is executed, the exporting of the distributed object causes the server process to begin to listen and wait for clients to connect and request the service of the object.
- An RMI object server is a concurrent server: each request from an object client is serviced using a separate thread of the server. Note that if a client process invokes multiple remote method calls, these calls will be executed concurrently unless provisions are made in the client process to synchronize the calls.

The Client-side Software

The program for the client class is like any other Java class.

The syntax needed for RMI involves

- locating the RMI Registry in the server host,

  and
- looking up the remote reference for the server object; the reference can then be cast to the remote interface class and the remote methods invoked.

Summary - 1

- The distributed object paradigm is at a higher level of abstraction than the message-passing paradigm.
- Using the paradigm, a process invokes methods of a remote object, passing in data as arguments and receiving a return value with each call, using syntax similar to local method calls.
- In a distributed object system, an object server provides a distributed object whose methods can be invoked by an object client.

Summary - 2

- Each side requires a proxy which interacts with the system's runtime support to perform the necessary IPC.
- an object registry must be available which allow distributed objects to be registered and looked up.
- Among the best-known distributed object system protocols are the Java Remote Method Invocation (RMI), the Distributed Component Object, Model (DCOM), the Common Object Request Broker Architecture (CORBA) , and the Simple Object Access Protocol (SOAP).

Summary - 3

- Java RMI is representative of distributed object systems.
  - The architecture of the Java Remote Method Invocation API includes three abstract layers on both the client side and the server side.
  - The software for a RMI application includes a remote interface, server-side software, and client-side software.
  - What are the tradeoffs between the socket API and the Java RMI API?