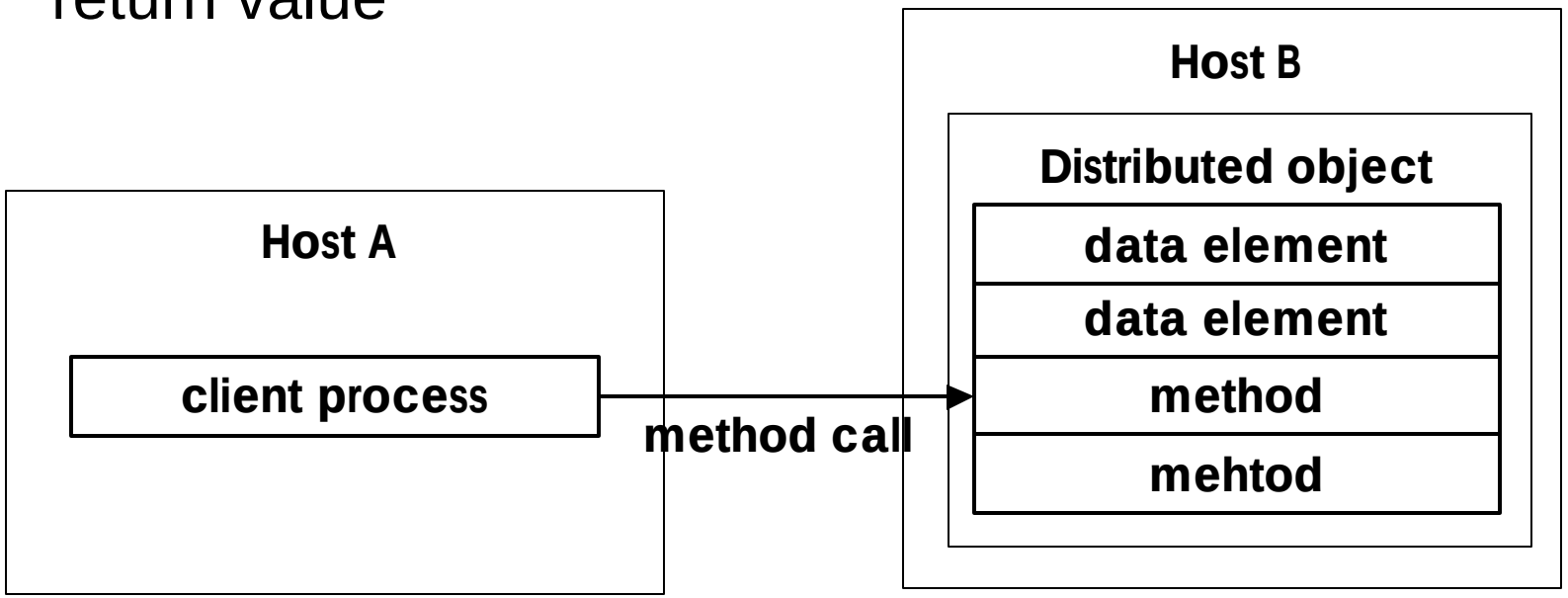
Distributed Objects and Java RMI basics

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 (remote) 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

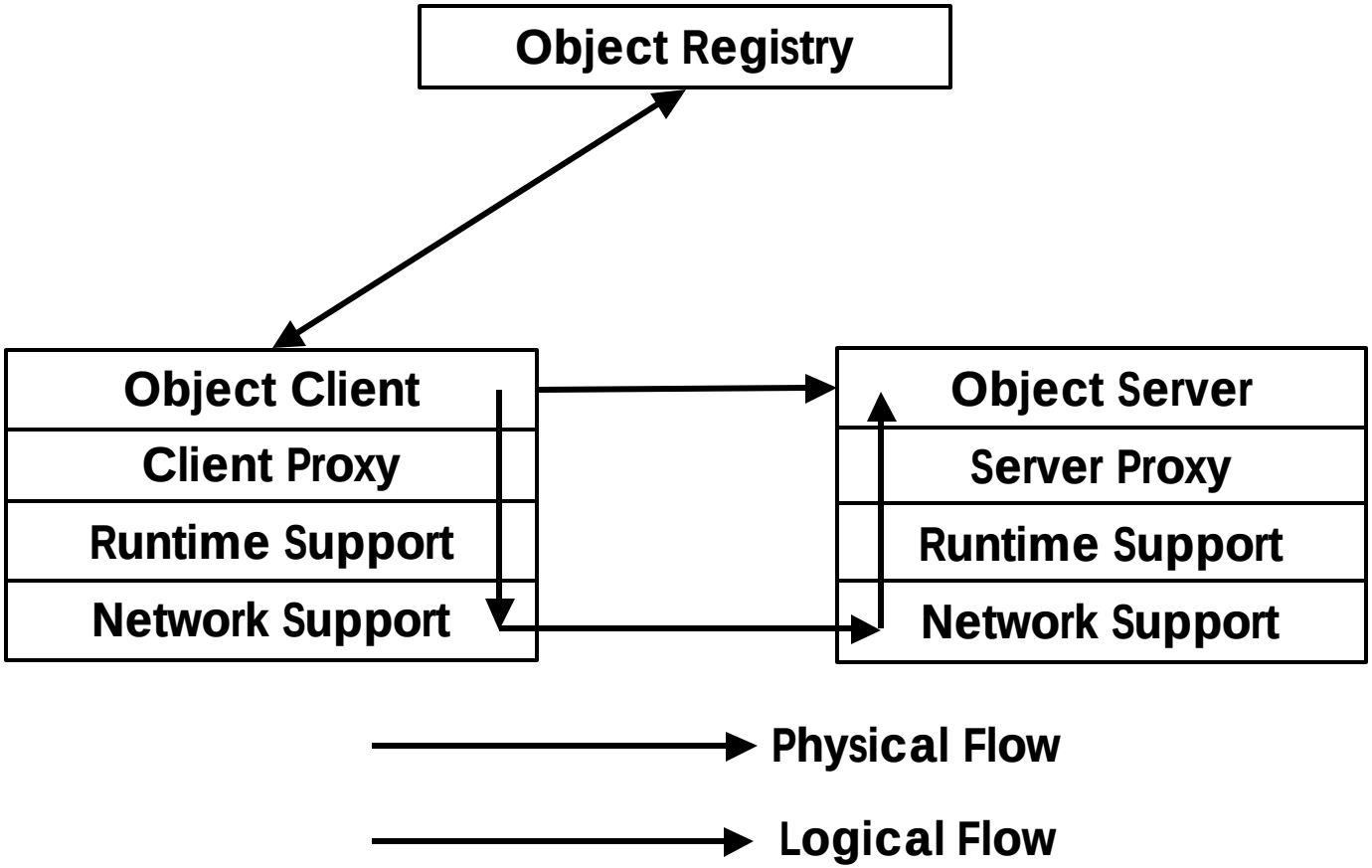
- Network resources are represented by distributed objects
- To request service from a network resource, a process invokes one of its operations or methods, passing data as parameters to the method
- The method is executed on the remote host, and the response is sent back to the requesting process as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



The Distributed Object Paradigm - 2

- A process running in host A makes a method call to a distributed object residing on host B, passing the parameters, if any
- The method call invokes an action performed by the method on host A, and a return value, if any, is passed from host A to host B
- 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)

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 distributed objects to be registered
- A reference is a “handle” for an object; it is a representation through which an object can be located in the machine where the object resides
- To access a distributed object, a process – an object client – looks up the object registry for a reference to the object. This reference is used by the object client to make calls to the methods

Distributed Object System - 2

- Logically, the object client makes a call directly to a remote method
- Actually, 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

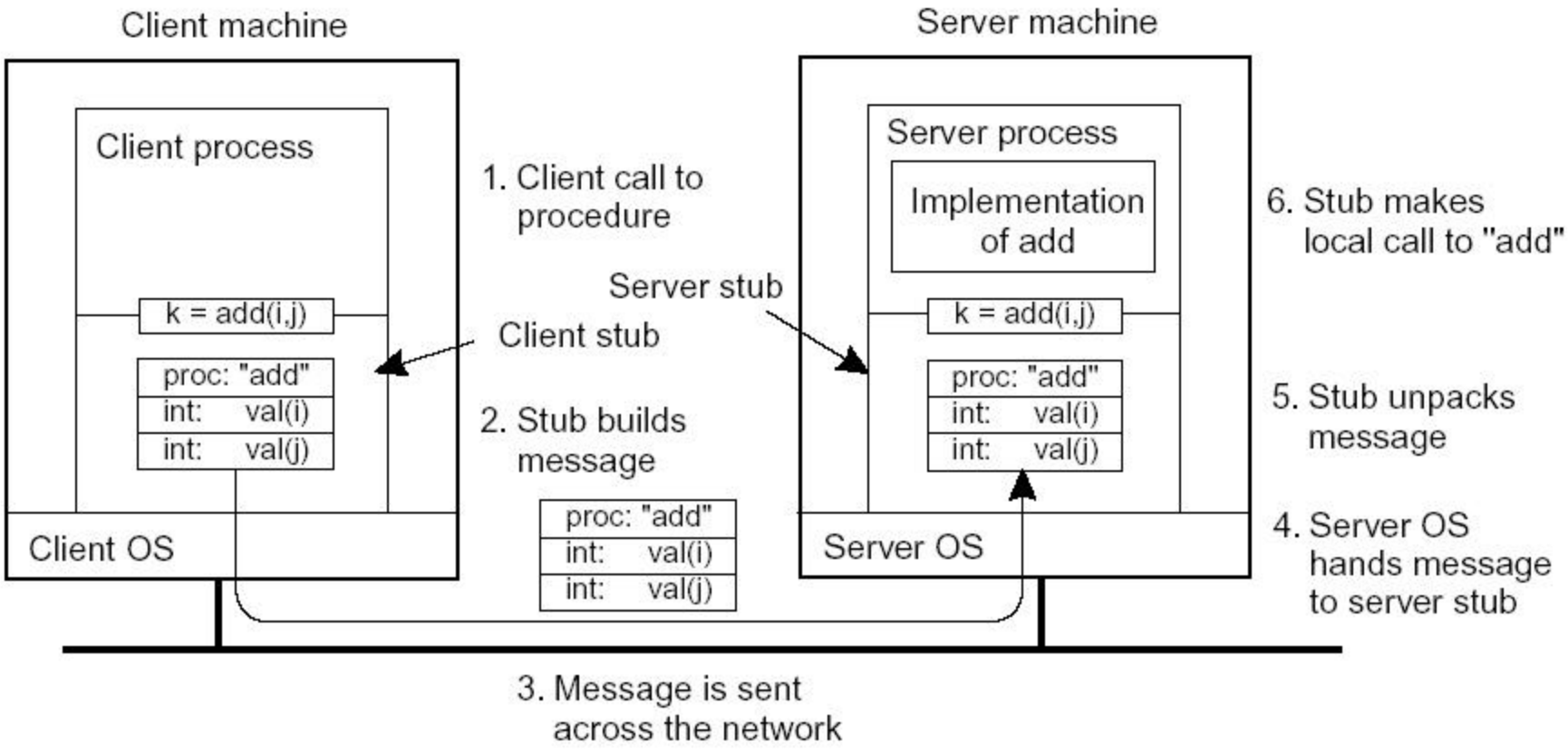
Distributed Object Mechanisms

- A large number of mechanisms based on the paradigm are available
  - Java Remote Method Invocation (RMI)
  - Common Object Request Broker Architecture (CORBA) systems
  - Distributed Component Object Model (DCOM)
  - Mechanisms that support the Simple Object Access Protocol (SOAP)

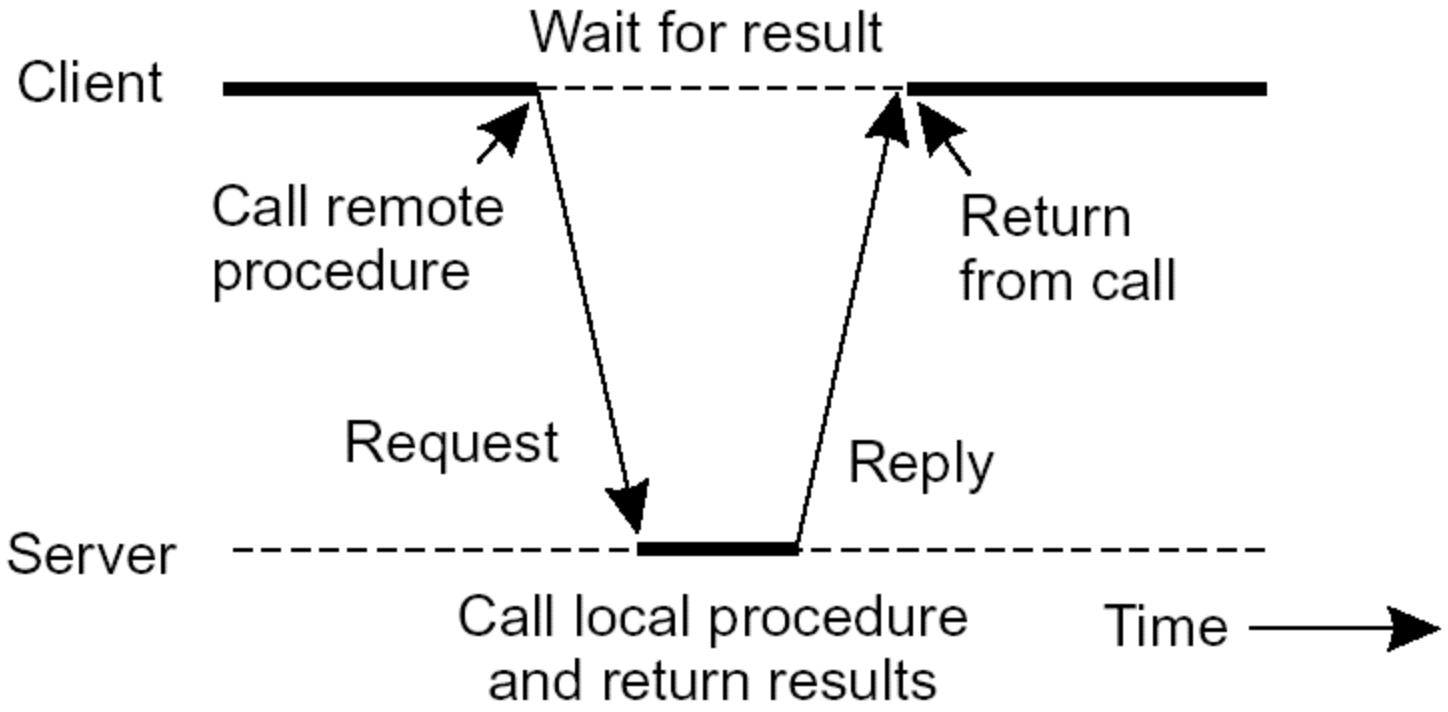
Remote Procedure Call (RPC)

- 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 called to the caller

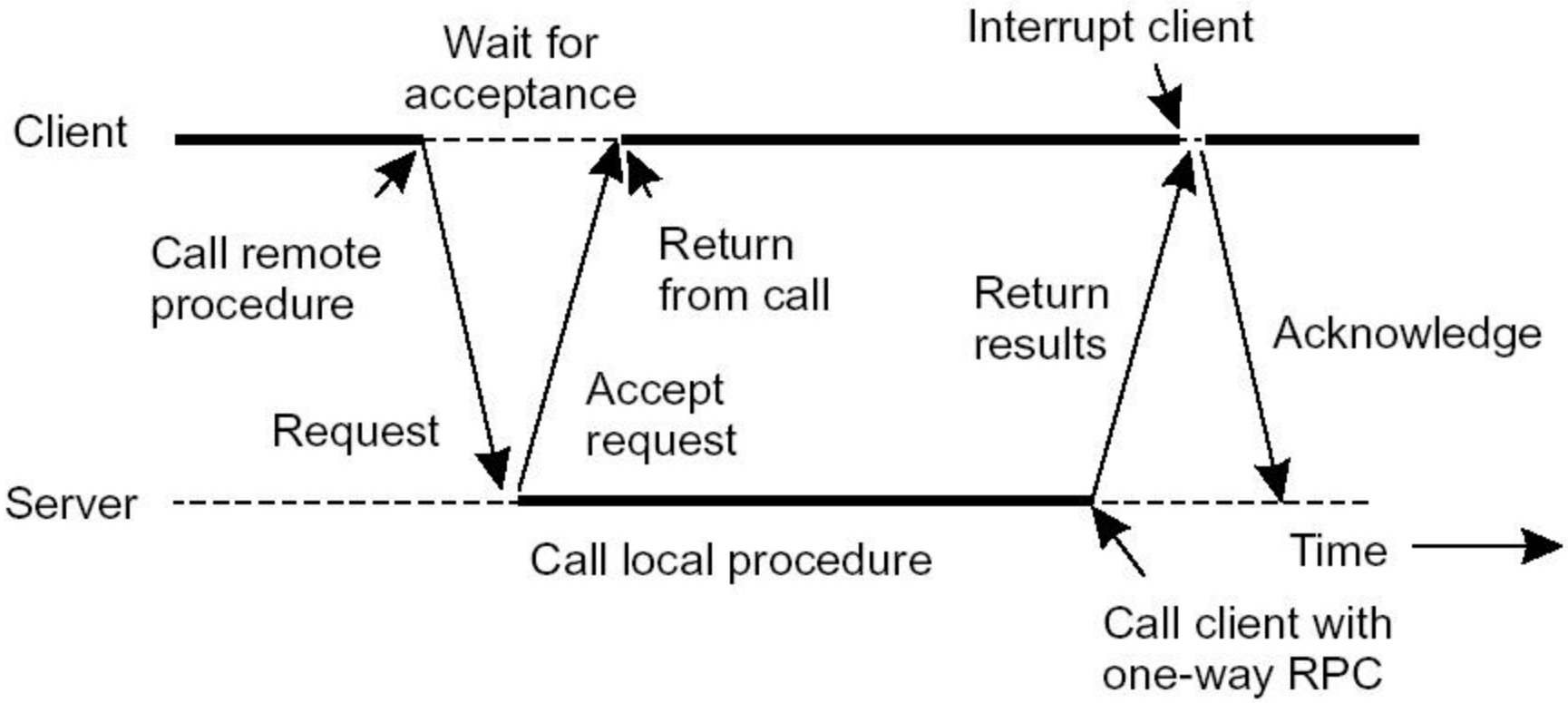
RPC: Remote Computation



RPC: Remote Computation - 2



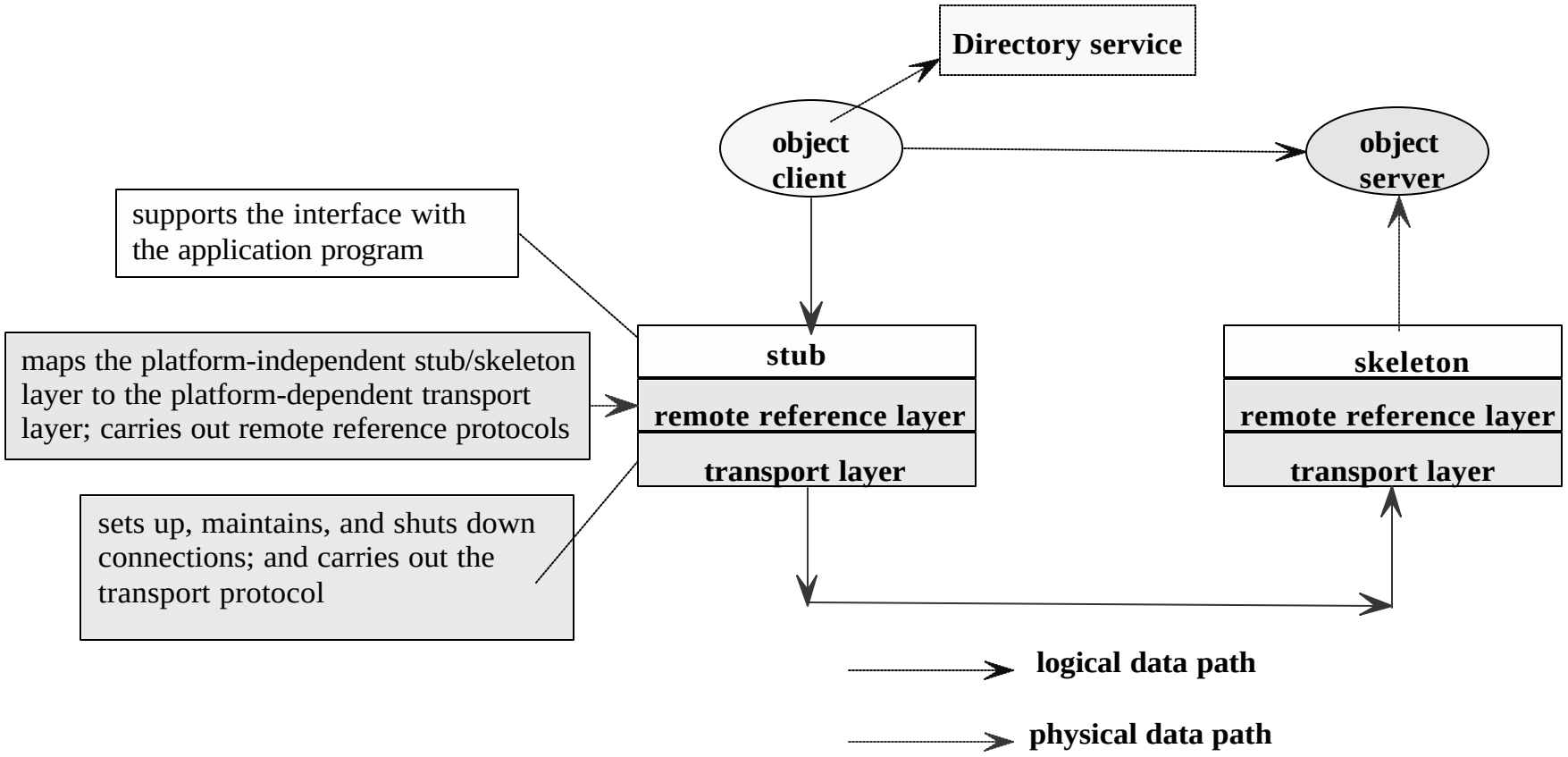
RPC: Remote Computation - 3



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
  - The remote interface is implemented by the remote object
  - An object client accesses the object by invoking the remote methods associated with the objects

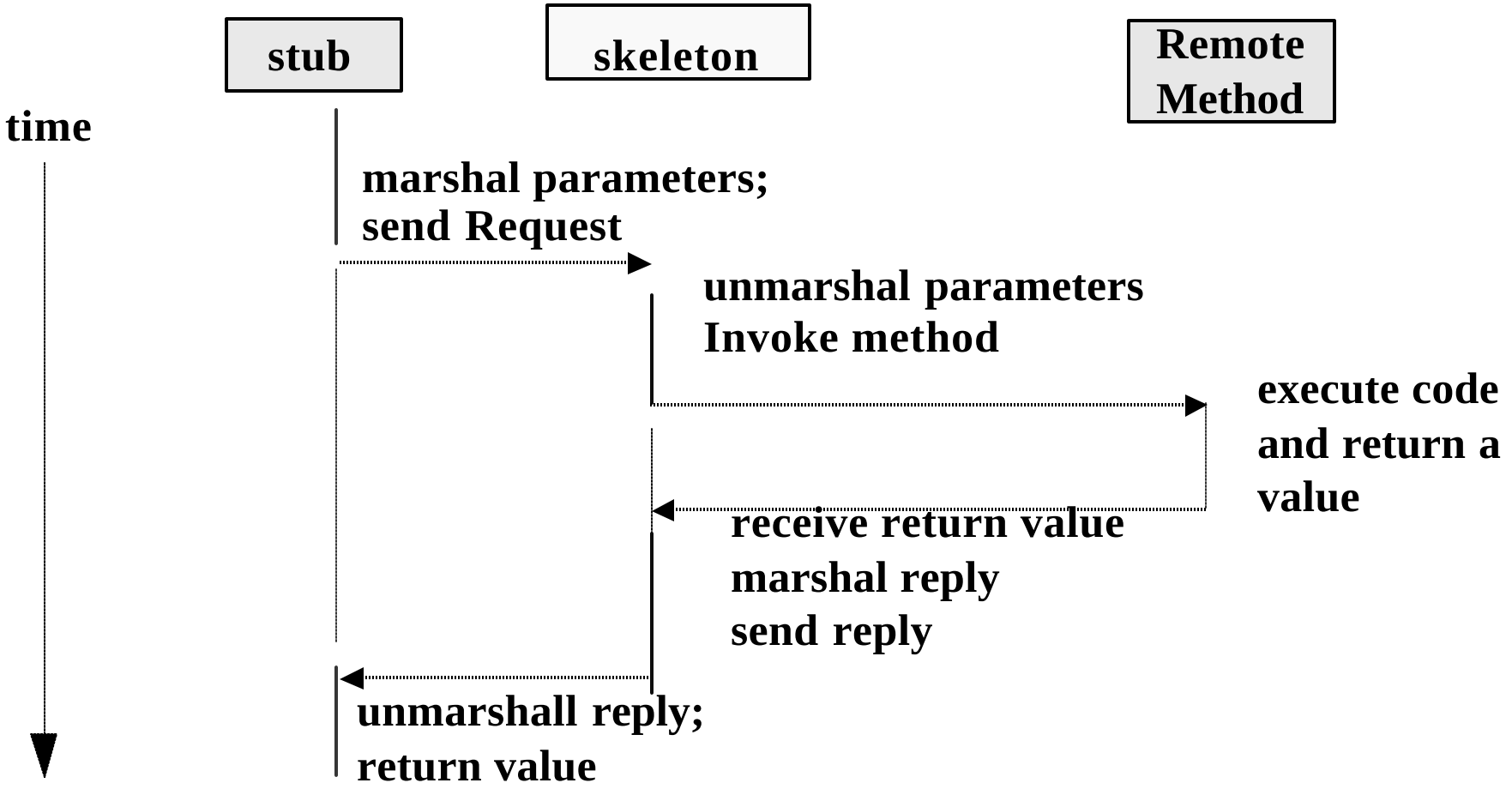
The Java RMI Architecture



Object Registry

- The RMI API allows a number of directory services to be used 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)
- 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
- An alternative naming/directory service is 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

The interaction between the stub and the skeleton



(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 Generation
  - The Object Server
- The Client-side Software

The Remote Interface

- A Java remote interface is an interface that inherits from the Java Remote interface
- 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

A sample remote interface

```
import java.rmi.*
public interface SomeInterface extends Remote {
    public String someMethod1( )
        throws java.rmi.RemoteException;

    public int someMethod2( float f) throws
              java.rmi.RemoteException;

   [signature of other remote methods may follow]
}
```

A sample remote interface - 2

- The java.rmi.RemoteException must be listed in the *throw* clause of each method signature
- This exception is raised when errors occur during the processing of a remote method call, and the exception is required to be caught in the method caller's program
- Causes of such exceptions include exceptions that may occur during interprocess communications, such as access failures and connection failures, as well as problems unique to remote method invocations, including errors resulting from the object, the stub, or the skeleton not being found

The Server-side Software

- The object server provides the methods of 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

The Remote Interface Implementation

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

public class SomeImpl extends UnicastRemoteObject
     implements SomeInterface {
   public SomeImpl() throws RemoteException {
      super( );
   }
   public String someMethod1( ) throws RemoteException {
      // code to be supplied
   }
   public int someMethod2( float f )
          throws RemoteException {
      // code to be supplied
   }
}
```

Stub Generation

- In RMI, each distributed object requires a proxy for the object server and the object client
- This proxy is generated from the implementation of a remote interface using a tool provided with the Java SDK: the RMI compiler *rmic*
  - **`rmic –v1.2 SomeImpl`**
- As a result of the compilation, the proxy class will be generated, prefixed with the implementation class name:
  - **`SomeImpl_Stub.class`**

The Object Server

The object server class instantiates and exports an object implementing the remote interface

```
import java.rmi.*;

public class SomeServer  {
   public static void main(String args[]) {
    […]
    try{
      SomeImpl exportedObj = new SomeImpl();
      Naming.rebind(“some”, exportedObj);
          System.out.println("Some Server ready.");
       }
    […]
   }
}
```

The Object Server - 2

- The Naming class provides methods for storing and obtaining references from the registry
- The rebind method allows an object reference to be stored in the registry with a URL
- The rebind method will overwrite any reference in the registry bound with the given reference name
  - if the overwriting is not desirable, there is also a bind method
- The host name should be the name of the server, or simply localhost. The reference name is a name of your choice, and should be unique in the registry

The RMI Registry

- The RMI Registry is required to run on the host of the server which exports remote objects
- It can be activated by hand using the *rmiregistry* utility as follows:

```
rmiregistry  <port number>
```

  where the port number is a TCP port number
- If no port number is specified, port number 1099 is assumed
- The registry will run continuously until it is shut down (via CTRL-C, for example)

The Object Server - 3

- 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 services of the object
- An RMI object is a concurrent server: each request from an object client is served using a separate thread
- 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
  - looking up the remote reference for the server object
  - the reference can then be cast to the remote interface and the remote methods invoked

The Client-side Software - 2

```
import java.rmi.*;

public class SomeClient {
   public static void main(String args[]) {
       […]
       try {
       String registryURL =
         "rmi://localhost:" + portNum + "/some";
       SomeInterface h =
         (SomeInterface)Naming.lookup(registryURL);
       String result = h.someMethod1();
       […]
       }
       catch (Exception e) {[…]}
   }
}
```

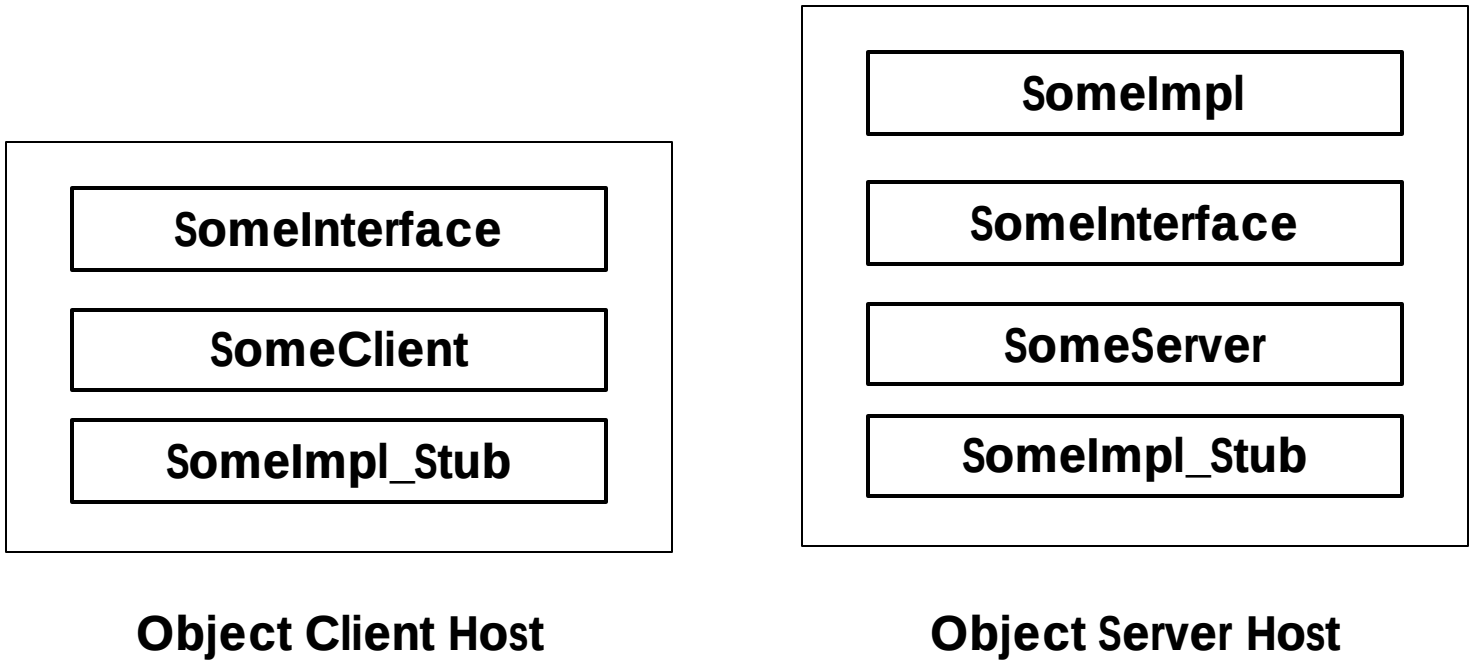
Invoking the Remote Method

- The syntax for the invocation of the remote methods is the same as for local methods
- It is a common mistake to cast the object retrieved from the registry to the interface implementation class or the server object class
- It should be cast as the interface

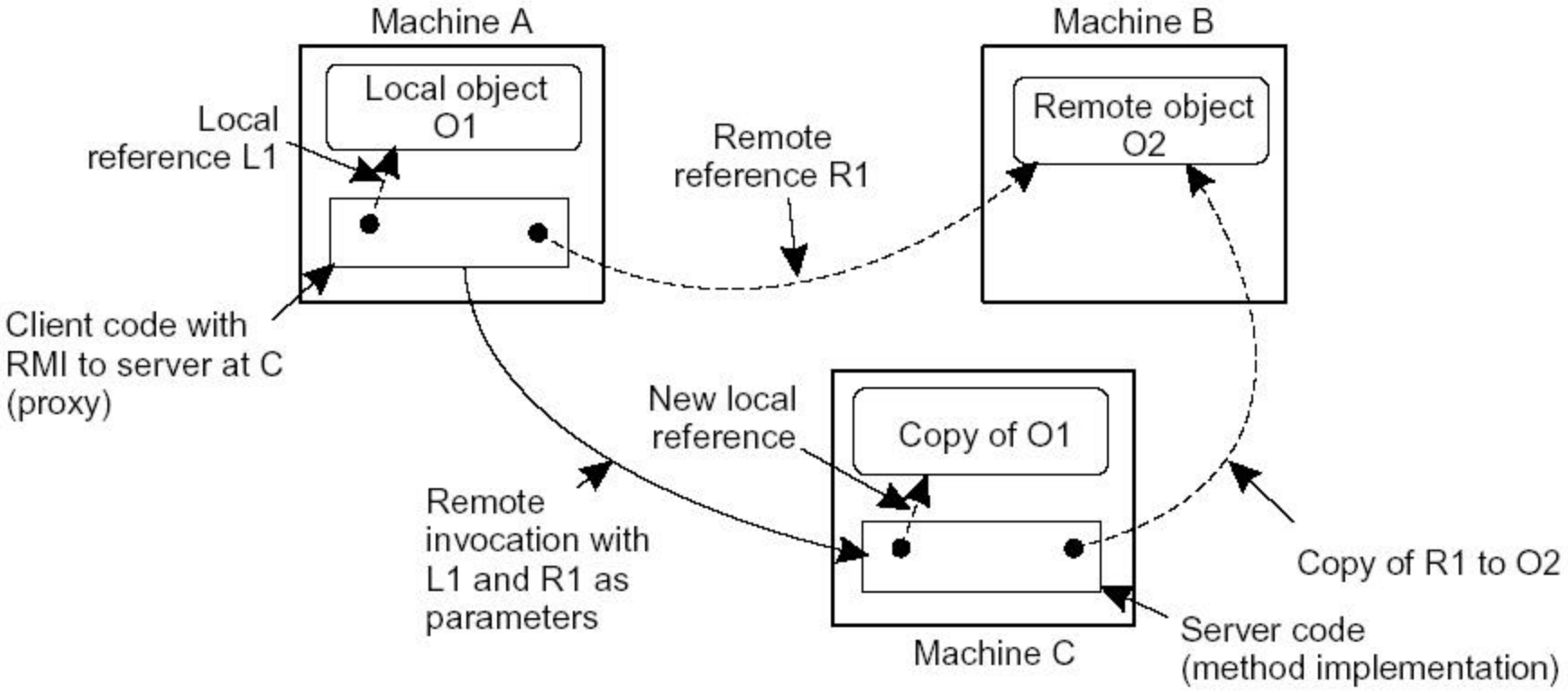
The stub file for the object

- The stub file for the object, as well as the remote interface file, must be shared with each object client as these files 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

Placement of classes for an RMI application



Local and Remote References



RMI and Sockets

- The remote method invocation can be used in lieu of the socket API in a network application
- Some of the tradeoffs between the RMI API and the socket API are as follows:
  - The socket API is closely related to the operating system, and hence has less execution overhead. For applications which require high performance, this may be a consideration
  - The RMI API provides the abstraction which eases the task of software development. Programs developed with a higher level of abstraction are more comprehensible and hence easier to debug