Communication in Distributed Systems –Part 2

REK's adaptation of

Tanenbaum's

Distributed Systems

Chapter 2

Communication Paradigms

- *Using the Network Protocol Stack*
- *Remote Procedure Call - RPC*
- *Remote Object Invocation - Java Remote Method Invocation*
- *Message Queuing Services - Sockets*
- *Stream-oriented Services*

Remote Object Invocation

Distributed Objects

Remote Method Invocation

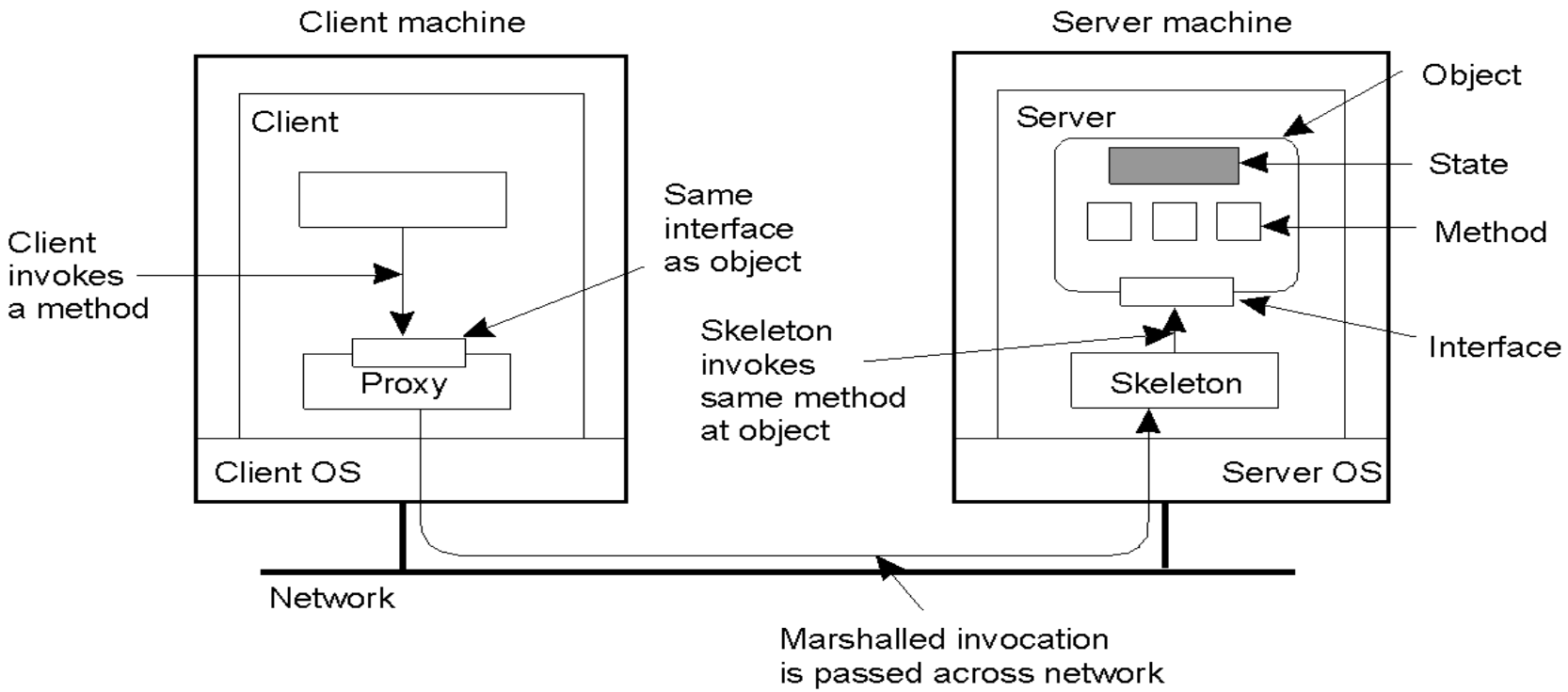
Distributed Objects

- *The idea of distributed objects is an extension of the concept of remote procedure calls.*
- *In a system for distributed objects, the unit of distribution is the object. That is, a client imports a "something".*
- *"full blown" object-based distributed sysems include* Corba *and* DCOM.

Distributed Objects

- *Key feature of an object:: it encapsulates data, the state, and the operations on those data, the methods.*
- *Methods are made available through an interface. The separation between interfaces and the objects implementing these interfaces is crucial for distributed systems.*

Distributed Objects



Common organization of a remote object with client-side proxy.

Distributed Objects

- *When a client binds to a distributed object, an implementation of the object's interface, a proxy, is loaded into the client's address space.*
- *The proxy marshals method invocations into messages and unmarshals reply messages to return the result of the method invocation to the client.*

Distributed Objects

- *The actual object resides at a server.*
- *Incoming invocation requests are first passed to a server stub, a skeleton, that unmarshals them to proper method invocations at the object's interface at the server.*
- *The skeleton also marshals replies and forwards replies to the client-side proxy.*
- *A characteristic of most distributed objects is that their state is **not** distributed.*

Compile-time vs Run-time Objects

- *The most obvious form of objects, compile-time objects, are directly related to language-level objects supported by Java and C++.*
- *A class is a description of an abstract type in terms of a module with data elements and operations on that data.*

Compile-time vs Run-time Objects

- *Using compile-time objects in distributed systems makes it easier to build distributed applications.*
- *An object can be fully defined by means of its class and the interfaces that the class implements.*
- *Compiling the class definition results in code that allows it to instantiate Java objects.*
- *The interfaces can be compiled into client-side and server-side stubs that permit Java objects to invoked by a remote machine.*

Compile-time vs Run-time Objects

- *Compile-time objects are dependent on particular programming language.*
- *With run-time objects, the implementation is left "open" and this approach to object-based distributed systems allows an application to be constructed from objects written in multiple languages.*
- *This scheme may use object adapters that act as wrappers that give implementations an object appearance.*

Binding a Client to an Object

- *Unlike RPC systems, systems supporting distributed objects usually support systemwide object references that can be passed between processes on different machines as parameters to method invocations.*
- *When a process holds an object reference, it must first bind to the referenced object before invoking any of its methods.*
- *Binding results in a proxy being placed in the process's address space, implementing an interface containing the methods.*
- *The binding can be implicit or explicit.*

Binding a Client to an Object

(a) An example of implicit binding using only global references.

```
Distr_object* obj_ref;          //Declare a systemwide object reference
    obj_ref = ...;              // Initialize the reference to a distributed object
    obj_ref-> do_something();   // Implicitly bind and invoke a method
```

(b) An example of explicit binding using global and local references.

```
Distr_object objPref;           //Declare a systemwide object reference
    Local_object* obj_ptr;      //Declare a pointer to local objects
    obj_ref = ...;              //Initialize the reference to a distributed object
    obj_ptr = bind(obj_ref);    //Explicitly bind and obtain a pointer to the local proxy
    obj_ptr -> do_something();  //Invoke a method on the local proxy
```

Implementation of Object References

- *A simple object reference would include:*
  - *The network address of the machine where the actual object resides*
  - *An endpoint identifying the server that manages the object.* [This corresponds to a local port assigned by the server's OS.]
  - *An indication of which object – an object number assigned by the server.*
- *This scheme can use a location server to keep track of the location of an object's server.*

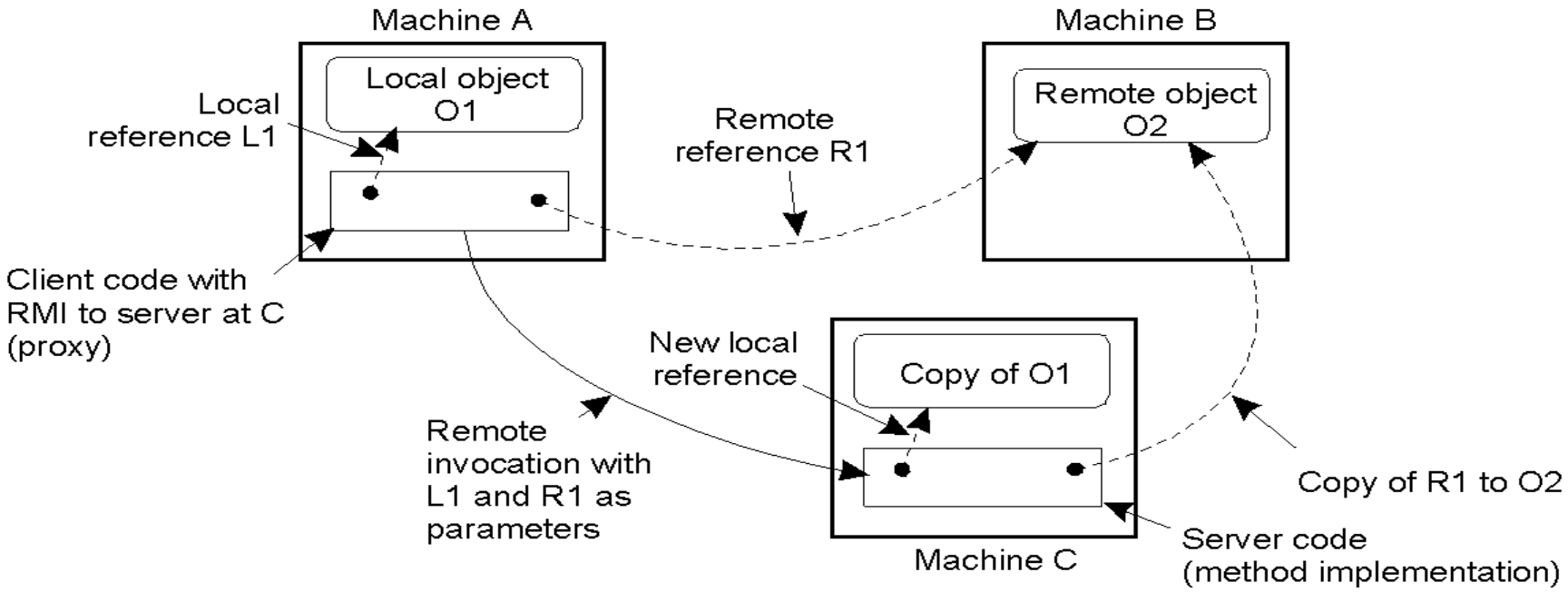
Remote Method Invocation

- *After a client is bound to an object, it can invoke the object's method through the proxy.*
- *Such a remote method invocation (RMI) is similar to a RPC with respect to marshaling and parameter passing.*
- *The difference is that RMI supports systemwide object references.*

Remote Method Invocation

- *An interface definition language is used in RMI to specify the object's interface.*
- *Static invocation implies using object-based languages (e.g., Java) to predefine interface definitions.*
- *Dynamic invocation permits composing a method invocation at run-time.*

Parameter Passing



The situation when passing an object by reference or by value.

Java RMI

- *Distributed objects have been integrated in the Java language with a goal of a high degree of distribution transparency.*
- *Java adopts remote objects as the only form of distributed objects. [i.e., objects whose state only resides on a single machine]*
- *Java allows each object to be constructed as a monitor by declaring a method to be synchronized.*

Java RMI

- *However there are problems with distributed synchronization.*
- *Thus, Java RMI restricts blocking on remote objects only to the proxies.*
- *This means remote objects **cannot** be protected against simultaneous access from processes operating on different proxies by using synchronization methods.*
- *Explicit distributed locking techniques must be used.*

Java Remote Object Invocation

- *Any primitive or object type can be passed as a parameter to an RMI provided the type can be marshaled. [i.e, it must be serializable.]*
- *Platform dependent objects such as file descriptors and sockets cannot be serialized.*
- *In Java RMI reference to a remote object is explained on slide 14.*
- *A remote object is built from a server class and a client class.*

Java Remote Object Invocation

- *Proxies are serializable in Java.*
- *This means proxies can be marshaled.*
- *In actually an implementation handle is generated, specifying which classes are needed to construct the proxy.*
- *The implementation handle replaces the marshaled code as part of a remote object reference.*
- *This passing of proxies as parameters works only because each process is executing the same Java virtual machine.*