CMPSCI 677 Operating Systems **Spring 2019**

Lecture 9: February 25

Lecturer: Prashant Shenoy *Scribe: **Ge Shi***

9.1 Lightweight RPC (LRPC)

Lightweight RPCs are the special case of RPCs where calling process and the called process are on the same machine.
Remember the two forms of communication of a distributed system – explicit (passing data) and implicit (sharing memory). You can think of not using a RPC system for the special case that both processes are on the same machine but using a shared piece memory. The optimization is to construct the message as a buffer and simply write to the shared memory region.
When client and server both are two processes on the same machine and you make RPC calls between two components on the same machine, following are the things which can make it better over the traditional RPC:

1. No need for the marshalling here.
2. We can get rid of explicit message passing completely. Rather shared memory is used as a way of communication.
3. Stub can use the run-time flag can be used to decide whether to use TCP/IP or shared memory.
4. No XDR is required.

Steps of execution of LRPC:

1. Arguments of the calling process are pushed on the stack,
2. Trap to kernel is send,
3. After sending trap to kernel, it either constructs an explicit shared memory region and put the arguments there or take the page from stack and simply turn it into shared page,
4. Client thread executed procedure (OS upcall),
5. Then the thread traps to the kernel upon completion of the work,
6. Kernel again changes back the address space and returns control to client,

Q: If the two processes are on the same machine, why do we need to use a networks stack because you are not communicating over network.
A: If you are using standard RPCs, your package will go down OS and the IP layer will realize they are the process on the same machine. It will come back up and call another process.

Note: RPCs are called "doors" in SUN-OS (Solaris).

9.2 Other RPC models

Traditional RPC uses Synchronous/blocked RPC, where the client gets blocked making an RPC call and gets resumed only after getting result from the called process. There are three other RPC models described below:

Figure 9.1: Traditional RPC

9.2.1 Asynchronous RPC

In Asynchronous or non-blocking RPC call, the client is not blocked after making an RPC call. Rather it sends the request and waits for an acknowledgement from the called process and after getting the acceptance, it resumes the execution.

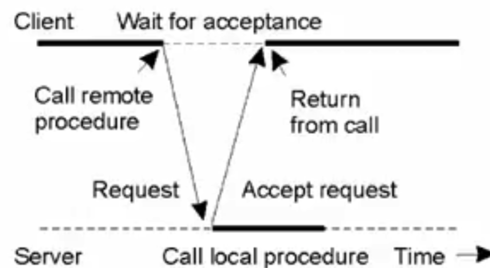

Figure 9.2: Asynchronous RPC

9.2.2 Deferred synchronous RPC

This is just a variant of Non-blocking RPC as in this Client and server interact through two asynchronous RPCs. Here the server send the replies to the client by making another asynchronous RPC call.

9.2.3 One-way RPC

It is also a form of asynchronous RPC where the waits for an acknowledgment from the server. But in this one-way RPC call, the client doesn't even for an acknowledgment from the server. It starts its execution

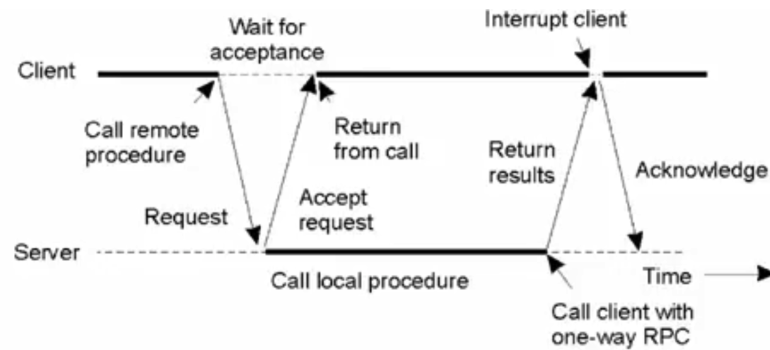

Figure 9.3: Deferred Synchronous RPC

just after sending a request. This model has one disadvantage that it doesn't guarantee the reliability as the client doesn't know whether the request reaches the server or not.

9.3 Remote Method Invocation (RMI)

RMIs are RPCs in Object Oriented Programming Mode i.e., they can call the methods of the objects (instances of a class) which are residing on a remote machine. Here the objects hide the face that they are remote.The functional is called just like it is called on a local machine. For eg: obj.foo() , where obj is the object and foo is its public function. Some important facts about RMIs:

1. There is equivalent separation between interface and the implementation as the interface is residing on the client machine whereas the implementation is on server machine.
2. It supports system-wide object references i.e parameters can be passed as object references here (which is not possible in normal RPC)

Figure 9.4 is showing an RMI call between the distributed objects. Just like a normal RPC, here also there is no need to setup socket connections separately by the programmer. Client stub is called the proxy and the server stub is called the skeleton and the instantiated object is one which is grayed in figure.

Now, when the client invokes the remote method, the RMI call comes to the *stub* (Proxy), it realizes that the object is on the remote machine. So it sets up the TCP/IP connection and the marshalled invocation is sent accross the network. Then the server unpacks the message, perform the actions and send the marshalled reply back to the client.

9.3.1 Proxies and Skeletons : Client and Server Stub

- Working of Proxy : Client stub
    1. Maintains server ID, endpoints, object ID.
    2. Sets up and tears down connection with the server
    3. Serializes (Marshalling) the local object parameters.

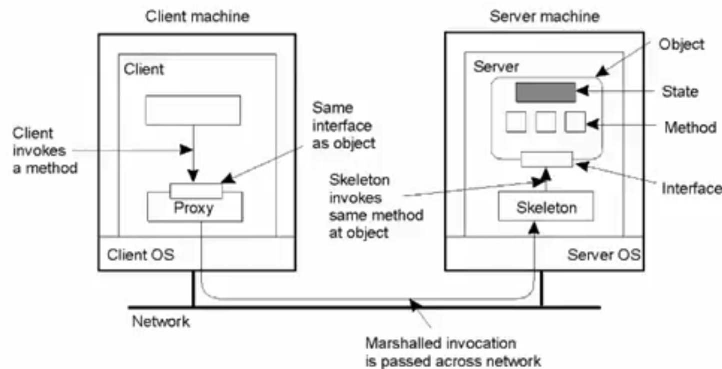

Figure 9.4: Distributed Objects

- Working of Skeleton : Server stub

  It deserializes and passes parameters to server and sends results back to the proxy.

  Q: If the remote object has some local state (variable), you make a remote call and changed the variable, will the local machine see the change?
  A: There's only one copy on the server and no copy of it on the client at all. The objects on the server and client are distinct objects. The change will be visible to subsequent methods from client. It doesn't mean there's a copy of the object on the client. If you make another call and see what's the value that variable, you'll get the new one.

9.3.2 Binding a Client to an Object

Binding can be of two types : implicit and explicit. Section (a) of Figure 9.5 shows an implicit binding, which is using just the global references and it is figured out on the run-time that it is a remote call (by the client stub). In section (b), explicit binding is shown, which is using both global and local references. Here, the client is explicitly calling a bind function before invoking the methods. Main difference between both the methods is written in Line 4 of the section (b), where the programmer has written an explicit call to the bind function.

```
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
```

(a)

```
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
```

(b)

Figure 9.5: Implicit and Explicit Binding of Clients to an Object

9.3.3 Parameter Passing

RMIs are less restrictive than RPCs as it supports system-wide object references. Here, Passing a reference to an object means passing a pointer to its memory address over the network. In Java, local objects are passed by value, and remote objects are passed by reference. Figure 9.6 shows an RMI call from Machine A(client) to the Machine C (server - called function is present on this machine) where Object O1 is passed as a local variable and Object O2 is passed as a reference variable. Machine C will maintain a copy of Object O1 and access Object O2 by dereferencing the pointer.
Note: Since a copy of Object O1 is passed to the Machine C, so if any changes are made to its private variable, then it won't be reflected in the Machine C. Also, Concurrency and synchronization needs to be taken care of.

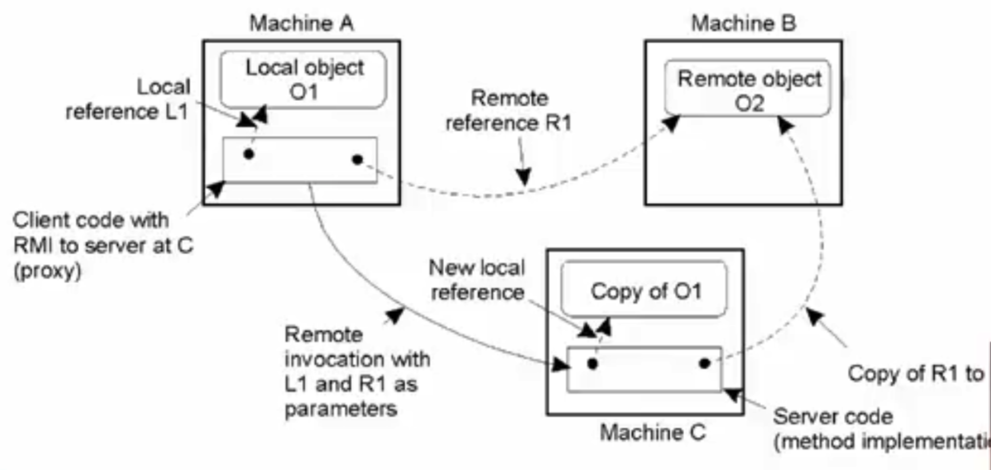

Figure 9.6: Parameter Passing : RMI

Q: How do network pointers interact with Java's garbage collection? A: Garbage collection of Java is going to delete the memory that is not in use. A short answer is the remote machine shouldn't do garbage collection because you don't know if the object is being used by other machine.

9.3.4 DCE Distributed-Object Model

Below are some examples of how invocation of remote objects works across many systems. Clients making requests to server using RMI calls. In section (a) of Figure 9.7, every request instantiates a new thread and new local copy of the object (private to that thread). Here after sending the reply, thread is deleted. In section (b), all requests are operated on the same object.There are multiple threads and they all are calling methods of the same object. So all threads are using the state of the object. If one thread make the changes, they will be visible to other threads. Here also, concurrency and synchronisation needs to be taken care of.

9.4 Java RMI

- Server:

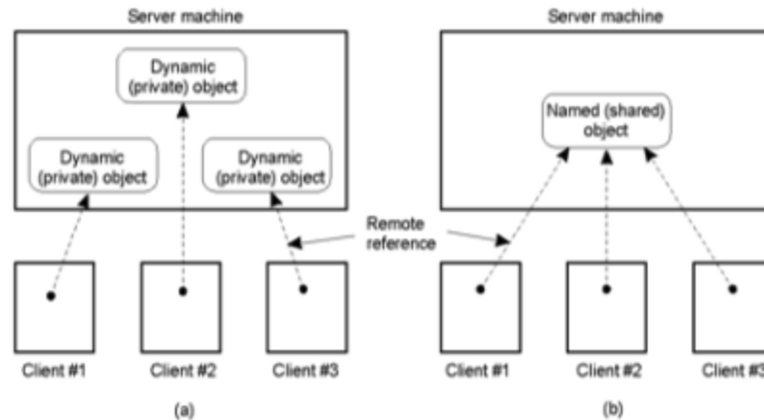

Figure 9.7: Distributed Object Models

- The server defines the interface and implements the interface methods. The server program creates a server object and registers object with ”remote object” registry (Directory service).

- Client:
  - It looks up the server in remote object registry, and then make the normal call to the remote methods.

- Java tools:
  - `rmiregistry`: Server-side name server
  - `rmic`: Uses server interface to create client and server stub. it is a RMI Compiler, which creates an autogenerated code for stubs.

9.4.1 Java RMI and Synchronisation

Java supports monitors , which are the synchronised objects. The same method can be used for remote method Invocation which allows concurrent requests to come in and synchronisation them. So for synchronisation, lock has to applied on the object which is distributed amongst the clients. How to implement the notion of the distributed lock?

1. Lock at the server : Here, clients will make requests to the server, where they will contend for the lock and will be blocked (waiting for the lock).

2. Lock at the client (proxy) collectively : They will have some protocol which decides which client will get the lock and rest others will be blocked (waiting for the lock)

Note: Java uses proxies for blocking : which means client side blocking.

9.5 Message-oriented Transient Communication

Here in Figure 9.8, It has been shown how the client and sever can communicate using vanilla network sockets. It is the most primitive form of network communication that uses TCP/IP or UDP/IP. Here, firstly a socket is created and then it is used to connect the client to a server and then use that network for communication between them.
Steps taken by the client and server to establish socket networking:

1. Create a new socket
2. Bind the socket or assign the port number to it which will allow the socket to listen on that port number.
3. Then the server is ready to accept the network packets and will wait for the client to send the packet.
4. Then the client will make a socket call (to construct a socket) and make a connection to the server using the server's IP address and the port number to which it is listening.
5. Then they will be connected and they will use the read-write or send-receive call.
6. Once done Close the socket connection

Note: Socket communication is blocking/synchronous in nature.

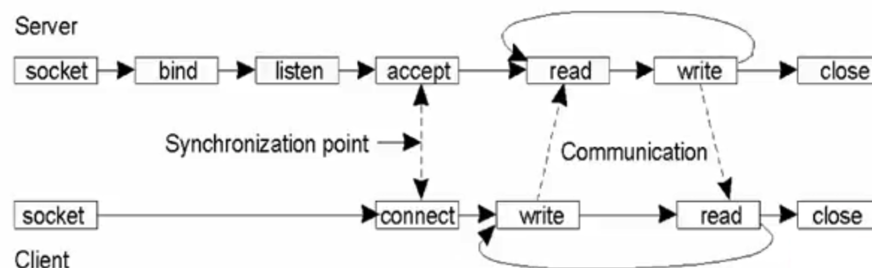

Figure 9.8: Socket communication

One example of sockets are the Berkeley sockets(Figure 9.9 since they appeared in Berkeley System Distribution UNIX (BSD UNIX), but now they are part of the POSIX standards.

9.6 Message-Passing Interface (MPI)

MPI(Message Passing Interface) is a library/middleware designed for parallel or distributed applications which need to do a lot of IPC. Since traditional TCP/IP imposes a high overhead, so there is a need of an application which provides control over how data is send and receive that normal socket programming doesn't allow. MPI allows many types of send and receive whereas in socket programming, there is only one type of send-receive.

Figure 9.10 shows a list of MPI primitives. MPI has six different variants of send and 2 variants of receive. Here are some of them explained below:

Primitive	Meaning
Socket	Create a new communication endpoint
Bind	Attach a local address to a socket
Listen	Announce willingness to accept connections
Accept	Block caller until a connection request arrives
Connect	Actively attempt to establish a connection
Send	Send some data over the connection
Receive	Receive some data over the connection
Close	Release the connection

Figure 9.9: Berkeley Primitives

Primitive	Meaning
MPI_bsend	Append outgoing message to a local send buffer
MPI_send	Send a message and wait until copied to local or remote buffer
MPI_ssend	Send a message and wait until receipt starts
MPI_sendrecv	Send a message and wait for reply
MPI_isend	Pass reference to outgoing message, and continue
MPI_issend	Pass reference to outgoing message, and wait until receipt starts
MPI_recv	Receive a message; block if there are none
MPI_irecv	Check if there is an incoming message, but do not block

Figure 9.10: MPI Primitives

1. **MPI_bsend**: Message placed in local buffer (but not sent out) and then return. (Non-blocking communication)
2. **MPI_send**: Wait until message gets to remote machine and waits for acknowledgement.
3. **MPI_sscend**: Wait until receipt starts. It is like Asynchronous/Non-blocking RPC call.
4. **MPI_sendrecv**: After sending message, it will wait for reply. Just like in Berkeley primitives (socket programming) or Synchronous RPC.
5. **MPI_isend**: Send a pointer to buffer, not even send a copy.
6. **MPI_recv**: It is a blocking receive.
7. **MPI_irecv**: It is a Asynchronous/Non-blocking receive.