CMPSCI 677 Operating Systems **Spring 2014**

Lecture 9: February 26

Lecturer: Prashant Shenoy *Scribe: Nimish Gupta*

9.1 RPC Implementation issues

RPC on LAN using UDP works fine because data packets don't get lost in the network, but it doesn't work over WAN since routers gets congested and packet dropping etc might disrupt the RPCs. TCP is a better way to send packets over WAN because of its reliability properties, but adds communication overhead when compared to UDP. Thus a RPC programmer has to choose a protocol to implement RPC.

There is significant overhead involved in sending a RPC message over network. When a RPC call is made, it goes from client into the stub, in the stub the parameters are copied and then a message is formed to send it to the server, the message is then handed to the kernel which then adds network layer's headers and pushes packet out of NIC card over the medium. Similar process is done at the server side. Thus a lot of copies of a message are made which can contribute to overhead, specially when there are lot of message with fairly large number of arguments.

9.1.1 Lightweight RPCs

Lightweight RPC is suitable when client and server processes are both running on the same machine. Although kernel tries to minimize overhead when it detects that the packets are addressed to itself, the overhead of constructing a message still cannot be avoided. Thus rather than sending an explicit network message, the client just passes a buffer from client to the server via a shared memory region where client puts in the RPC request and the parameters and tells the server to access it from there.

The client pushes the arguments onto the stack, trap to the kernel, the kernel in turn just takes the memory region where arguments were pushed in the stack and change the memory map of the client so that that memory region now becomes available to the server. The server then takes this request from the memory region and processes it. Once the execution on server side finishes the reply is sent back to the client in a similar fashion. Using this unstructured form of communication, overhead of sending messages over network could be avoided.

The decision of using Lightweight RPCs can be made either at compile time or at run time. For compile time support, linking against Lightweight RPCs is forced over usual RPC using a compiler flag. The other option is to compile in both Lightweight and usual RPC support and choose one at runtime. It is then handled by the RPC runtime system, it can decide whether to send a message over TCP or using shared buffers.

Doors in solaris OS was the first implementation of Lightweight RPCs.

9.1.2 Other RPC Models

- *Synchronous RPC* - Synchronous remote procedure call is a blocking call, i.e. when a client has made a request to the server, the client will wait until it receives a response from the server.

- *Asynchronous RPC* - Client makes a RPC call and it waits only for an acknowledgement from the server and not the actual response. The server then processes the request asynchronously and send back the response asynchronously to the client which generates an interrupt on the client to read response received from the server. This is useful when the RPC call is a long running computation on the server, meanwhile client can continue execution.
- *Deferred Synchronous RPC* - The client sends a RPC request to the server, and client waits only for acknowledgement of received request from server, post that the client carries on with its computation. Once the server processes the request, it sends back the response to the client which generates an interrupt on client side, the client then sends an response received acknowledgement to the server.
- *One-Way RPC* - The client sends an RPC request and doesn't wait for an acknowledgement from the server, it just sends an RPC request and continues execution. The reply from the server is handled through interrupt generated on receipt of response on client side. The downside here is that this model is not reliable. If it is running on non-reliable transport medium such as UDP, there will be no way to know if the request was received by the server.

9.2 Remote Method Invocation (RMI)

RPC abstractions applied to objects in Object Oriented World is called as RMI. Applications are written as classes and the classes are instantiated as objects during runtime, application can be distributed, thus some objects may run at client side and some on the server. Method in object 1 wants to invoke a method that is exported by object 2. Thus if the method belonging to that class resides on a different server then a remote method call will be sent to that server. The objects that reside on the local machine are called local objects, objects which are present on other machines are called remote objects, collectively these are called distributed objects.

RMI differs from traditional RPCs in two aspects

- One major difference between Java RMI's and RPCs is that RMI's support system wide object references, i.e. they allow references to objects system wide, thus this enables passing of arguments by reference. In RPCs this can't be done. In Java RMI's arrays or other data structures can't be passed as reference but objects can be passed as reference. However, array can be a member variable of a object and that object can be passed as reference in RMI call. Passing by reference reduces the complexity from users perspective but increases the runtime complexity of the system as the system needs to figure out the system wide global reference corresponding to the passed object. References are essentially network pointers and may comprise of ip of target machine, process id of on target machine etc.
- Another difference is that a programmer doesn't need to know RPC interface at compile time. Binding to an interface is performed at run-time. There are two ways in which an interface could be bound in RMI world: Implicit and Explicitly. In implicit binding the runtime system (upon the very first remote call) figures out the details of the target object and stores it for subsequent operations. In explicit binding a programmer needs to perform a bind call explicitly.

9.2.1 DCE Distributed-object Model

In this system, the client server application is implemented using private objects. Whenever a request comes in to the server, the server creates a new object to service that request. This can be thought of as spawning a new thread to process a request. This is a transient model, the object is destroyed once the request is serviced by it. Thus state cannot be included in this server, it will be a stateless server, as the object gets destroyed once the request is serviced. This implementation is not available in Java.

If the state has to be kept in this kind of model, it will require *Shared Objects*. Shared objects are persistent, these get created when server starts up and stays for the lifetime of the server.

Thus while making a RMI call, the programmer can state whether the call has to goto private objects or shared objects. The shared objects can run on multiple threads which execute different methods of those objects.

This is not used anymore.

9.2.2 JAVA RMI

- At the server side an interface is defined, i.e. it is the set of methods that the server is exporting to any client which wishes to make RMI calls. For each method exposed by interface, there has to be separate implementation written for it. Thus there is a separation between interface and implementation. The interface is exported and registered with the server. Once the server starts up, the remote object is registered with the "remote object" registry which is similar to directory service.
- When the client starts up it has to look up the remote object registry to find where a particular remote object resides.
- Rmiregistry is the server-side name server.

9.2.3 JAVA RMI and Synchronization

When there are objects distributed across machines, then problem of synchronization become more evident. In a multi threaded java program, locks and monitors are implemented to get synchronization. Now if the program is distributed over machines it becomes a challenge to implement locking but still it needs to be done to attain synchronization. The locking functionality can be implemented at the client end or the server end.

- *Synchronize at the server end*, when multiple request from client come in, the access to the shared object on server needs to be synchronized. Essentially, a new request tries to grab a lock on the object, which the current request is processed, other request are queued and processed sequentially when the lock is released.
- *Synchronize at the client end*, each client will have locking mechanism and they will coordinate the synchronization. A client will have to wait for a lock on the object, if another client has locked on that resource, this essentially becomes distributed lock.

9.3 Message-oriented Transient Communication

When a server and client want to communicate, each has to instantiate a socket and then connect the sockets together. Sockets are low-level primitives in network programming.

- At server end, after a socket is instantiated it is bound to a port number using *bind* call. Then server implements *listen* call which listens for any requests on that port. When a new request comes in,*accept* call is made to accept the request from the client. Until the client tries to connect to the server, accept implements a blocking call.

- At the client end, after the socket is instantiated, it uses connect call to connect to a specified server.
- Once the connection is established between client and server, *read* and *write* calls maybe used to read and write data from and to the socket. Sockets are duplex, they support both read and write calls.
- At the end of communication, *close* call can be used to close to socket and hence the communication between client and server.

9.4 Message-Passing Interface (MPI)

These are efficient low-level socket type communication primitives. The major use case is for parallel tasks running over clusters to communicate with one another. MPI is a middleware which is implemented for high performance communication such as astronomical computations etc. It can give better abstraction and lower overheads as compared to TCP / IP. It is designed for parallel applications or transient communication. MPI Primitives provides abstraction for both synchronous and asynchronous communications.

Some examples of MPI Primitives:-

- MPI_bsend is equivalent to a One Way RPC, the machine sends the message and doesn't wait for the message to reach the other end.
- MPI_send and MPI_ssend are equivalent to asynchronous RPCs, the machine waits for the message to be received at the other end and then continue execution.
- MPI_sendrecv is equivalent to Synchronous RPC, the machine waits for the request to be process and once it receives the repines, it continues execution.