CMPSCI 677 Operating Systems **Spring 2013**

Lecture 9: February 20

Lecturer: Prashant Shenoy *Scribe: Siddharth Gupta*

9.1 Failure Semantics

Failures should be taken into consideration when RPC calls are made. Failures can be due to client or server crash or data transmission loss in the network, thus we need to have predictable behavior in scenarios where such failures can occur. There should be mechanisms to figure out the cause of failures and the steps to be taken to overcome them.

- If the client is unable to find the server, then the client should return an error message.
- If client and server both are running but something goes wrong on the network and RPC request packet doesn't reach the server for processing or the server sent the reply back to client but client never received the reply. To handle this scenario the client should implement a timeout mechanism. If the client doesn't receive response from server before the timeout then client should resend the RPC request.
- Another approach of averting failures is by offloading the error correction task to the TCP layer, since TCP provides abstraction where the packet losses are dealt by the transport layer. Thus any packet that is sent will arrive at the other end eventually.
- If the RPC system is running over UDP, then timeout mechanism needs to be in place since UDP doesn't implement any error correction or handles packet losses in the network.
- If UDP is used to transfer data, before making a request to a stateful server one need to check if the RPC calls are idempotent, i.e. re-executing the calls on the server shouldn't cause errors.

9.1.1 Server Failure Semantics

There can be scenarios where the server crashes while it is executing the RPC call, thus client should be aware of what was the state of the server before it crashed, so that it will know what action to take when the server comes up. This can be handled by different semantics provided by the RPC system.

- *At least once semantics*, denotes that if the client received a reply from the server, it means that call has been executed at least once on the server. It may be possible server executed the call and then crashed before sending the reply, it then comes up and then again executes the call and sends the reply to the client.
- *At most once semantics*, if the client gets a reply from the server, it means that the RPC call have been executed at most once on the server.
- *Exactly once semantics*, this is the most desirable scenario, no matter what happened on the server side (crashes / restarts), if the client gets back reply exactly once, then it is confirmed that the RPC call was executed exactly once on the server.

Exactly once semantics are difficult to implement and systems generally implement weaker semantics and thus the error handling needs to be taken care explicitly.

9.1.2 Client Failure Semantics

There can be failures scenarios at the client side too. For example, client made a RPC request, but before the client received the response from the server, it crashed, in this case server will need to know the corrective action in order to maintain system sanity. Thus there are failure semantics provided by the RPC system to handle such scenarios.

- *I*f server executed a RPC call, but mean while the client crashed and is not available to receive the response, then most servers will discard the response
- If server was executing a long running RPC computation and in between the execution it comes to know that client has crashed then such calls are called *Orphan RPC calls*. In this scenario server can do one of the following
  - *Extermination* - when the server gets to know that client has crashed, it can terminate the on-going RPC computation. This will incur an overhead on server side to maintain list of clients and their current state.
  - *Reincarnation* - In order to minimize the overhead, the server can check for long running RPC calls within certain periodic time intervals. If it finds that a certain RPC call has been running for a long duration, then it goes and checks for the client state, if the client has crashed then server can delete the computation. Thus server doesn't need to keep state of every client connected to it.
  - *Gentle Reincarnation* - If the client that made the request has crashed, then server can broadcast and check if the client restarted with a different process id and then try to deliver the RPC response to it.
  - *Expiration* - Every time a RPC computation starts, the server gives it a timeout, once the timeout happens it checks if the client is still alive, and if it is then it continues computation with another timeout, this process recursively goes on until the response is sent back to the client.

9.1.3 Implementation Issues

RPC on LAN using UDP works fine because the transmission losses are less and data packets don't get lost in the network, but it won't pan out well when sent over WAN since routers gets congested and packet dropping etc might disrupt the RPCs. TCP works out to be a better way to send packets over WAN because packet drops, error correction everything is handled by TCP. Thus a RPC programmer has to choose which protocol he/she wants to implement for RPC.

There is a lot of overhead in sending the RPC message over the network, when a RPC call is made, the call goes from the 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 is copied on to the NIC card and sent over the medium, similar process is done at the server side. Thus we see that lot of copies of the message are made which can contribute to overhead, specially when there are lot of message with fairly large number of arguments.

9.1.4 Case Study : SUNRPC

To implement NFS, SUN Microsystems choose to use RPCs rather than doing socket communication between the client machine and the File Server. Thus they built an RPC layer and implemented NFS on top of it. They realized that the RPC layer was more generic, therefore other applications could be written using the same RPC abstraction, thus they made tools like RPC compilers available to programmers. This system was initially built on top of UDP since it was designed to work on LAN, but now-a-days it can work both on UDP and TCP which depends on the discretion of the programmer and application it is designed for. To implement Marshaling and Un-marshaling SUN came up with XDR (eXternal Data Representation), i.e. any packets that are sent out with arguments is converted to XDR format and sent as a message. SUNRPC provides At-least once semantics and thus not a preferred way, and calls need to be idempotent to avoid redundancies.

9.1.5 Lightweight RPCs

These occur when client and server processes both run on the same machine. In this scenario, the overhead of constructing a message is reduced as the communication is on the same machine and message need not be sent over a network. Thus rather than sending an explicit network message the client just passes a buffer from client to the server, essentially there is a shared memory region where client puts in the RPC request and the parameters and tells the server to access it from that.

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 in this request from the memory region and processes it, thus here we see that just a shared memory was passed to the server instead of sending the message over network. Once the execution on server side finishes the reply is sent back to the client in a similar fashion.

All of the above is handled by the RPC runtime system, it can decide whether to send a message over TCP or using shared buffers.

9.1.6 Other RPC Models

- *Synchronous RPC* - Synchronous call are generally blocking calls, i.e. when a client has made a request to the server, the client will wait / block until it receives a response from the server.
- *Asynchronous RPC* - Client makes a RPC call and it waits only till it receives 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 move on with its computations.
- *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 with its computation. 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)

RMI is when the abstraction of RPC is applied to objects in Object Oriented World. 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 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.

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.

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 an equivalent of spewing a new thread to process new request. This is a transient model, the object is destroyed once the request is serviced by it. Thus one conclusion is that state cannot be included in this server, it will be a stateless server, as the object gets destroyed once the request is serviced. This implementations 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.

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.

- *Implement lock 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.
- *Implement lock 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.

- 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.

9.3.1 Message-Passing Interface (MPI)

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 generally used in a cluster of servers. 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 even wait for the message to reach the other end.
- MPI_send and MPI_send 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.