CMPSCI 677 Operating Systems **Spring 2016**

Lecture 9: February 22

Lecturer: Prashant Shenoy *Scribe: Keping Bi*

9.1 RPCs continued

9.1.1 Lightweight RPCs

Lightweight RPC is a method of optimizing RPCs 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. No explicit message passing is needed as memory is shared to send and receive messages.

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 bit. 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 memory 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* -Here, the client and server interact through two asynchronous RPCs. 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, pass by reference is not allowed. In Java RMIs, arrays or other data structures can't be passed as reference but only objects can be passed as reference. Passing by reference reduces the complexity from users perspective but increases the runtime complexity of the system.
- 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 JAVA RMI

Client and server machines have some code called stubs. The client stub is known as 'proxy' and server stub is known as 'skeleton'. In this system, the client server application is implemented using objects. A server object has states and methods. These methods can be called from another machine. These objects have state, hence this is stateful. Interfaces are the API that the server will be exporting. 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 initially binds to the server, it imports the interface into the client address space at run-time and then it can call any of those methods defined in the interface. Rmiregistry is the server side name server and Rmic uses server interface to client and server stubs. Proxy maintains server ID,figures where the server is and what is its end-point. It is responsible for marshalling or de-marshalling of arguments known as Serialization.

Binding a client to an object can be done in two ways:

- Implicit Binding: No additional functions are called to bind to the server and a method on the server is called directly.

- Explicit Binding: bind() is called before a remote method call.

In an RMI call, local object is passed by value and a remote object is pass by reference by default. A copy of the local object is made in the server.

9.2.2 DCE Distributed-Object Model

Its an outdated middleware system. The idea is that Objects could be distributed across multiple machines and clients used objects to make RMI calls. They had a notion of named and private objects. Calls to private objects instantiate new copy of the object, unique to that call. In Java world, there are only named(shared) objects.

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. The locking functionality can be implemented at the client end or the server end. JAVA supports synchronization as a part of the specification.

- *Synchronize at the server end*, when multiple request from clients 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, while the current request is processed, other request are queued and processed sequentially when the lock is released. If the client crashes while blocked, the lock may not be given up or we block at the client.
- *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. Client crashes are simpler to handle. Java uses proxies for blocking.

9.3 Communication

Assume there are two end points that want to communicate with each other. There are multiple intermediaries, such as nodes and routers. Also, there are OSs that implement the TCP/IP. There are a few characteristics of a communication.

- *Persistence*:Persistence means that the network is capable of storing messages for a arbitrary period of time until the next receiver is ready. Email and ground deliveries are good examples.
- *Transient*:Messages are always forwarded to next hop assuming it is always available. If the next hop is not available then the message gets discarded. For example, Transport-level communication discards the message if the process crashes for any reason. The system will not store messages.
- *Asynchronous*:Asynchronous communication means that the sender is doing non-blocking sending. It continues immediately after submitting the message.
- *Synchronous*:Synchronous communication blocks the process until the message is received or the sender gets a response from the server.

9.3.1 Persistent synchronous communication

The sender is blocked when it sends the message, waiting for a acknowledgement to come back. The message is stored in a local buffer, waiting for the receiver to run and receive the message. Some instant message applications, such as Blackberry messenger, are good examples. When you send out a message, the app shows you the message is delivered but not read. After the message is read, you will receive another acknowledgement.

9.3.2 Persistent asynchronous communication

When sender sends a message, the receiver need not be running. The message is stored somewhere along the path. Message is delivered to receiver when it comes online. When the reciever comes online and receives the message, it is not required that the sender be running at that time. Example: email.

9.3.3 Transient asynchronous communication

Since the message is transient, both entities have to be running. Also, the sender doesnt wait for responses because it is asynchronous. UDP is an example.

9.3.4 Receipt-based transient synchronous communication

The acknowledgement sent back from the receiver indicates that the message has been received by the other end. The receiver might be working on some other process.

9.3.5 Delivery-based transient synchronous communication

The acknowledgement comes back to the sender when the other end actually takes control of the message. Asynchronous RPC is an example.

9.3.6 Response-based transient synchronous communication

The sender blocks until the receiver processes the request and sends back a response. RPC is an example. There is no clean mapping of TCP to any type of communication. From an application standpoint it maps to transient asynchronous communication. However, in a protocol standpoint, it maps to a receipt-based transient synchronous communication if it only has a one-size window.

9.4 Message-oriented transient communication

A typical way to write a client-server application is using an abstraction called socket. A socket is a communication end point to which an application can write data that are to be sent out over the underlying network, and from which incoming data can be read. It can use TCP/IP or UDP protocols to make communication, and is a much lower level abstraction of writing program of client and server communication (compared with RPCs). The slides shows an example (Berkeley sockets) of communication using socket with TCP connection.

The server declares a socket and bind the socket with the IP address of the server and a port number. Listen primitive is called to allows the local operating system to reserve enough buffers for a specified maximum number of connections that the caller is willing to accept. A call to accept primitive blocks the caller until a connection request arrives. When a request comes in, the server accepts the request and makes a TCP/IP connection with the client in this case.
The client declares a socket which doesn't need explict binding to a local address, since the operating system can dynamically allocate a port when the connection is set up. It needs the IP address and port number to which a connection request is to be sent. The client is blocked until a connection has been set up successfully, after which both sides can start exchanging information through the send and receive primitives. Finally, closing a connection is symmetric when using sockets, and is established by having both the client and server call the close primitive.
In this way, you have more control of communication process, but you have to deal with all the steps of communication. Besides, in this case, transient communication means both server and client need to be running, once one of them crashed, the connection breaks, and all the messages are discarded.

9.5 Message-Passing Interface (MPI)

Sockets are designed for simple send and receive primitives and one-to-one communication using general-purpose protocol stacks such as TCP/IP. MPI is designed for parallel applications and as such is tailored to transient communication. MPI can be used for communication between clusters of clients and servers, which have intensive communication, and the overhead of TCP/IP is very high for this scenario. MPI also has more advanced primitives of different forms of buffering and synchronization.

- **MPI_bsend**: Transient asynchronous communication is supported by means of the this primitive.
- **MPI_send**: The primitive MPLsend may either block the caller until the specified message has been copied to the MPI runtime system at the sender's side, or until the receiver has initiated a receive operation. (receipt based)
- **MPI_ssend**: Synchronous communication by which the sender blocks until its request is accepted for further processing is available through the MPI_ssend primitive. (delivery based)
- **MPL_sendrecv**: When a sender calls MPL_sendrecv, it sends a request to the receiver and blocks until the latter returns a reply, which provides the strongest form of synchronous communication. Basically, this primitive corresponds to a normal RPC. (reply based)
- **MPL_isend**: A sender passes a pointer to the message after which the MPI runtime system takes care of communication. The sender immediately continues. When the messages to be sent is very large, memory copy overhead can be substantial. So it sends a pointer to a local buffer instead of the whole copy of the data.
- **MPL_issend**: Variant of MPL_ssend with sending pointer instead of a copy.
- **MPL_recv**: The operation MPLrecv is called to receive a message; it blocks the caller until a message arrives.
- **MPL_irecv**: It's an asynchronous variant of MPL_recv, by which a receiver indicates that is prepared to accept a message. The receiver can check whether or not a message has indeed arrived, or block until one does.