Distributed Systems

Data Communications:
RPC & Group Communications
Building Distributed Systems

- To enable a distributed system, one processor must communicate with another processor
- We can do this by building our own communication programs using sockets
- With sockets we are responsible for:
  - Encoding all data
  - Having the client locate the server
  - Byte ordering
  - Client and server processes must agree on the format of exchanged buffers
  - Client and server processes must agree on handshaking for exchange of data
  - Server must include all support code for handling multiple clients concurrently
- Summary: With sockets the programmer is intimately familiar with the details of communications programming
Remote Procedure Calls

• In 1984, Birrell and Nelson devised a new way to allow distributed programs to communicate with each other
• They developed the Remote Procedure Call (RPC)
• RPC’s look like traditional procedure (function) calls from the perspective of the programmer
• RPC’s abstract data communications issues and allow a process on one machine to invoke a procedure on another machine
• RPC’s are usually supported by tools that generate all of the communications code that is used by both the client and server programs
How a Local Procedure Call Works

int add(int a, int b) {
    return a+b;
}
void main() {
    printf("%d\n",add(5,4));
}

• To make a local procedure call (LPC):
  – Parameters are put on the stack
  – A call is made that jumps (sets the PC) to the address of the LPC
  – The local procedure accepts the parameters from the stack
  – The local procedure performs its operation and loads any return value into a register
  – The original calling procedure obtains the return value from a register
  – The original calling procedure restores the system stack
Parameter Passing
Conventions

- **Call by value:**
  - The parameter’s value is passed from the calling to the called procedure.
  - The called procedure works directly with the value passed by the calling procedure.

- **Call by reference:**
  - The address of the parameter is passed from the calling to the called procedure.
  - The called procedure may alter the value of the calling procedures variable because it holds the address of the parameter.

- **Call by copy/restore:**
  - The parameters value is passed from the calling to the called procedure.
  - When the call finishes, the value of the parameter is copied back overwriting the calling procedures original value.
  - Not supported by C or C++.
Remote Procedure Calls

• Desire is to enable a remote procedure call to support call by reference and call by value conventions

• Problem: traditional language syntax is not descriptive enough to describe if the call is by value or by reference

• Problem: how do we handle call by reference in RPC’s where the address space of the called procedure is different from that of the calling procedure
  – Passing a pointer will not work
  – If we use a pointer, how much storage are we pointing to?

• Result: We need additional knowledge in RPC’s to properly support parameter exchange between the calling and called procedures
RPC Architecture

- Client stub marshals the parameters that are passed from the client program to the server-based remote procedure
- Client stub implements communications code to send arguments to the server stub
- Server stub unmarshals the client parameters
- Server stub calls the RPC
- Server stub passes results back to the client code via the client stub
Performing an RPC

• The key to performing a successful RPC is the marshaling and demarshaling code that is performed by the client and server stubs

• Marshaling handles
  – Encoding and decoding of the procedure parameters
  – Marshaling code must know how to encode all standard data types
  – User must be able to augment the basic marshaling facility to instruct the stubs on how to properly handle user-defined types

• The marshaling code must know how to properly handle pointers
  – How much storage do we have to exchange between the client and servers address space
IDL: The Key to Enabling RPC’s

• IDL is an interface definition language
• Includes interface code only, no implementation code is included in the IDL file
• Each argument in an IDL file is augmented with special tags that indicate:
  – Direction: in, out, or in/out
    • Used to optimize communication code
  – Size of buffers, character arrays
  – Is the argument a string
  – Data types and user defined data types

Sample IDL File:

```plaintext
interface test {
  int add(in int a, in int b);
}
```
Building an RPC Program

- IDL
- IDL Compiler
  - Client Code
  - Client Stub
  - Server Stub
  - RPC Implementation
    - Compiler & Linker
      - Client Application
    - Compiler & Linker
      - Server Application
Static or Dynamic Binding

• Static Binding
  – Client specifies the location (IP Address) of the server application during the IDL compilation phase
  – When the client runs it attempts to communicate with the server based on the provided address

• Dynamic Binding
  – Server registers its presence with a name server
  – Client contacts the name server for the location of the server process
  – Client builds and uses a binding handle to communicate with the server
  – Programmer must create the binding handle
    • Typically assisted by the RPC environment API’s
Handling Errors

• RPC’s are more error prone than LPC’s
  – Due to the reliability of the network
• Problems that must be addressed:
  – The client is unable to locate the server
  – The request message from the client to the server is lost
  – The reply message from the server to the client is lost
  – The server crashes after receiving a request
  – The client crashes after sending a request
• Handling these errors properly is essential
Client Can Not Locate The Server

- RPC’s are not as reliable as LPC’s because the client may not be able to locate the server
- Server may be down
- How is the best way to handle this situation considering that the RPC is designed to look like an LPC
- LPC only able to return a single value, how do we notify the caller that the server is down or that we are unable to establish a connection with the server?
- Typically this is handled using exceptions

```java
try {
    int x = add(6,7); //rpc call
} catch (RPCException e) {
    //handler code goes here
}
```
Lost Request Messages

• What happens if the request message is lost during transmission from the client to the server
• Typically this is handled by the communications subsystem of the operating system
• Protocols such as TCP/IP guarantee delivery if the server is available
• Within the TCP/IP protocol stack, device drivers use timers to wait for acknowledgements
• If an acknowledgement is not received within a specified period of time, the message is assumed lost and resent
• TCP/IP protocol stack will return a “server not found” error if retrying does not work
Lost Reply Messages

- Lost replies are more difficult to handle than lost requests
- Client does not know why a reply was not received
- Possibly the server is performing a long running operation
  - Timeouts are typically unpractical
- It may not be safe to retry the request if no reply is received
- Not all operations are safe to be retried
  - Example: Insert a row into a database, debit a savings account
- Idempotent request: An idempotent request is one that does not have any unwanted side-effects
  - Idempotent request may be resent without causing any errors
Lost Reply Messages

- If a RPC does not have the idempotent property then something else has to be done
- One way to handle is to have the server recognize duplicate requests and only process the initial request
- Client may need to identify additional requests (retransmissions) from original requests to help out the RPC server
- Client may also use sequence numbers to help the server distinguish original requests from retransmissions
Server Crashes

- If the server crashes the client does not know if the operation has been performed
  - Idempotent issues
- Client may not detect that the server crashed for a while
  - May use “heartbeat” messages to check on the health of servers
- Failure scenarios
  - Different types of failures must be handled appropriately
  - Client does not know if server actually executed the request
Ways to Handle Server Crashes

- At least once semantics: Client retries the request once the server reboots. RPC has been carried out at least once, but possibly more than once.

- At most once semantics: Client detects the server crashes and does not retry. Result: Operation has been carried out at most one time, but perhaps none at all.

- Exactly once semantics: Client request is guaranteed to be executed exactly one time. Typically this is not possible unless reliable (persistent) message queues are used
  - IBM MQ Series
Client Crashes

• Client sends a request to a server, the server performs the operation on the clients behalf, and the client crashes before it can process the response from the server
• Situation results in an orphan RPC call
• Potential solutions:
  – Client logs all RPC request so that they can be handled after a reboot
  – After a reboot a client broadcasts that it is online so that servers who have orphaned clients can respond
  – When a server detects an orphan it waits a specified period of time (for the client to reboot) and then sends a reply
• All of the above scenarios require the client to be able to respond to an out of scope request
Communication Techniques

• Sometime the clients and servers in an RPC environment need to exchange a lot of data

• Stop and wait protocol: Client breaks down its request into packets, sends each packet, and waits for an acknowledgement for each packet that is sent
  – Advantages: If a packet is lost only the lost packet needs to be resent
  – Disadvantages: Performance -> each packet must be acknowledged

• Blast protocol: Client sends all of its packets and only waits for an acknowledgement for all of its packets
  – Advantages: Performance -> only one acknowledgement is needed for all packets
  – Disadvantages: A lost packet requires all packets to be resent
Flow Control

• The RPC stubs, along with the communication device drivers must properly manage flow control
• Network interface cards can often not send and receive information over a network as fast as the information is generated by the processor
• Network interface cards have buffers to provide some relief
• However, often flow control techniques are needed to govern the amount of information being sent by one processor and received by another processor (and vice versa)
• If not carefully managed, overrun errors can occur because the buffers on the network cards overflow
RPC Issues

• RPC’s may be quite slow because excessive coping of information is necessary
  – On the sending end, parameters are copied from user space to kernel space, and then recopied from the kernel space to the network interface card buffers
  – On the receiving end, the network buffers must be copied into kernel buffers and then into the user address space

• Timer management
  – Both the client and servers use timers (usually through the communication device drivers) to ensure proper message delivery
  – Managing and keeping track of timers for each message send over the network is complex
RPC Applications

• RPC’s are used to build distributed programs

• One of the most popular RPC environments is DCE/RPC
  – See www.osf.org for details

• RPC’s are used to build distributed operating system features

• NFS is based on RPC’s