CISC 434: Communication - II

- Layered Protocols
- Types of Communications
- Remote Procedure Call (RPC)
  - Conventional procedure call to RPC
- Message Oriented Communication
  - Message Oriented Transient Communication
  - Message Oriented Persistent Communication
- Stream-Oriented Communication
- Multicast Communication
Remote Procedure Call

- To call a procedure in a different machine
- To make a remote procedure call look as much as possible like a local one
- To achieve access transparency which may not be possible through explicit exchange of messages between processes
Conventional Procedure Call

- count = read(fd, buf, nbytes) /* C Statement */
  a) Parameter passing in a local procedure call: the stack before the call to read
  b) The stack while the called procedure is active
  - Call by value, call by reference
Client and Server Stubs

- **Client Stub** - Takes its parameters, packs them into a message (marshalling), and sends them to the server stub.
- **Server Stub** - Transforms requests coming in over the network into local procedure calls.
- **Basic Principle of RPC between a client and server program**
Steps of a Remote Procedure Call

1. Client procedure calls client stub in normal way
2. Client stub builds message, calls local OS
3. Client’s OS sends message to remote OS
4. Remote OS gives message to server stub
5. Server stub unpacks parameters, calls server
6. Server does work, returns result to the stub
7. Server stub packs it in message, calls local OS
8. Server’s OS sends message to client’s OS
9. Client’s OS gives message to client stub
10. Stub unpacks result, returns to client
Steps for Remote Computation through RPC

1. Client call to procedure

2. Stub builds message

3. Message is sent across the network

4. Server OS hands message to server stub

5. Stub unpacks message

6. Stub makes local call to "add"
Parameter Passing Issues

- **Passing as value**
  - Representation of data structures such as integers, characters
  - Order in numbering bytes

- **An Example Problem: Pentium → SPARC**
  - Intel Pentium number their bytes from right to left (little endian), while Sun SPARC (big endian) numbers in the opposite way
An Example Problem – contd.

<table>
<thead>
<tr>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>5</td>
</tr>
<tr>
<td>7</td>
<td>6</td>
<td>5</td>
<td>4</td>
</tr>
<tr>
<td>L</td>
<td>L</td>
<td>I</td>
<td>J</td>
</tr>
</tbody>
</table>
(a)

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>5</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
</tr>
<tr>
<td>J</td>
<td>I</td>
<td>L</td>
<td>L</td>
</tr>
</tbody>
</table>
(b)

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>5</td>
</tr>
<tr>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
</tr>
<tr>
<td>L</td>
<td>L</td>
<td>I</td>
<td>J</td>
</tr>
</tbody>
</table>
(c)

a) Original message on the Pentium
b) The message after receipt on the SPARC
c) The message after being inverted.

The numbers in dotted boxes indicate the address of each byte
Parameter Passing Issues

- **Passing as reference** — e.g., passing an array by reference
  - Copy the array into a message and send it to the server
  - The server stub then call the server with a pointer to this array
  - If the server makes any change using the pointer, that changes are reflected in the buffer present in the server stub
  - When the server finishes, the message is sent back to the client stub which is then sent to the client
  - How can you make the above technique twice as efficient?
Parameter Specification and Stub Generation

Both client & server should agree on the following

- Define message format
- Define the representations of data structures
- Make agreement on the message exchange protocol
- Define the interface using IDL

An Example: Both client and server should know the message format shown

```c
foobar( char x; float y; int z[5] )
{
    ....
}
```

(a)
Extended RPC - Asynchronous RPC

Different Forms of Synchronization

- In traditional RPCs, the client and server synchronize – the client is blocked until it receives the service it requested from the server – Synchronous RPC
- Client immediately continues after sending the request – one way asynchronous RPC
- Client waits for the acknowledgment of the request it made to the server, and continues immediately after it has received the acknowledgment
- Combining two asynchronous RPCs – deferred synchronous RPC
Asynchronous RPC – contd.

a) The interconnection between client and server in a traditional RPC
b) The interaction using asynchronous RPC
Asynchronous RPC – contd.

- A client and server interacting through two asynchronous RPCs
Message Oriented Transient Communication

- Message Oriented Communication
  - Message Oriented Transient Communication
  - Message Oriented Persistent Communication

- Example Message Oriented Transient Communication – Sockets
  - A communication end point for an application to write (send) and read (receive) data to/from the underlying network
  - An abstraction over the actual communication end point that is used by the local OS for a specific transport protocol
# Berkeley Sockets Primitives for TCP/IP

<table>
<thead>
<tr>
<th>Primitive</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>Socket</td>
<td>Create a new communication endpoint</td>
</tr>
<tr>
<td>Bind</td>
<td>Attach a local address to a socket</td>
</tr>
<tr>
<td>Listen</td>
<td>Announce willingness to accept connections, used only for connection oriented communication. It is a non-blocking call that allows the local OS to reserve enough buffers for a specified maximum number of connections that the caller is willing to accept.</td>
</tr>
<tr>
<td>Accept</td>
<td>Block caller until a connection request arrives. When such a request arrives, the OS creates a new socket with the same properties as the original one.</td>
</tr>
<tr>
<td>Connect</td>
<td>Actively attempt to establish a connection</td>
</tr>
<tr>
<td>Send</td>
<td>Send some data over the connection</td>
</tr>
<tr>
<td>Receive</td>
<td>Receive some data over the connection</td>
</tr>
<tr>
<td>Close</td>
<td>Release the connection</td>
</tr>
</tbody>
</table>
Berkeley Sockets – contd.

- Connection-oriented communication pattern using sockets
Disadvantages of Sockets

- Abstraction level supports only simple send and receive primitives
- Designed for general purpose protocol stacks such as TCP/IP
- Not suitable for high-speed interconnection networks/high performance server clusters – they require an interface with more advanced features such as buffering and synchronization – Message Passing Interface
The Message Passing Interface

**MPI**

- H/W & platform independent standard for message passing
- Directly uses the underlying networks – no communication servers
- Assumes that communications take place within known group of processes
- Each process is identified by an ID (groupId, processId) to send a message
- Assumes no recovery from serious failures such as process crashes
Some intuitive message-passing primitives of MPI

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

Remote Procedure Call (RPC)
- Conventional procedure call
- Basic steps in an RPC between a client and a server
- Issues in parameter passing in RPCs
- Extended RPCs

Message Oriented Communication
- Message Oriented Transient Communication
  - Berkley Sockets
  - The Message Passing Interface