CIS 630 - Fall 2005
Distributed Systems

Lecture 3
Interprocess Communication

University of Oregon
Department of Computer and Information Science
Business and Logistics

- Guest lecturer
  - Dan Keith
- Research paper
  - First paper available
- Term project
  - Send survey if not already done so
  - Send team member preferences
Lecture Objectives

- General characteristics of interprocess communication
  - Datagram and stream communication
  - Internet protocols
- Look at Java applications that use Internet protocols
- Design issues for request-reply protocols
- Representation of objects and collection of objects
  - Data representation
  - Serialization
- Multicast communication
- Group communication
Middleware Layer Focus

- Two parts
  - Communication protocols from programmer’s view
  - Integration into a programming paradigm
- Transport-level protocols (UDP, TCP) in Chapter 3

* Graphics from Distributed Systems: Concepts and Design, Coulouris, Dollimore, and Kindberg
Interprocess Communication

- Characteristics of protocols for communication between processes on different computer systems
- Types of internet communication
  - Datagrams - individuals packets of data
  - Streams - sequenced communication data
- Protocols specify data representation
- Protocols specify references to remote objects
- Protocols to support two communication patterns
  - Client-server
  - Group communication
API for Internet Protocols

- Internet transport-level protocols
  - UDP: Unified Datagram Protocol
  - TCP: Transport Control Protocols
- Message passing between pair of processes
  - Same or different machines
- Two message communication operations
  - send and receive
  - Defined in terms of destinations and messages
  - Messages sent by one process as a string of bytes to a destination which receives messages byte-by-byte
  - Communication and synchronization
Message Communication Types

- Queues associated with message source/destination
  - Output queue (source) and input queue (destination)
  - *Synchronous* communication
    - synchronization on every message
    - Send process blocks until corresponding receive issued
    - Receive process blocks until message arrives
  - *Asynchronous* communication
    - send operation is *non-blocking*
    - receive operation can be *blocking* or *non-blocking*
      - non-blocking must poll or interrupt on buffer receive
      - blocking with threads has non-blocking advantages
Message Destinations

- Messages sent to \((\text{Internet address}, \text{local port})\) pairs
- Local \textit{port} is a message destination within a computer
  - One receiver, but can have many senders
  - Multiple ports can be used by processes
  - Any process knowing port number can send to it
- Servers publicize their port numbers for use by clients
  - If use fixed Internet address, service must always run on same computer
  - Location transparency
    - \textit{name server} translates to server Internet locations
    - OS supports location independent identifiers (Mach)
Sockets

- UDP and TCP use socket abstraction
  - Endpoint for communication between processes
  - Originated from BSD Unix and in most Unix versions
- Interprocess communication transmits a message from one socket to another
  - To receive, socket must be bound to a local port
  - Only process whose socket is associated with the Internet address and port number can receive
  - Can send and receive on a single socket
- Computers provide a large number of port numbers
- A process may use multiple port numbers
Sockets and Ports

- Each socket is associated with a particular protocol
  - UDP or TCP
- $2^{16}$ possible port numbers for local processes
- No port sharing except in the possible case of multicast

Internet address = 138.37.94.248

Internet address = 138.37.88.249

* Graphics from Distributed Systems: Concepts and Design, Coulouris, Dollimore, and Kindberg
UDP Datagram Communication

- Message sent without acknowledgement or retry
- One process sends and another receives
- Sockets must be first created
- IP protocol allows message sizes up to $2^{16}$ bytes
- Non-blocking *sends* and blocking *receives*
- Timeouts can be set on sockets on receive (why?)
- Invocation of receive gets a message from any origin
  - Internet address and local port of sender returned
- Failure model
  - Dropped messages and out of order messages
Use of UDP

☐ May be acceptable to use a service that is inherently unreliable and does not maintain data packet ordering

☐ *DNS (Domain Naming Service)* uses UDP

☐ Avoids main overheads with guaranteed messaging
  - Need to store state information at source / destination
  - Transmission of extra messages and header data
  - Longer latencies for sender (why?)
import java.net.*;
import java.io.*;
public class UDPClient{
    public static void main(String args[])
    {
        // args give message contents and server hostname
        DatagramSocket aSocket = null;
        try {
            aSocket = new DatagramSocket();
            byte [] m = args[0].getBytes();
            InetAddress aHost = InetAddress.getByName(args[1]);
            int serverPort = 6789;
            DatagramPacket request = new DatagramPacket(m, args[0].length(), aHost, serverPort);
            aSocket.send(request);
            byte[] buffer = new byte[1000];
            DatagramPacket reply = new DatagramPacket(buffer, buffer.length);
            aSocket.receive(reply);
            System.out.println("Reply: " + new String(reply.getData()));
        } catch (SocketException e){System.out.println("Socket: " + e.getMessage());
        } catch (IOException e){System.out.println("IO: " + e.getMessage());
        } finally {if(aSocket != null) aSocket.close();}
    }
}

* Graphics from Distributed Systems: Concepts and Design, Coulouris, Dollimore, and Kindberg
import java.net.*;
import java.io.*;
public class UDPServer{
    public static void main(String args[]){
        DatagramSocket aSocket = null;
        try{
            aSocket = new DatagramSocket(6789);
            byte[] buffer = new byte[1000];
            while(true){
                DatagramPacket request = new DatagramPacket(buffer, buffer.length);
                aSocket.receive(request);
                DatagramPacket reply = new DatagramPacket(request.getData(), request.getLength(), request.getAddress(), request.getPort());
                aSocket.send(reply);
            }
        }catch (SocketException e){System.out.println("Socket: "+ e.getMessage());}
        }catch (IOException e) {System.out.println("IO: " + e.getMessage());}
        }finally {if(aSocket != null) aSocket.close();}
    }
}

* Graphics from Distributed Systems: Concepts and Design, Coulouris, Dollimore, and Kindberg
TCP Stream Communication

- Abstraction of byte stream from sender to receiver
- Hides network characteristics with stream abstraction
  - *Message size*: TCP uses enough packets to send data
  - *Lost messages*: TCP uses acknowledgement scheme
  - *Flow control*: TCP attempts to match the speeds of the processes reading and writing to the stream
  - *Message duplication and ordering*: stream of bytes provided by message identifiers and numbering
  - *Message destinations*: TCP establishes a stream connection (*connect, accept, close*)
- Sockets have *input* stream and *output* stream
Issues Related to Stream Communication

- Matching of data items (remember it’s a byte stream)
  - Need to agree as to the contents of data transmitted
  - Need to agree on type, format, and size

- Blocking
  - Reading processes may block because of lack of data
  - Writing processes may block because of flow control

- Threads
  - Servers typically create new threads to communicate with new clients
  - Allows server to block without delaying clients
  - Without threads, can test for socket data
TCP Communication

- **Failure model**
  - TCP uses checksums to detect/reject corrupt packets
  - TCP uses sequence numbers to detect/reject duplicates
  - Timeouts and retransmissions to deal with loss
  - Processes notified of broken connections

- **TCP is used for many frequently used services**
  - **HTTP** *(HyperText Transfer Protocol)*
  - **FTP** *(File Transfer Protocol)*
  - **Telnet**
  - **SMTP** *(Simple Mail Transfer Protocol)*
TCP Clients Connects to Server, Sends, Receives

```java
import java.net.*;
import java.io.*;
public class TCPClient {
    public static void main (String args[]) {
        // arguments supply message and hostname of destination
        Socket s = null;
        try{
            int serverPort = 7896;
            s = new Socket(args[1], serverPort);
            DataInputStream in = new DataInputStream( s.getInputStream());
            DataOutputStream out = new DataOutputStream( s.getOutputStream());
            out.writeUTF(args[0]); // UTF is a string encoding see Sn 4.3
            String data = in.readUTF();
            System.out.println("Received: "+ data);
        }catch (UnknownHostException e){ System.out.println("Sock:"+e.getMessage());}
        }catch (EOFException e){System.out.println("EOF:"+e.getMessage());}
        }catch (IOException e){System.out.println("IO:"+e.getMessage());}
        }finally {if(s!=null) try {
            s.close();}
catch (IOException e){System.out.println("close:"+e.getMessage());}
    }
}
```

* Graphics from Distributed Systems: Concepts and Design, Coulouris, Dollimore, and Kindberg
import java.net.*;
import java.io.*;
public class TCPServer {
    public static void main (String args[]) {
        try{
            int serverPort = 7896;
            ServerSocket listenSocket = new ServerSocket(serverPort);
            while(true) {
                Socket clientSocket = listenSocket.accept();
                Connection c = new Connection(clientSocket);
            }
        } catch(IOException e) {System.out.println("Listen :"+e.getMessage());}
    }

    // this figure continues on the next slide

* Graphics from Distributed Systems: Concepts and Design, Coulouris, Dollimore, and Kindberg
class Connection extends Thread {
    DataInputStream in;
    DataOutputStream out;
    Socket clientSocket;
    public Connection (Socket aClientSocket) {
        try {
            clientSocket = aClientSocket;
            in = new DataInputStream( clientSocket.getInputStream());
            out =new DataOutputStream( clientSocket.getOutputStream());
            this.start();
        } catch(IOException e) {System.out.println("Connection:"+e.getMessage());}
    }
    public void run(){
        try {
            // an echo server
            String data = in.readUTF();
            out.writeUTF(data);
        } catch(EOFException e) {System.out.println("EOF:"+e.getMessage());
        } catch(IOException e) {System.out.println("IO:"+e.getMessage());
        } finally{ try {clientSocket.close();}catch (IOException e){/*close failed*/}}
    }
}

* Graphics from Distributed Systems: Concepts and Design, Coulouris, Dollimore, and Kindberg
External Data Representation and Marshalling

- Data structures must be “flattened” to be sent in messages as a sequence of bytes
- There are different types of data and different representations of primitive data types on computers
- Byte ordering variants and coding differences
  - Big-endian: most significant byte comes first
  - Little-endian: most significant byte comes last
- Methods for data exchange
  - Agree on a standard (external) transmission format
  - Transmit in sender’s format with format description, and the recipient converts if necessary
External Data Representation

- To support remote procedure calling (RPC or RMI) parameters and results must be sent (structured items)
- *External data representation* is the agreed standard of data structure and primitive value representation
- *Marshalling* is to assemble collection of data items into form suitable for transmission in a message
- *Unmarshalling* disassembles message into equivalent data collection at the receiver

- Data representation forms
  - Binary : CORBA CDR, Java object serialization
  - ASCII : HTTP
CORBA Common Data Representation (CDR)

- 15 primitive types: `short`, `long`, `float`, `double`, `char`, ...
- Constructed types: `sequence`, `string`, `array`, `struct`, ...
- Range of composite types
- Sender’s ordering: big-endian or little-endian
- Floating point follows IEEE standard
- Constructed types are sequenced together
- Data type information is not given with the data representation in the message
- Sun XDR is another external data representation
  - Used in message communication in Sun NFS
## CORBA CDR for Structured Types

<table>
<thead>
<tr>
<th>Type</th>
<th>Representation</th>
</tr>
</thead>
<tbody>
<tr>
<td>sequence</td>
<td>length (unsigned long) followed by elements in order</td>
</tr>
<tr>
<td>string</td>
<td>length (unsigned long) followed by characters in order (can also</td>
</tr>
<tr>
<td></td>
<td>can have wide characters)</td>
</tr>
<tr>
<td>array</td>
<td>array elements in order (no length specified because it is fixed)</td>
</tr>
<tr>
<td>struct</td>
<td>in the order of declaration of the components</td>
</tr>
<tr>
<td>enumerated</td>
<td>unsigned long (the values are specified by the order declared)</td>
</tr>
<tr>
<td>union</td>
<td>type tag followed by the selected member</td>
</tr>
</tbody>
</table>

* Graphics from Distributed Systems: Concepts and Design, Coulouris, Dollimore, and Kindberg
## Example CORBA CDR Message

The flattened form represents a `Person` struct with value: `{'Smith', 'London', 1934}`

<table>
<thead>
<tr>
<th>index in sequence of bytes</th>
<th>4 bytes</th>
<th>notes on representation</th>
</tr>
</thead>
<tbody>
<tr>
<td>0–3</td>
<td>5</td>
<td>length of string</td>
</tr>
<tr>
<td>4–7</td>
<td>&quot;Smit&quot;</td>
<td>‘Smith’</td>
</tr>
<tr>
<td>8–11</td>
<td>&quot;h___&quot;</td>
<td>length of string</td>
</tr>
<tr>
<td>12–15</td>
<td>6</td>
<td>‘London’</td>
</tr>
<tr>
<td>16–19</td>
<td>&quot;Lond&quot;</td>
<td>unsigned long</td>
</tr>
<tr>
<td>20-23</td>
<td>&quot;on___&quot;</td>
<td></td>
</tr>
<tr>
<td>24–27</td>
<td>1934</td>
<td></td>
</tr>
</tbody>
</table>
Java Object Serialization

- Objects and primitive data values as arguments/results
  - An object is an instance of a Java class
  - Stating that a class implements the `Serializable` interface allows its instances to be serialized

- `Serialization` refers to the activity of flattening an object or a set of objects into a serial form suitable for storing on disk or transmitting in a message

- `Deserialization` is the reverse
  - Process has no prior knowledge of the types
  - Some information about the class of each object is included in the serialized form
Java Object Serialization (continued)

- Java objects can contain references to other objects
- All objects referenced are serialized together
  - Serialized as handles to object within serialized form
  - More efficient if object referenced more than once
- Object serialization process
  - Class information written, with class handles
  - Types and names of instance variables, with class info
  - Primitive types written in portable binary format
  - Strings written in Universal Transfer Format (UTF)
- Java object serialization uses reflection to find out
  class information and to create a class based on it
## Indication of Java Serialized Form

<table>
<thead>
<tr>
<th>Person</th>
<th>8-byte version number</th>
<th>h0</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>int year</td>
<td>java.lang.String</td>
<td>class name, version number</td>
</tr>
<tr>
<td></td>
<td>name:</td>
<td>java.lang.String</td>
<td>number, type and name of instance variables</td>
</tr>
<tr>
<td>1934</td>
<td>5 Smith</td>
<td>6 London</td>
<td>values of instance variables</td>
</tr>
</tbody>
</table>

The true serialized form contains additional type markers; h0 and h1 are handles.
Remote Object References

- A remote object reference is an identifier for a remote object that is valid throughout the distributed system.
- To invoke a method on a remote object, a remote object reference is passed in the invocation message to specify which object to invoke.
- Must be unique over space and time.
  - Represent with Internet address, port number, time, and local object number incremented on each creation.
  - Can use as an address if remote object doesn’t move.
- Also need to include info on the interface of the remote object such that methods can be queried.
Client-Server Communication

- *Request-reply* protocol
  - Request messages are matched with replies
  - Normal case is synchronous because client blocks
  - Reliable because server reply is acknowledgement
  - Asynchronous operation may be used

- Designed to support roles and message exchanges in typical client-server interactions

- Most RMI and RPC systems support similar protocol
Send / Receive Operations in Java API for UDP

- Communication primitives
  - doOperation, getRequest, sendReply
  - Tailored for supporting RMI for remote objects
  - Matches requests to replies
  - If UDP used, deliver guarantees must be provided
  - Reply message acts as an acknowledgement

- Message identifiers used for reliable message delivery
  - Request identifier and sender process identifier
Request-Reply Communication

dooOperation

(wait)

(continuation)

Client

Request
message

Reply
message

Server

getAddress
select object
execute
method
sendReply
Operations of Request-Reply Protocol

public byte[] doOperation (RemoteObjectRef o, int methodId, byte[] arguments)
    Sends a request message to the remote object and returns the reply.
    The arguments specify the remote object, the method to be invoked and the
    arguments of that method.

public byte[] getRequest ();
    Acquires a client request via the server port.

public void sendReply (byte[] reply, InetAddress clientHost, int clientPort);
    Sends the reply message reply to the client at its Internet address and port.
Failure Model of Request-Reply Protocol

- UDP datagram communication can be unreliable
- Protocol can also suffer from failure of processes
  - `doOperation` recovery at sender
    - `Timeouts` with limited message retransmission
    - Can result in message duplication
- Duplicate message discarding at receiver
  - Sequence numbers are important for this purpose
  - Cannot guarantee reply message received
- Suspicion of lost reply message
  - Reply message is resent
  - Idempotent regeneration of result or result history
RPC Exchange Protocols

- Three protocols with differing failure semantics
  - *Request* (R) protocol
    - No value returned to client
    - No confirmation back to client of receipt
  - *Request-reply* (RR) protocol
    - Server reply can be regarded as acknowledgement
    - Subsequent client call regarded as acknowledgement of server’s reply message
  - *Request-reply-acknowledge* (RRA) protocol
    - Acknowledge reply message contains the *requestId*
    - Acknowledges all reply messages with lower *requestIds*
    - More redundancy and helps release reply history
Use of TCP Streams to Implement Request-Reply

- Allows request-reply protocols to not worry about message delivery reliability issues
- TCP ensure reliable, sequenced delivery of request and reply messages of any size with flow control
- Simplifies application protocol implementation
- TCP is more costly than UDP
  - Pay overhead of connection establishment
- Specially tailored protocols implemented over UDP
  - Sun NFS transmits fixed-size file blocks
  - File operations are idempotent and no history required
  - Can achieve reliable performance at less cost
HTTP: A Request-Reply Protocol

- Used for client requests to web servers and replies
- Specifies messages in a request-reply exchange
  - Methods, arguments, results, marshalling
- HTTP is implemented over TCP
  - Connection setup, client send, server reply, close
  - Persistent connections used in HTTP 1.1
    - stays open over a series of request-reply exchanges
    - close connection by either sender or receiver any time
- ASCII text string marshalling with compression
- MIME encoding for multipart data
- Universal Resource Locator (URL)
Group Communication

- Pairwise message exchange is not the best model for communication between processes in a group
- *Multicast* communication is better
  - Sends a single message from one process to each of the members of a group of processes
  - Membership in group is transparent to the sender
- Multicast messages provides a useful infrastructure for constructing distributed systems
Multicast Distributed System Characteristics

- Fault tolerance based on replicated services
  - A replicated service consists of a group of servers
  - Client requests are multicast to the server group

- Finding discovery servers in spontaneous networking
  - Register or look up services in distributed system

- Better performance through replicated data
  - Each time data changes, new values are multicast
  - Processes managing replicas update their values

- Propagation of event notification
  - Multicast to notify group of processes of an event

- See IP multicast in book
Interprocess Communication in Unix

- IPC primitives in BSD 4.x versions of Unix
  - Provided as system calls
  - Implemented as layer over Internet TCP and UDP

- Interprocess communication operations based on socket abstraction
  - Sending and receiving queues
  - Message destinations
    - Specified as *socket addresses* (Internet address, local port number)
  - Any process can create a socket for communication using the *socket() system call*
Socket Operations

- Socket system call arguments
  - Communication domain (usually the Internet)
  - Type (datagram or stream)
  - Protocol (TCP or UDP)
- Socket call returns a descriptor for future reference
- Before a pair of processes can communicate
  - Receiver must bind socket descriptor to socket address
  - Send must bind also if it expects a reply
- Socket address are public in the sense that they can be used as destinations by any process
Sockets used for Datagrams

Sending a message

```
s = socket(AF_INET, SOCK_DGRAM, 0)
bind(s, ClientAddress)
sendto(s, "message", ServerAddress)
```

Receiving a message

```
s = socket(AF_INET, SOCK_DGRAM, 0)
bind(s, ServerAddress)
amount = recvfrom(s, buffer, from)
```

`ServerAddress` and `ClientAddress` are socket addresses
Sockets used for Streams

Requesting a connection

\[
\begin{align*}
  s &= \text{socket(AF_INET, SOCK_STREAM, 0)} \\
  \text{connect}(s, \text{ServerAddress}) \\
  \text{write}(s, "message », length)
\end{align*}
\]

Listening and accepting a connection

\[
\begin{align*}
  s &= \text{socket(AF_INET, SOCK_STREAM, 0)} \\
  \text{bind}(s, \text{ServerAddress}) \\
  \text{listen}(s, 5) \\
  \text{sNew} &= \text{accept}(s, \text{ClientAddress}) \\
  n &= \text{read}(\text{sNew}, \text{buffer}, \text{amount})
\end{align*}
\]

\text{ServerAddress} \text{ and } \text{ClientAddress} \text{ are socket addresses}