CIS 630 - Fall 2010
Distributed Systems

Lecture 3
Interprocess Communication

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

- **Surveys**
  - Almost all surveys are in (thanks!)
  - Will publish general results by tomorrow

- **Term project**
  - See term project description and begin thinking of ideas
  - Send team preferences (if any) this week

- **Term paper**
  - Think of it as a survey of a research topic
  - You do not have to produce new results

- **Book on reserve in library**

- **Programming assignment due Oct. 22**
Research Paper Approach and Structure

- Identify a topic of interest to you
  - Always best if you like what you are researching

- Gather papers that discuss major research themes, results, and projects in the area
  - Newer papers will reference earlier important papers

- Summarize what the papers tell you about the topic
  - Important problems and solutions
  - Major projects and state of the art
  - Outstanding issues and future directions
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
☐ Work through most of Chapter 3
Interaction Model

- How a set of processes communicate to achieve a task
- In distributed systems, these are distributed algorithms
  - Steps taken by each participant
  - Messages transmitted between them to coordinate
- Processes are assumed to have private state
  - State is revealed through messages only
- Important limiting factors
  - Communication performance
    - latency, bandwidth, jitter
  - Lack of global time
    - multiple local clocks, clock drift, synchronization error
Variants in Interaction Model

- Interactions models are distinguished by two classes

- Synchronous distributed systems
  - each step has known lower/upper time bounds
  - messages are received in bounded time
  - each process has local clock with known drift rate

- Asynchronous distributed systems
  - no bounds on process execution speeds
  - no bounds on message transmission delays
  - no bounds on clock drift rates

- Real systems tend to be asynchronous
  - Bounds are difficult to quantify and they can change
Real-Time Ordering of Events

* Graphics from Distributed Systems: Concepts and Design, Coulouris, Dollimore, and Kindberg
Processes and Channels

We typically assume a simplified channel when we want to reason about communication between processes for modeling purposes.

* Graphics from Distributed Systems: Concepts and Design, Coulouris, Dollimore, and Kindberg
Failure Models

- Allow precise definition of failure classes
- Process and messaging failures

<table>
<thead>
<tr>
<th>Class of failure</th>
<th>Affects</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fail-stop</td>
<td>Process</td>
<td>Process halts and remains halted. Other processes may detect state.</td>
</tr>
<tr>
<td>Crash</td>
<td>Process</td>
<td>Process halts and remains halted. Other processes may not detect.</td>
</tr>
<tr>
<td>Omission</td>
<td>Channel</td>
<td>A message inserted in an outgoing buffer never arrives.</td>
</tr>
<tr>
<td>Send omission</td>
<td>Process</td>
<td>A process completes a send, but message not put in buffer.</td>
</tr>
<tr>
<td>Receive omission</td>
<td>Process</td>
<td>A message is put in incoming buffer, but never received</td>
</tr>
<tr>
<td>Arbitrary (Byzantine)</td>
<td>Process or Channel</td>
<td>Process/channel exhibits arbitrary behavior.</td>
</tr>
</tbody>
</table>
## Failure Models

### ☐ Timing failures

<table>
<thead>
<tr>
<th>Class of failure</th>
<th>Affects</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Clock</td>
<td>Process</td>
<td>Process’s local clock exceeds the bounds on its rate of drift from real time.</td>
</tr>
<tr>
<td>Performance</td>
<td>Process</td>
<td>Process exceeds the bounds on the interval between two steps.</td>
</tr>
<tr>
<td>Performance</td>
<td>Channel</td>
<td>A message’s transmission takes longer than the stated bound.</td>
</tr>
</tbody>
</table>
Reliability of One-to-One Communication

- Validity and integrity define reliability characteristics
  - Related both to security and failure

- Validity
  - Any message in an outgoing buffer will eventually arrive at/in a corresponding incoming buffer

- Integrity
  - A message received is identical to the one that was sent without duplication

- Threats to integrity
  - Protocols that retransmit but do no reject duplicates
  - Malicious users who inject, replay, or tamper messages
Networks 101

- A network is composed of
  - Transmission media
  - Hardware devices (switches, routers, interfaces, …)
  - Software components
- Hardware devices and software components are collectively “communications subsystem”
- Devices and computers connected to the network for communication are called *hosts*
- The term *node* refers to any computer or switching device attached to the network
- The internet is a single communications subsystem split into distinct subnetworks acting as routing units
Network Requirements and Types

- Performance
- Scalability
- Reliability
- Security
- Mobility
- Quality of service
- Multicasting
- ...
Network Principles

- Network technologies are built around fundamental principles that are directed to meeting requirements
  - Concepts for building network solutions
- Packet transmission
- Data streaming
- Switching schemes
- Protocols and protocol stacks
- Routing algorithms and hardware
- Congestion control
- Internetworking
- ...
Packet Transmission and Streaming

- How data is moved from one point to another
- Packets
  - Data moves as fixed-length packets containing the content of the message and information relevant for routing and transport protocols
  - Packets are separable transmission units
- Streaming
  - Data flows through the network in a more continuous fashion with guarantees and bounds on performance (e.g., latency)
  - Popular for multimedia applications
Network Switching

- Broadcast
  - Message sent to everyone
  - Usually due to shared network (e.g., Ethernet)
- Circuit switching
  - Path is set up in network and assigned to communication
- Packet switching
  - Store-and-forward packet communication
- Frame relay
  - Think of as circuit switching + packet switching
  - Small packet units minus store-and-forward
Network Protocols

- Set of rules and formats used for communication
  - Specification of sequence of messages to be exchanged
  - Specification of the format of these messages
- Protocols exist for different levels of abstraction
  - Network protocols
  - High-level application-specific protocols
- Network software protocols are multi-layered
Protocol Layers

- Layers are based on abstractions
- Low levels are abstractions above the wire protocol
- Middle levels abstract raw data transmission / reliability
  - Example: TCP, IP, UDP
- Higher levels abstract specific activities
  - Example: FTP, HTTP

* Graphics from Distributed Systems: Concepts and Design, Coulouris, Dollimore, and Kindberg
Network Protocol Stacks and Encapsulation

- Open Systems Interconnection (OSI) stack
  - Standardized by ISO
- Stack abstraction allows different implementations to be switched in/out
  - Separation of concerns
- Stacks could have performance concerns
- Encapsulation
  - Lower levels packages messages for higher levels

* Graphics from Distributed Systems: Concepts and Design, Coulouris, Dollimore, and Kindberg
Protocols for the Internet

- Main internet protocols originated in the 1970s
  - ARPANET project
  - TCP: Transmission Control Protocol
  - IP: Internet Protocol (IPv4, IPv6)

- Example of successful open standards and protocol layering
  - TCP/IP does not specify anything below Internet datagram layer

* Graphics from Distributed Systems: Concepts and Design, Coulouris, Dollimore, and Kindberg
API for Internet Protocols

- Internet transport-level protocols
  - UDP: User Datagram Protocol
  - TCP: Transport Control Protocols
  - Layered about IP (host-to-host protocol)

- UDP
  - Basic and close to IP to incur less overhead than TCP
  - No guarantee of reliable transmission or acknowledgement that messages are received

- TCP
  - Provides connection-oriented reliable transmission
  - Stream-oriented with sequencing and flow control
Interprocess Communication

- Distributed systems require process-to-process (interprocess) communication (IPC)
  - Communication protocols from programmer’s view
  - Integration into a programming paradigm
- IPC (Ch. 4) and RMI (Ch. 5) are about middleware

* Graphics from Distributed Systems: Concepts and Design, Couloris, 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
Message Communication

- Processes communicate by sending messages
- Fundamental building block of distributed systems
- IPC is all about message passing between processes
  - Sender and Receiver
  - 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
Asynchronous Communications

- Different issues are present than in synchronous case
- Consider how synchronous receive works
  - Receiver makes a call to receive and provides a buffer
    `recv(src, &buf, BUFSIZE, ...);`
  - Receiver blocks until receive completes
  - When the receive completes and the receiver unblocks
    - buffer is guaranteed to have the message in it
    - or an error occurred and an error code is returned
- How is asynchronous different?
Asynchronous Communications

- In the asynchronous receive, the receive is posted and the buffer is provided, but the call returns immediately
  - Not guarantee on what the buffer actually contains
  - Buffer will get filled in by the communications subsystem when the message arrives are some point in the future
- What does this require then?
  - Need to take care to check if communication has completed before using the contents of the buffer
- Similar issues on the sending side
  - Need to make sure transmission has occurred and it is safe to reuse the buffer
Asynchronous Communications Advantage

- Distributed systems hardware is providing more support for asynchronous communications.
- The point of asynchronous messaging is to avoid having to block.
  - Do useful work instead.
  - Overlap communications with other work.
- Writing asynchronous applications are more tedious.
- But asynchronous applications can deliver benefits.
- High-performance computing applications can better utilize communication AND computing resources.
  - Asynchronous messaging in MPI.
Message Destinations

- Messages sent to \((\text{Internet address, local port})\) pairs
  - Example: \text{www.cs.uoregon.edu:80}
- 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
  - Agree on port sending data to
- $2^{16}$ possible port numbers for local processes
- No port sharing except in the possible case of multicast
- Many processes allowed to send to same port

Internet address = 138.37.94.248

Internet address = 138.37.88.249

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

On Unix systems we can see these socket/port relationships using the `netstat` command.

```
[dyana-171:CIS-630/Fall-2010/Lecture] malony% netstat
Active Internet connections
Proto Recv-Q Send-Q Local Address      Foreign Address         (state)
tcp4   0      0  dyna6-171.cs.uor.55983  pz-in-f99.1e100..http  ESTABLISHED
tcp4   178934 0  dyna6-171.cs.uor.55234 216.105.44.131.8110  ESTABLISHED
tcp4   0      0  dyna6-171.cs.uor.54255  ix-4.cs.uoregon..ssh  ESTABLISHED
tcp4   0      0  dyna6-171.cs.uor.61774  163-211.static.q.12350  ESTABLISHED
tcp4   0      0  dyna6-171.cs.uor.50149  ucsf-81-34.uncsf..7550  ESTABLISHED
tcp4   0      0  dyna6-171.cs.uor.54648  vitalstatistix.c.micro  ESTABLISHED
```

Internal (NAT) address   Outward facing port   Other side of socket (me ssh’ing to ix)
Finding Hosts by Name

- How do you find hosts on the network to connect to?
- Internet “Domain Name Service” (DNS) resolves symbolic names to the IP addresses assigned to the host

Java: java.net.InetAddress
  - InetAddress.getByName()
    Given a name, return an object representing the IP address
  - InetAddress.getAddress()
    Get the raw byte array for the address

C: not as straightforward
  - struct sockaddr_in addr;
  - addr.sin_addr.s_addr = inet_addr(hostname); …
UDP Datagram Communication

- Now we need to talk to the hosts we found
- UDP messages 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 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?)
UDP Blocking Semantics and Failure Model

- Send blocks until data is safely handed to underlying IP
- Receive blocks until a packets arrives
  - Timeouts can be set to handle delayed or lost datagrams
  - Can use polling to check for arrival
  - Can use multiple threads to split waiting / blocking
- Socket level queues messages arriving before receive
  - Network layer in OS provides queue
  - Messages arriving for a port that no process has open are discarded
- UDP failure model
  - Omission failure: no guarantee of message delivery
  - Ordering: message may not arrive in order sent
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, m.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
Uses of UDP

- Applications where you want lower overhead and occasional omission failures are OK
- Consider Voice over IP (VoIP)
  - A packet represents a small time interval of sound
  - If one is lost, it might not even be noticed
  - Algorithms can compensated at playback
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
- Persistent connection
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
  - Resend limits being exceeded may terminate connection

- TCP is used for many frequently used services (Why?)
  - HTTP (HyperText Transfer Protocol)
  - FTP (File Transfer Protocol)
  - Telnet
  - SMTP (Simple Mail Transfer Protocol)
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());}
        }
    }
}
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());}
    }
}

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

- Sockets are concerned with connections
  - Data is treated as raw bytes with no meaning
- Need to send “structured” data requires
  - Data structures must be “flattened” to be sent in messages as a sequence of bytes
  - Requires agreed upon data representation
- There are different types of data and different representations of primitive data types on computers
- Byte ordering variants and coding differences
  - Big-endian versus Little-endian
- Methods for data exchange
  - Agree on a standard (external) transmission format
- Heterogeneity makes this a challenge
External Data Representation (XDR)

- 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
  - CORBA CDR, Java object serialization, Sun XDR
  - HTTP, XML
Common Features of XDRs

- Platform-neutral representation of primitive types
- Recursive representation of structured types
  - C structs
  - C++ classes, Java classes
  - Union, enumerations, …
- Metadata beyond the type and contents
  - Array lengths, dimensions, …
  - String lengths
## 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 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

<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></td>
<td></td>
<td>‘Smith’</td>
</tr>
<tr>
<td>4–7</td>
<td>&quot;Smit&quot;</td>
<td>length of string</td>
</tr>
<tr>
<td></td>
<td></td>
<td>‘London’</td>
</tr>
<tr>
<td>8–11</td>
<td>&quot;h     &quot;</td>
<td></td>
</tr>
<tr>
<td>12–15</td>
<td>6</td>
<td></td>
</tr>
<tr>
<td>16–19</td>
<td>&quot;Lond&quot;</td>
<td></td>
</tr>
<tr>
<td>20-23</td>
<td>&quot;on    &quot;</td>
<td></td>
</tr>
<tr>
<td>24–27</td>
<td>1934</td>
<td>unsigned long</td>
</tr>
</tbody>
</table>

The flattened form represents a Person struct with value: {‘Smith’, ‘London’, 1934}
Java Object Serialization

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

- **Deserialization** is the reverse process:
  - Process has no prior knowledge of the types.
  - Some information about the class of each object is included in the serialized form.

- 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.
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 at runtime and to create a class based on it
- This is how the serialization system can automatically scan through an object and determine types and names
## Indication of Java Serialized Form

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

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

*Serialized values*
Considerations of XDRs

- **Pro**
  - Standardized external representations eliminate a significant hurdle to heterogeneous systems

- **Con**
  - **Performance**
    - data must be encoded and decoded
    - this takes time
  - **Lack of a single standard**
    - there are several and several are used
    - limits interoperation between distributed systems built using different middleware packages
Remote Object References

- Systems like CORBA or Java allow a process of a distributed program to refer to objects that are actually stored in the memory of another process.
- 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

- Given an object instance, what can we do with it?
  - Look at it’s data
  - Invoke methods on it (*remote method invocation*)

- How do we make this happen?

  - *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
Request-Reply Protocol

- Protocol defines the set of messages passed back and forth from client (caller) to serve (callee)

- Communication primitives
  - doOperation: used by the client to invoke remote operations given a remote object reference
  - getRequest: used by the server to retrieve request submitted by clients and execute them
  - sendReply: used by the server to respond to the request with the reply, possibly containing return value; client unblocks when reply received

- Message identifiers used for reliable message delivery
  - Request identifier and sender process identifier
  - Matches requests to replies
Request-Reply Communication

Client

- doOperation
- (wait)
- (continuation)

Server

- getRequest
- 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.
Consideration: Over TCP or UDP?

- Requests are followed by replies
  - Reply acts as an acknowledgement
  - TCP ACKs are redundant
- Establishing a connection requires message exchange in addition to the request/reply pair
  - Wasteful extra communication overhead with TCP
- Majority of RPC calls pass few and small arguments and return values
  - Flow control largely unnecessary
- Is Request-Reply for RMI ok for UDP?
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 ensures 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

\[
\text{s} = \text{socket(AF_INET, SOCK_DGRAM, 0)} \\
\quad \text{bind(s, ClientAddress)} \\
\quad \text{sendto(s, "message", ServerAddress)}
\]

Receiving a message

\[
\text{s} = \text{socket(AF_INET, SOCK_DGRAM, 0)} \\
\quad \text{bind(s, ServerAddress)} \\
\quad \text{amount = recvfrom(s, buffer, from)}
\]

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

Requesting a connection

```c
s = socket(AF_INET, SOCK_STREAM,0)
connect(s, ServerAddress)
write(s, "message", length)
```

Listening and accepting a connection

```c
s = socket(AF_INET, SOCK_STREAM,0)
binding(s, ServerAddress);
listen(s,5);
sNew = accept(s, ClientAddress);
```

```
n = read(sNew, buffer, amount)
```

ServerAddress and ClientAddress are socket addresses