Interprocess Communication (IPC)

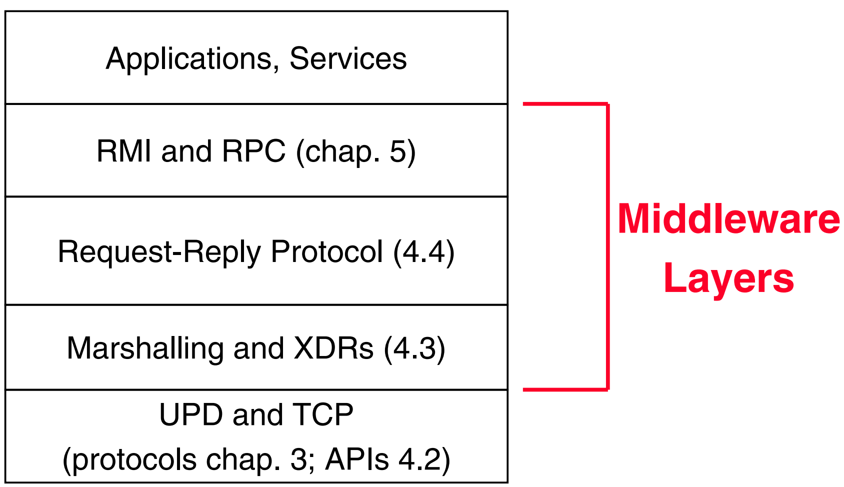
Prof. Dave Bakken

Cpt. S 464/564 Lecture
Textbook, Chapter 4
Sept 25+27, 2000

Administrative Items

- Handouts today
  - None: slides are online!
  - Also “Request-Reply-Implementation” is there too!
- Section 4.6 of text is not required reading or testable
  - May be useful or interesting for some to read…
- **How is Project 1 going?**

Middleware Layers and IPC



Outline of Lectures

- **<u>API for Internet protocols (4.2)</u>**
  - **Characteristics of IPC**
  - **Sockets**
  - **UDP datagram communication**
  - **TCP stream communication**
- Marshalling and external data representations (4.3)
- Client-server communication (4.4)
- Group communication (4.5)

Characterizations of IPC

- Message passing can be supported with two primitives: send and receive
  - Defined in terms of destinations and messages
  - One process sends a particular message to a destination
  - That destination then receives that message
  - This may also involve synchronization
  - Note: "sender" == "sending process" & "receiver" == "receiving process"
- Each message destination has a queue of messages
  - Sending processes cause messages to be enqueued
  - Receiving processes cause messages to be dequeued
- Communication can be synchronous or asynchronous…

Synchronous and Asynchronous IPC

- Synchronous IPC: senders and receivers synchronize at every message
- Both send and receive are blocking operations
  - Sender blocks until the corresponding receive is issued
  - Receiver blocks until the corresponding send is issued
- Asynchronous IPC: send is non-blocking
  - Sending process can continue once message has been copied
- Receive in asynch. IPC can block or not block
  - Blocking: similar to synchronous IPC for receiver
  - Non-blocking: receiver continues while its buffer is filled in the background
  - Then receiver must poll or get callback
  - Nonblocking receive not frequently supported

Message Destinations

- A message has to be delivered to a destination
- In Internet protocols: < IP Address, local port>
  - Port has exactly one receiver
  - Port can have multiple senders
  - Processes may use multiple ports for receiving messages
    - Why?
    - Examples?
  - Any process that knows the port can send a message to it
  - Servers publicize their port numbers, or use established "well known" ones
- Fixed IP Address ➔ service must always run on that host
  - Avoid by using a name and mapping it onto the address with a name server or binder (chapter 9)
  - Distributed OS (e.g. Mach) provides location-independent communication IDs
- Alternative to ports: destination is a process ID (PID; e.g. Chorus)
  - Ports let a process have multiple points of entry with IPC
- Some systems have groups of ports or processes (on different hosts) as destinations

Reliability and Ordering of IPC

- Reliability of point-to-point communication:
  - Validity: any message in the sender's outgoing message buffer is eventually delivered to the incoming message buffer of the receiver
  - Integrity: the message received is identical to the one sent, and none delivered twice
- Reliability: a point-to-point IPC service is
  - Deemed reliable if messages are guaranteed to be delivered despite a "reasonable" number of packets being dropped or lost
  - Deemed unreliable if messages are not guaranteed to be delivered even with only a single packet being dropped or lost
  - Deemed to have integrity: messages arrive uncorrupted and without duplication
- Sender order: the order in which the sender transmits messages (calls the send primitive)
  - Some applications require a receiver to be deliver messages in sender order
  - Failure to do this is regarded as a failure by these apps

Sockets

- Socket abstraction: an endpoint for communication between processes
- Receiver must have the socket bound to a local port and one of its computers IP addresses
  - Only that process can receive messages sent to that port
  - Many senders can send to a port
  - Same port may be used by a process for both sending and receiving
  - A given port may be used with TCP or UDP, not both
  - Ports are 16 bit numbers in TCP and UDP
- Java class *InetAddress* represents and IP address
- Constructor can be given a DNS hostname:

  InetAddress aComputer = InetAddress.getByName("bakken.eecs.wsu.edu")

  Can throw *UnknownHostException*
- Note: interface for this class does not depend on #bytes in an IP address
  - IPv4: 4 bytes
  - IPv6: 16 bytes

UDP Datagram Communication

- UDP sends a datagram without
  - Acknowledgements
  - Retries
  - …. So if any failure happens, it may not be delivered
- Senders and receivers must
  - Create a socket bound to <IP Address, local port>
  - Server: binds to a specific server port, well-known or made known to clients
  - Client: binds to any free local port
- Receive returns the message, *plus* the <IP Address, local port> of sender

Some UDP Datagram Issues

- Message size
  - Receiver must specify array of bytes of a certain size
  - Too small: truncated (oops…)
  - IP allows messages of up to 2^{16} bytes (header+message)
  - Most systems impose much smaller max (8K typical)
  - Apps requiring larger must fragment and reassemble
- Blocking
  - UPD supports non-blocking *send* and blocking *receive*
  - *Receive* can specify a timeout
  - If receiver has other work to do while waiting, it creates a thread
- Timeouts
  - Can't always wait forever… can set a timeout on a socket
- Receive from any
  - UDP *receive* does not specify message origin

UDP Failure Model and Uses

- UDP Failure model
  - Omission failures: can have send-omission or receive-omission failures
  - Ordering failures: not in sender order
- Can provide a reliable delivery service on top of UDP with
  - ACKs
  - Sequence numbers
- UDP's failure model is quite acceptable for some applications
- UDP does not suffer from overheads of guaranteed message delivery:
  - Storing state at sender and receiver
  - Transmission of extra messages to provide the guarantees
  - Extra latency for sender

Java API for UDP datagrams

- Two main classes: *DatagramPacket* and *DatagramSocket*
- *DatagramPacket*: constructor (ctor) for sending takes
  - Array of bytes storing the message
  - Length of message
  - IP address
  - Port
- *DatagramPacket* ctor for receiving
  - Array
  - Length
- *DatagramPacket* mainly used as a parameter to *DatagramSocket*

Java API for UDP datagrams (cont.)

- *DatagramSocket*: supports sending and receiving UDP datagrams
  - One-parameter ctor: port to use
  - No-parameter ctor: uses any free port
- *DatagramSocket* methods
  - *send (filled_packet_to_send)*
  - *receive(empty_packet_to_fill*
  - *setSoTimeout*: set timeout for receive
  - *connect:* connect to a particular <IP Address, port>
    - Can only send and receive messages to/from there

Java UDP Client

```
import java.net.*;
import java.io.*;
public class UDPClient {
  public static void main(String args[]){
    // args give message contents and destination hostname
    try {
        DatagramSocket 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()));
        aSocket.close();
    } catch (SocketException e){System.out.println("Socket: " + e.getMessage());
    } catch (IOException e){System.out.println("IO: " + e.getMessage());
  }
}
```


Java UDP Server (echo server)

```
import java.net.*;
import java.io.*;
public class UDPServer{
  public static void main(String args[]){
    try{
        DatagramSocket 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());}
  }
}
```

TCP Stream Communicaiton

- API to TCP provides abstraction of a reliable stream of bytes that can be written to and read from
- Many network characteristics hidden by this abstraction
  - Message size
    - Application chooses size it likes
    - TCP implementation generally chooses how long to wait to send an IP packet
  - Lost messages: ACK and resending compensates
  - Flow control: Slow down a fast sender to match slower receiver
  - Message duplication and reordering
  - Message destinations: once stream established, just use it, not addresses

TCP Stream Communication (cont.)

- Connection setup is very expensive (compared to sending a few bytes)
- Stream API assumes, for connection setup, one end is client, other is server
  - After setup, both are peers
- Client connection setup
  - Create stream socket bound to any port
  - Make a *connect* call asking for a connection to an <IP Addr, port>
- Server connection setup
  - Creates a *listening* socket (bound to a port) to wait for connection requests
  - Call *accept* for the connect, which creates a new stream for that interaction
- Each stream socket has an input stream and an output stream
- Closing a socket: won't read/write any more to it
  - Buffered data are flushed
  - Further uses by other side results in EOF indication

TCP Stream Issues, Failure Model, Uses

- Issues
  - Matching of data items: sender and receiver have to agree on a data protocol for their interaction (one int, two doubles, …)
  - Blocking: receiver blocks if no data, sender blocks if receiver too far behind
  - Threading: servers that accept a connection usually create a thread for it
    - Select call in Unix lets one block until data comes on one of many streams
- Failure model: guarantees even with some problems
  - Checksums "guarantee" integrity
  - Sequence numbers reject duplicate packets
  - Timeouts and retransmissions mask lost packets
  - But does not guarantee in the face of all possible common problems
- Uses of TCP: HTTP, FTP, Telnet, SMTP, …

Java API for TCP Streams

- Main classes: *ServerSocket and Socket*
- *ServerSocket*
  - For listening to *connect* requests from clients
  - *ServerSocket.accept* gets a connect request from the queue (blocking)
  - Results in one *Socket* instance
- *Socket*
  - For use by a pair of processes
  - Client constructs with DNS name and port, then calls *connect*
  - *Socket.getInputStream* returns an instance of *InuptStream*; same for Output

Java TCP Client

```
import java.net.*;
import java.io.*;
public class TCPClient {
   public static void main (String args[]) {
      // arguments supply message and hostname
      try {
          int serverPort = 7896;
          Socket s = new Socket(args[1], serverPort);
          DataInputStream in = new DataInputStream( s.getInputStream());
          DataOutputStream out = new DataOutputStream( s.getOutputStream());
          out.writeUTF(args[0]);      // write the line of data to the stream
          String data = in.readUTF();                    // read a line of data from the stream
          System.out.println("Received: "+ data) ;
          s.close();
      } catch (UnknownHostException e){System.out.println("Socket:"+e.getMessage());
      } catch (EOFException e){System.out.println("EOF:"+e.getMessage());
      } catch (IOException e){System.out.println("readline:"+e.getMessage());}
   }
}
```


Java TCP Server

```
import java.net.*;
import java.io.*;
public class TCPServer {
   public static void main (String args[]) {
      try {
          int serverPort = 7896; // the server port
          ServerSocket listenSocket = new ServerSocket(serverPort);
          while (true) {
             Socket clientSocket = listenSocket.accept();
             Connection c = new Connection(clientSocket);
          }
      } catch (IOException e) {System.out.println("Listen socket:"+e.getMessage());}
   }
}
class Connection extends Thread {
          DataInputStream in;
          DataOutputStream out;
          Socket clientSocket;
```


Java TCP Server (cont.)

```
          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()     // read a line of data from the stream
                out.writeUTF(data);
                clientSocket.close();
             } catch (EOFException e){System.out.println("EOF:"+e.getMessage());
             } catch (IOException e) {System.out.println("readline:"+e.getMessage());}
          }
}
```


Outline of Lectures

- API for Internet protocols (4.2)
- **Marshalling and external data representations (4.3)**
  - **CORBA's Common Data Representation (CDR)**
  - **Java object serialization**
  - **Remote object references**
- Client-server communication (4.4)
- Group communication (4.5)

External Data Representation

- Abstraction gap:
  - Applications store information in data structures
  - Networks deliver messages in a stream of bytes
- The gap must be bridged
  - Senders flatten (aka linearize or marshall) data structures into a stream of bytes, and send in message
  - Receivers unmarshall bytes in message into data structures
- Other sender-receiver problems with data: data representations
  - Floating point: many different representations
  - Characters: ASCII vs. EBCDID vs. Unicide
  - Integers: Big-endian, Little-endian
- Ways to handle data representation
  - Canonical intermediate form
  - Receiver makes right
- An external data representation (XDR) is an agreed way to flatten and represent data
  - Note: Sun XDR …

Marshalling

- Marshalling: taking collection of data items and assembling them so they can be transmitted in a message
- Unmarshalling: the opposite
- Can marshal into
  - Binary format
  - ASCII or other text
- Two different kinds of XDR and marshalling
  - CORBA's Common Data Representation (CDR), for data structures and primitive types; useable by a number of programming languages
  - Java's object serialization
  - (Blurring the line: CORBA's new value types (Object by Value: OBV))

CORBA's CDR

- Can represent all data types that can be CORBA parameters and return values
  - 15 primitive types: **short**, **long**, …

Type	Representation
sequence	length (unsigned long) then elements in order
string	length (unsigned long) then characters in order
struct	elements in the order declared in the struct
enumerated	unsigned long (values generated from order declared)
union	type tag followed by the selected member

CORBA CDR Message

struct Person {
    string name;
    string place;
    long year;
};

Index in bytes	4 bytes' contents	Notes
0—3	5	Length of string
4—7	"Smit"	'Smith'
8—11	"h" 0 0 0	
12—15	6	Length of string
16—19	"Lond"	'London'
20—23	"on" 0 0	
24—27	1934	unsigned long

Java Object Serialization

- In Java RMI, not only primitive data values, but also data values may be passed as arguments and return values
- Serialization: Java term for flattening an object or connected set of objects into a form suitable for storing on disk or sending in a message
- Deserialization: opposite
- A class can be serializable by adding to its definition line
  class foo … implements Serializable
- Serialization also serializes objects refered to in an object's fields
  - Tricky serialization: has to ensure only serialized once
- Java serialization generally automatic without programmer
  - Programmer can intervene to provide own serialization
  - Programmer may declare a field *transient* to note it should not be serialized (local info: port, socket, file descriptor, …)
- Java serialization uses *reflection*: ability to inquire about properties of a class

Remote Object References

- Remote object reference: identifier for a remote object that's valid in the entire system
  - Must be generated to ensure this uniqueness
  - Even if the object has long exited forever ….
- To generate a remote object reference
  - <IP Address>
  - Port
  - Interface of remote object
  - Local distinguishing info
- Local distinguishing info possibilities
  - <time of creation, creation count>
  - <creation count>

Outline of Lectures

- API for Internet protocols (4.2)
- Marshalling and external data representations (4.3)
- **Client-Server Communicaiton (4.4)**
- Group communication (4.5)

Client-Server Communication

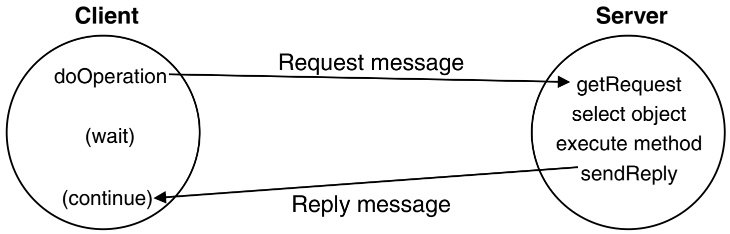
- Request-reply communications here are synchronous
  - Client usually blocks until reply arrives
  - Note: realible, in a sense, since reply is an ACK
- Client-server communciation expressed in terms of *send* and *receive*
- Building *send* and *receive* over UDP avoids TCP overheads
  - TCP ACKs are redundant: replies serve as ACKs
  - Connection establishment requires two more pairs of messages
  - Flow control is not needed for most applications: not much data in params
- Underlying operations to build on top of send and receive:

public byte[] doOperation(RemoteObjectRef o, int methodID,byte [] arguments)

public byte[] getRequest()

public void sendReply(byte[] reply, InetAddress clientHost, int clientPort)

Request-Reply Communication



Request-reply message structure

Field Name	Type
messageType	int (0=Request; 1=Reply)
requestID	Int
objectReference	RemoteObjectRef
methodID	int or Method
arguments	// array of bytes

Request-Reply Implementation

- (See then handout “Request-Reply-Implementation” … it won’t be readable on an overhead!!!!)
- (It is on the web page….)

Failure Model of Request-Reply Protocol

- doOperation, getRequest, and sendReply use UDP send/receive, so
  - Suffer from omission failures
  - Not guaranteed to be delivered in sender order
  - Process failures further complicate things
- Lots of techniques to handle things that can go wrong
  - Timeouts
  - Discarding duplicate request messages
  - Lost reply messages with idempotent server: re-execute request
  - Lost reply messages with non-idempotent server: send reply from history buffer

RPC Exchange Protocols

- **R** (Request)
  - Client sends Request
  - No Reply needed
- **RR** (Request-Reply)
  - Request and reply
  - No explicit ACKs needed
- **RRA** (Request-Reply Acknowledge)
  - Request and Reply
  - Client sends ACK of reply to server
  - Server can trim its history buffer
- Read more about this in text, plus HTTPexample …

Outline of Lectures

- API for Internet protocols (4.2)
- Marshalling and external data representations (4.3)
- Client-Server communication (4.4)
- **Group communication (4.5)**
  - **IP Multicast: an example**
  - **Reliability and ordering of multicast**

Group Communication

- Multicast send: a primitive that allows one message to be sent to multiple destinations (a group)
- Sender does not have to know the destinations, just a group ID/name
- Different uses for multicast
  - Fault tolerance with replicated servers
    - Also can optimize reads to closest replica
  - Finding the discovery servers in spontaneous networking
  - Better performance through replicated data: push updates of data to replicas
  - Propagation of event notifications

IP Multicast

- Basic ideas
  - Built on top of IP
  - Sender unaware of endpoint IP addresses
  - IP multicast group specified by a Class D Internet address
  - Joining the group means you start receiving packets sent to it
  - Only available with UDP
- Failure Model
  - Same as UDP datagrams! ….
  - Messages may not be delivered to any given group member, even if only one UDP omission failure occurs
  - So IP multicast is called *unreliable multicast*
  - Also sender order is not preserved
  - We will discuss reliable multicast in Chapter 11 later on…

Effects of Unreliable+Unordered Multicast

- So how do they affect our potential uses of multicast?
  - Fault tolerance with replicated servers
    - Also can optimize reads to closest replica
  - Finding the discovery servers in spontaneous networking
  - Better performance through replicated data: push updates of data to replicas
  - Propagation of event notifications
- Need reliable multicast
  - Atomic multicast: received by "all or none" of the group members
- Need ordered multicast
  - Messages received in consistent order specified by sender
- Multicast orderings (strongest to weakest)
  - Total: all members receive all messages from all senders in same order
  - Causal: all members receive all messages from all senders in order of "potential causality", i.e. logical time
  - FIFO: sender order
  - Unordered: no guarantees on delivery order, but still may be reliable