Interprocess communication

- How do two processes communicate with each other?
- This is the fundamental building block of distributed systems.
- Other techniques, like remote method invocation, are implemented on top of this layer. They are just an abstraction above raw interprocess communication.

Interprocess communication (IPC)

- IPC (chapter 4) and RMI (chapter 5) are all about middleware. They are the abstraction layer above lower level protocols that actually move data between processes.

	Layers	
	Applications, services	
	RMI and RPC	Middleware layers
This chapter	request-reply protocol	Middleware layers
This chapter	marshalling and external data representation	Middleware layers
	UDP and TCP	

© Pearson Education 2005

Interprocess communication

- IPC is all about *message passing*.
- Message passing involves two sides:
  - Sender: The source of the outgoing message.
  - Receiver: The destination of the outgoing message.

- Typically processes on the endpoints use *send* and *receive* operations in their API to perform this action of sending or receiving.
  - C sockets
  - Java sockets

Characteristics of IPC

- Synchronous vs. asynchronous messages
- Message destinations
  - Addressing, ports.
- Reliability
- Ordering

Synchronous vs. asynchronous comms.

- Synchronous messaging: Both sender and receiver synchronize for every message.
  - This is achieved by blocking on the paired send/recv until the communication is complete.
- Asynchronous messaging: One or both sides does not synchronize with the other.
  - Send or receive does not block, and immediately returns before the communication completes.
- Sometimes the difference is called blocking vs. non-blocking communications.

Asynchronous communications

- Clearly some issues arise that aren't present in the synchronous case.
- Consider first how a synchronous receive works.
- The receiver makes a call to `recv`() and provides a buffer into which it wants the message to be placed.
  - E.g.: `recv(src, &buf, BUFSIZE, …);`
- When the `recv`() completes and the receiver unblocks, the buffer will be guaranteed to have the message in it (or some error code will have occurred to indicate a problem).
- What about asynchronous?

Asynchronous communications

- In the asynchronous receive, the receive is posted and the buffer provided, but the call returns immediately.
  - No guarantee on what the buffer actually contains. It will get filled in by the communications subsystem when the message arrives at some point in the future.
- So, with asynchronous messaging, one needs to take care to check if communications have completed before using the contents of the buffers.
- Similar issue arises on sending side. Need to make sure buffer has been transmitted and safe to overwrite if using asynchronous sends.

Asynchronous communications

- The book talks about today's systems not providing these forms of receive. This isn't universally true.
- The point of asynchronous messaging is to overlap computation with communication.
  - Instead of blocking, do useful work while the messages transmit.
- This is tedious to write. BUT, some systems do provide it.
- Most common in high performance computing, where every cycle is precious.
  - Example: MPI immediate send/receive calls

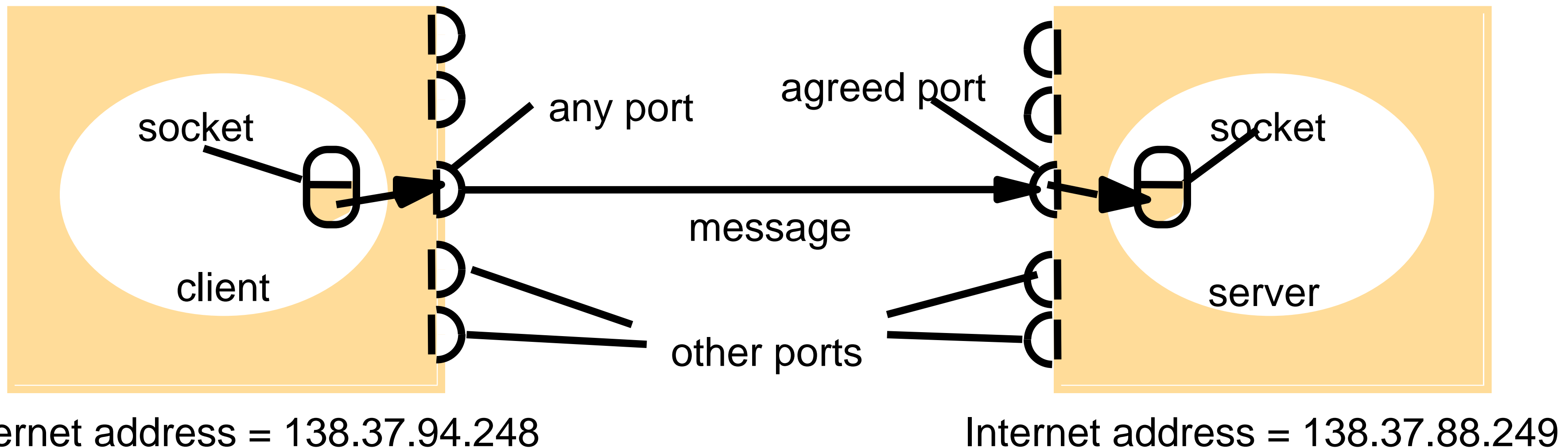
Message destinations

- Clearly the IPC layer must allow you to identify where a message is intended to go.
- Typically this uses the address/port scheme that TCP/IP systems use.
  - E.g.: www.cs.uoregon.edu:80 == Named host can be resolved into IP address, 80 is the port on the machine associated with the web server process.
- Other abstractions exist beyond the simple host/port scheme.
  - E.g.: Group communications.

Sockets

- Sockets are used for TCP and UDP to represent the endpoints of communications.
- Sockets are associated with a port on a host.
  - So, messages sent to a specific address and port will end up in a specific socket on that port, if one is open.
- Processes may use multiple ports for communications, but each port is associated with one process only for receiving purposes.
- Many processes are allowed to send to the same port.
  - E.g.: One web server process listens to port 80 on the web server, but many hosts can send HTTP requests to that port on the server.

Sockets



- Two processes agree on a port for sending data to, but the port out of which the data flows from the sender doesn't matter.

© Pearson Education 2005

Finding hosts by name

- One of the first things you care about as a programmer is finding hosts on the network to connect to.
- Internet "Domain Name Service" resolves symbolic names to the IP addresses assigned to the host.
- Java: `java.net.InetAddress`
  - `InetAddress.getByName()` : Given a name, return an `InetAddress` object representing the IP address(es).
  - `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);`
    - And some other junk

UDP Datagrams

- Once we can find hosts, we want to talk to them.
- UDP is a simple way. UDP allows data to be transmitted from a sending process to a receiving process without any acknowledgement of success or retries in the event of failure.
- The endpoints **bind** sockets to their local address and some port.
  - The server binds to the port that is known to clients so they can find it.
  - The clients bind to any free local port to send out of.
- The receive operation returns the data plus the information about who sent it in case the receiver wants to reply.

Blocking semantics

- Send: Blocks until data is safely handed to the underlying IP and UDP protocols.
- Receive: Blocks until a packet arrives.
  - Timeouts can be set in the event that a datagram is lost and never arrives.
  - Sometimes multiple threads used so one thread can block on the receive and others can work while it waits. Polling is often also possible to check if data is available before going into the blocking receive to avoid threading.
- Messages that arrive before the receive operation is invoked are queued at the socket level. The application programmer isn't responsible for the queue.
  - Messages that arrive for a port that no process has open are discarded.

UDP failure model

- Recall the failure model from the fundamental models.

- **Omission failures**: UDP doesn't guarantee messages will make it, so they are occasionally dropped.
- **Ordering**: UDP doesn't force messages to be delivered in order.

- Applications using UDP often provide application-level checks to deal with these.

Uses of UDP

- Applications where you want lower overhead and occasional omission failures are OK.
  - E.g.: Internet domain name services, Voice over IP
- VoIP is an interesting example. Losing a packet representing some tiny duration of sound is easily compensated for at playback.
- Where does overhead come from?
  - Statefulness of connection on sender/receiver sides
  - Transmission of extra messages for setting up a connection
  - Latency for the sender.

Java UDP

- See chapter 4.2 for sample code.
- Key points:
  - `java.net.DatagramSocket` : This sets up the sockets on the endpoints of the communication.
  - `java.net.DatagramPacket` : This represents the data that is sent, including the network information about the sender.

  - You can see in Figure 4.4 how the receiver uses the network information and data in the received datagram to construct a packet to echo back to the original sender.