CS162
Operating Systems and Systems Programming
Lecture 22

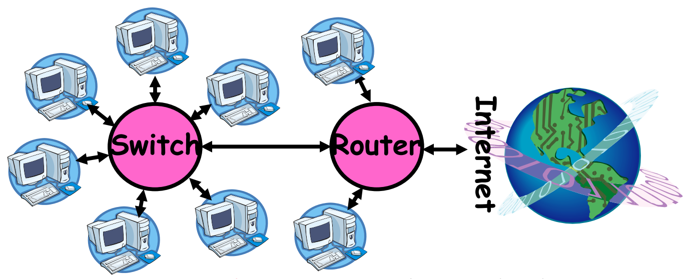
Networking II

November 19, 2007

Prof. John Kubiatowicz

http://inst.eecs.berkeley.edu/~cs162

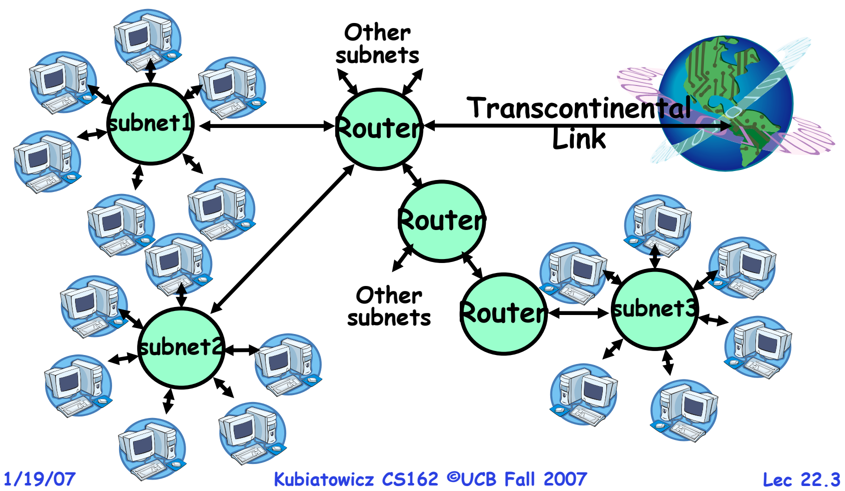
Review: Point-to-point networks



- **Point-to-point network:** a network in which every physical wire is connected to only two computers
- **Switch:** a bridge that transforms a shared-bus (broadcast) configuration into a point-to-point network.
- **Hub:** a multiport device that acts like a repeater broadcasting from each input to every output
- **Router:** a device that acts as a junction between two networks to transfer data packets among them.

Review: Hierarchical Networking (The Internet)

- How can we build a network with millions of hosts?
  - Hierarchy! Not every host connected to every other one
  - Use a network of Routers to connect subnets together



Review: Routing

- Routing: the process of forwarding packets hop-by-hop through routers to reach their destination
  - Need more than just a destination address!
    - » Need a path
  - Post Office Analogy:
    - » Destination address on each letter is not sufficient to get it to the destination
    - » To get a letter from here to Florida, must route to local post office, sorted and sent on plane to somewhere in Florida, be routed to post office, sorted and sent with carrier who knows where street and house is…
- **Internet routing mechanism: routing tables**
  - Each router does table lookup to decide which link to use to get packet closer to destination
  - Don't need 4 billion entries in table: routing is by subnet
  - Could packets be sent in a loop? Yes, if tables incorrect
- Routing table contains:
  - Destination address range → output link closer to destination
  - Default entry (for subnets without explicit entries)

Goals for Today

- Networking
  - Protocols
  - Reliable Messaging
    - » TCP windowing and congestion avoidance
  - Two-phase commit

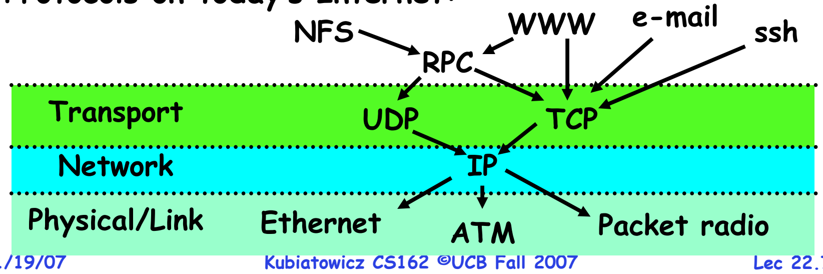
Note: Some slides and/or pictures in the following are adapted from slides ©2005 Silberschatz, Galvin, and Gagne. Many slides generated from my lecture notes by Kubiatowicz.

Setting up Routing Tables

- How do you set up routing tables?
  - Internet has no centralized state!
    - » No single machine knows entire topology
    - » Topology constantly changing (faults, reconfiguration, etc)
  - Need dynamic algorithm that acquires routing tables
    - » Ideally, have one entry per subnet or portion of address
    - » Could have "default" routes that send packets for unknown subnets to a different router that has more information
- Possible algorithm for acquiring routing table
  - Routing table has "cost" for each entry
    - » Includes number of hops to destination, congestion, etc.
    - » Entries for unknown subnets have infinite cost
  - Neighbors periodically exchange routing tables
    - » If neighbor knows cheaper route to a subnet, replace your entry with neighbors entry (+1 for hop to neighbor)
- In reality:
  - Internet has networks of many different scales
  - Different algorithms run at different scales

Network Protocols

- Protocol: Agreement between two parties as to how information is to be transmitted
  - Example: system calls are the protocol between the operating system and application
  - Networking examples: many levels
    - » Physical level: mechanical and electrical network (e.g. how are 0 and 1 represented)
    - » Link level: packet formats/error control (for instance, the CSMA/CD protocol)
    - » Network level: network routing, addressing
    - » Transport Level: reliable message delivery
- Protocols on today's Internet:



Network Layering

- Layering: building complex services from simpler ones
  - Each layer provides services needed by higher layers by utilizing services provided by lower layers
- The physical/link layer is pretty limited
  - Packets are of limited size (called the "Maximum Transfer Unit or MTU: often 200-1500 bytes in size)
  - Routing is limited to within a physical link (wire) or perhaps through a switch
- Our goal in the following is to show how to construct a secure, ordered, message service routed to anywhere:

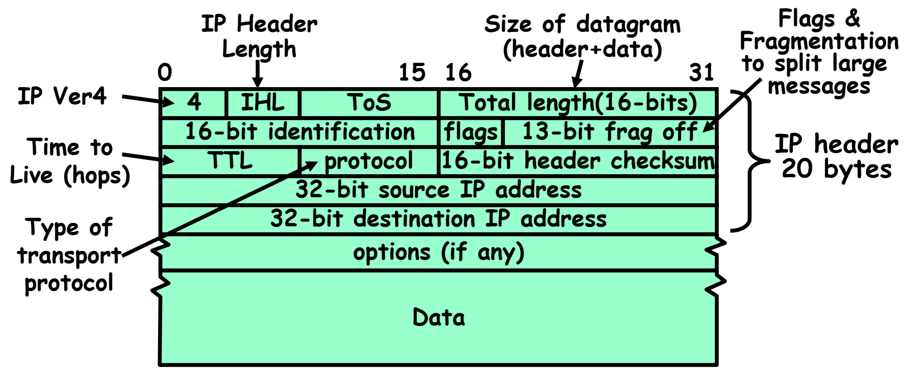
Physical Reality: Packets	Abstraction: Messages
Limited Size	Arbitrary Size
Unordered (sometimes)	Ordered
Unreliable	Reliable
Machine-to-machine	Process-to-process
Only on local area net	Routed anywhere
Asynchronous	Synchronous
Insecure	Secure

Building a messaging service

- Handling Arbitrary Sized Messages:
  - Must deal with limited physical packet size
  - Split big message into smaller ones (called fragments)
    - » Must be reassembled at destination
  - Checksum computed on each fragment or whole message
- Internet Protocol (IP): Must find way to send packets to arbitrary destination in network
  - Deliver messages unreliably ("best effort") from one machine in Internet to another
  - Since intermediate links may have limited size, must be able to fragment/reassemble packets on demand
  - Includes 256 different "sub-protocols" build on top of IP
    - » Examples: ICMP(1), TCP(6), UDP (17), IPSEC(50,51)

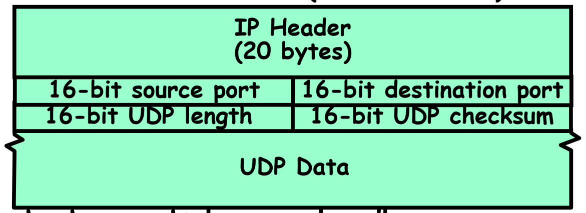
IP Packet Format

- IP Packet Format:



Building a messaging service

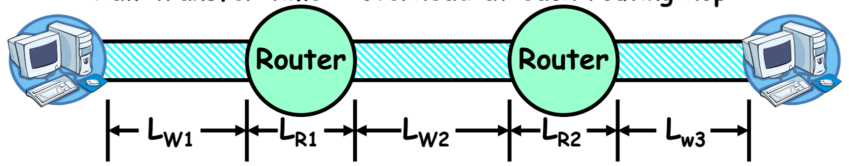
- Process to process communication
  - Basic routing gets packets from machine→machine
  - What we really want is routing from process→process
    - » Add "ports", which are 16-bit identifiers
    - » A communication channel (connection) defined by 5 items: [source addr, source port, dest addr, dest port, protocol]
- UDP: The Unreliable Datagram Protocol
  - Layered on top of basic IP (IP Protocol 17)
    - » Datagram: an unreliable, unordered, packet sent from source user → dest user (Call it UDP/IP)



  - Important aspect: low overhead!
    - » Often used for high-bandwidth video streams
    - » Many uses of UDP considered "anti-social" – none of the "well-behaved" aspects of (say) TCP/IP

Performance Considerations

- Before we continue, need some performance metrics
  - Overhead: CPU time to put packet on wire
  - Throughput: Maximum number of bytes per second
    - » Depends on "wire speed", but also limited by slowest router (routing delay) or by congestion at routers
  - Latency: time until first bit of packet arrives at receiver
    - » Raw transfer time + overhead at each routing hop



- Contributions to Latency
  - Wire latency: depends on speed of light on wire
    - » about 1-1.5 ns/foot
  - Router latency: depends on internals of router
    - » Could be < 1 ms (for a good router)
    - » Question: can router handle full wire throughput?

Sample Computations

- E.g.: Ethernet within Soda
  - Latency: speed of light in wire is 1.5ns/foot, which implies latency in building < 1 μs (if no routers in path)
  - Throughput: 10-1000Mb/s
  - Throughput delay: packet doesn't arrive until all bits
    - » So: 4KB/100Mb/s = 0.3 milliseconds (same order as disk!)
- E.g.: ATM within Soda
  - Latency (same as above, assuming no routing)
  - Throughput: 155Mb/s
  - Throughput delay: 4KB/155Mb/s = 200μ
- E.g.: ATM cross-country
  - Latency (assuming no routing):
    - » 3000miles * 5000ft/mile $\Rightarrow$ 15 milliseconds
  - How many bits could be in transit at same time?
    - » 15ms * 155Mb/s = 290KB
  - In fact, Berkeley$\rightarrow$MIT Latency ~ 45ms
    - » 872KB in flight if routers have wire-speed throughput
- Requirements for good performance:
  - Local area: minimize overhead/improve bandwidth
  - Wide area: keep pipeline full!

Sequence Numbers

- Ordered Messages
  - Several network services are best constructed by ordered messaging
    - » Ask remote machine to first do x, then do y, etc.
  - Unfortunately, underlying network is packet based:
    - » Packets are routed one at a time through the network
    - » Can take different paths or be delayed individually
  - IP can reorder packets! P_0,P_1 might arrive as P_1,P_0
- Solution requires queuing at destination
  - Need to hold onto packets to undo misordering
  - Total degree of reordering impacts queue size
- Ordered messages on top of unordered ones:
  - Assign sequence numbers to packets
    - » 0,1,2,3,4…..
    - » If packets arrive out of order, reorder before delivering to user application
    - » For instance, hold onto #3 until #2 arrives, etc.
  - Sequence numbers are specific to particular connection
    - » Reordering among connections normally doesn't matter
  - If restart connection, need to make sure use different range of sequence numbers than previously…

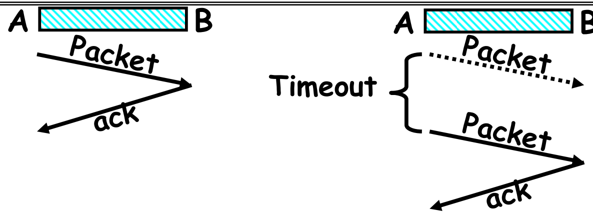
Administrivia

- Projects:
  - Project 4 design document due November 27th
    - » Although this is after Thanksgiving – make good use of time since this is a difficult project
- MIDTERM II: Dec 5th
  - Having trouble finding room….
    - » What do people think about having no midterm II?
  - Assuming that we have the midterm:
    - » All material from last midterm and up to Monday 12/3
    - » Lectures #12 – 26
    - » One cheat sheet (both sides)
- Final Exam
  - » Sat Dec 17th, 5:00-8:00pm
  - » All Material
  - » Two Cheat sheets.
- Final Topics: Any suggestions?
  - Please send them to me…

Reliable Message Delivery: the Problem

- All physical networks can garble and/or drop packets
  - Physical media: packet not transmitted/received
    - » If transmit close to maximum rate, get more throughput – even if some packets get lost
    - » If transmit at lowest voltage such that error correction just starts correcting errors, get best power/bit
  - Congestion: no place to put incoming packet
    - » Point-to-point network: insufficient queue at switch/router
    - » Broadcast link: two host try to use same link
    - » In any network: insufficient buffer space at destination
    - » Rate mismatch: what if sender send faster than receiver can process?
- Reliable Message Delivery on top of Unreliable Packets
  - Need some way to make sure that packets actually make it to receiver
    - » Every packet received at least once
    - » Every packet received at most once
  - Can combine with ordering: every packet received by process at destination exactly once and in order

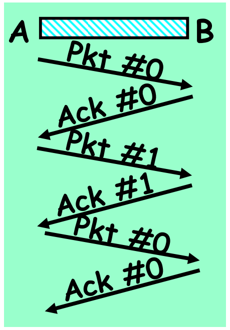
Using Acknowledgements



- How to ensure transmission of packets?
  - Detect garbling at receiver via checksum, discard if bad
  - Receiver acknowledges (by sending "ack") when packet received properly at destination
  - Timeout at sender: if no ack, retransmit
- Some questions:
  - If the sender doesn't get an ack, does that mean the receiver didn't get the original message?
    - » No
  - What if ack gets dropped? Or if message gets delayed?
    - » Sender doesn't get ack, retransmits. Receiver gets message twice, acks each.

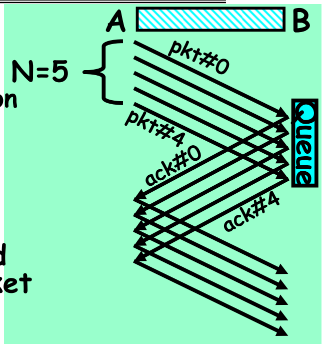
How to deal with message duplication

- Solution: put sequence number in message to identify re-transmitted packets
  - Receiver checks for duplicate #'s; Discard if detected
- Requirements:
  - Sender keeps copy of unack'ed messages
    - » Easy: only need to buffer messages
  - Receiver tracks possible duplicate messages
    - » Hard: when ok to forget about received message?
- Alternating-bit protocol:
  - Send one message at a time; don't send next message until ack received
  - Sender keeps last message; receiver tracks sequence # of last message received
- Pros: simple, small overhead
- Con: Poor performance
  - Wire can hold multiple messages; want to fill up at (wire latency × throughput)
- Con: doesn't work if network can delay or duplicate messages arbitrarily

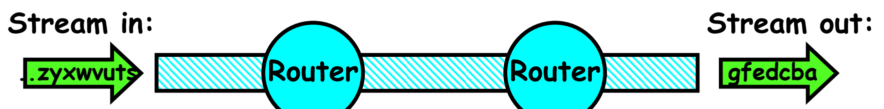


Better messaging: Window-based acknowledgements

- Window based protocol (TCP):
  - Send up to N packets without ack
    - » Allows pipelining of packets
    - » Window size (N) < queue at destination
  - Each packet has sequence number
    - » Receiver acknowledges each packet
    - » Ack says "received all packets up to sequence number X"/send more
- Acks serve dual purpose:
  - Reliability: Confirming packet received
  - Flow Control: Receiver ready for packet
    - » Remaining space in queue at receiver can be returned with ACK
- What if packet gets garbled/dropped?
  - Sender will timeout waiting for ack packet
    - » Resend missing packets⇒ Receiver gets packets out of order!
  - Should receiver discard packets that arrive out of order?
    - » Simple, but poor performance
  - Alternative: Keep copy until sender fills in missing pieces?
    - » Reduces # of retransmits, but more complex
- What if ack gets garbled/dropped?
  - Timeout and resend just the un-acknowledged packets

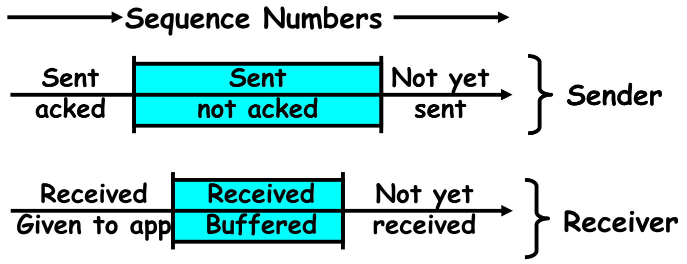


Transmission Control Protocol (TCP)



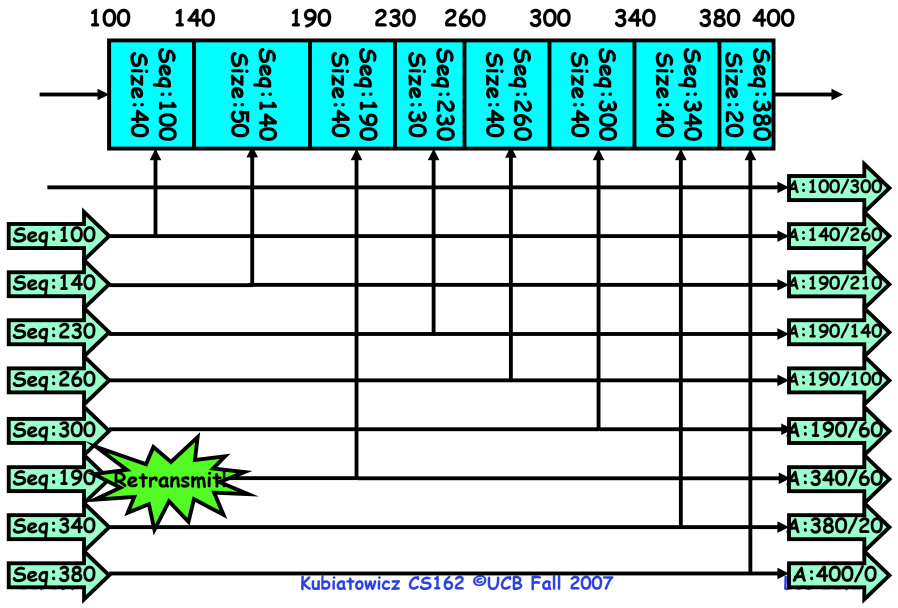
- Transmission Control Protocol (TCP)
  - TCP (IP Protocol 6) layered on top of IP
  - Reliable byte stream between two processes on different machines over Internet (read, write, flush)
- TCP Details
  - Fragments byte stream into packets, hands packets to IP
    - » IP may also fragment by itself
  - Uses window-based acknowledgement protocol (to minimize state at sender and receiver)
    - » "Window" reflects storage at receiver – sender shouldn't overrun receiver's buffer space
    - » Also, window should reflect speed/capacity of network – sender shouldn't overload network
  - Automatically retransmits lost packets
  - Adjusts rate of transmission to avoid congestion
    - » A "good citizen"

TCP Windows and Sequence Numbers

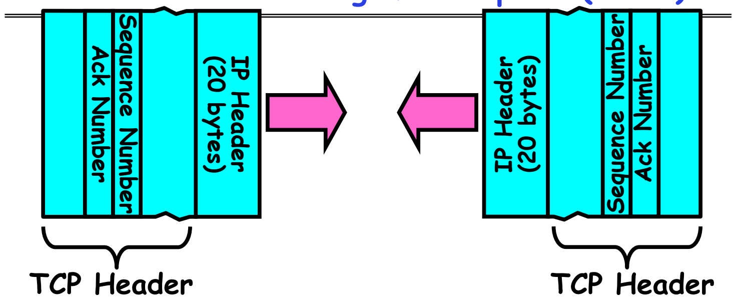


- Sender has three regions:
  - Sequence regions
    - » sent and ack'ed
    - » Sent and not ack'ed
    - » not yet sent
  - Window (colored region) adjusted by sender
- Receiver has three regions:
  - Sequence regions
    - » received and ack'ed (given to application)
    - » received and buffered
    - » not yet received (or discarded because out of order)

Window-Based Acknowledgements (TCP)



Selective Acknowledgement Option (SACK)



- Vanilla TCP Acknowledgement
  - Every message encodes Sequence number and Ack
  - Can include data for forward stream and/or ack for reverse stream
- Selective Acknowledgement
  - Acknowledgement information includes not just one number, but rather ranges of received packets
  - Must be specially negotiated at beginning of TCP setup
    - » Not widely in use (although in Windows since Windows 98)

Congestion Avoidance

- Congestion
  - How long should timeout be for re-sending messages?
    - » Too long→wastes time if message lost
    - » Too short→retransmit even though ack will arrive shortly
  - Stability problem: more congestion ⇒ ack is delayed ⇒ unnecessary timeout ⇒ more traffic ⇒ more congestion
    - » Closely related to window size at sender: too big means putting too much data into network
- How does the sender's window size get chosen?
  - Must be less than receiver's advertised buffer size
  - Try to match the rate of sending packets with the rate that the slowest link can accommodate
  - Sender uses an adaptive algorithm to decide size of N
    - » Goal: fill network between sender and receiver
    - » Basic technique: slowly increase size of window until acknowledgements start being delayed/lost
- TCP solution: "slow start" (start sending slowly)
  - If no timeout, slowly increase window size (throughput) by 1 for each ack received
  - Timeout ⇒ congestion, so cut window size in half
  - *"Additive Increase, Multiplicative Decrease"*

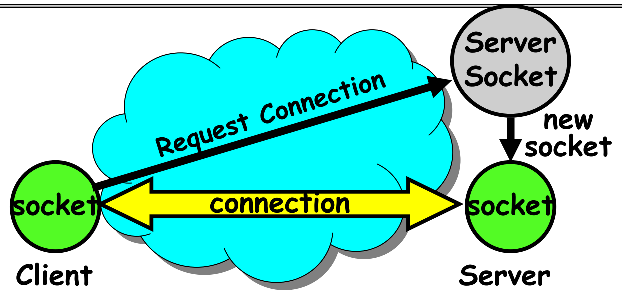
Sequence-Number Initialization

- How do you choose an initial sequence number?
  - When machine boots, ok to start with sequence #0?
    - » No: could send two messages with same sequence #!
    - » Receiver might end up discarding valid packets, or duplicate ack from original transmission might hide lost packet
  - Also, if it is possible to predict sequence numbers, might be possible for attacker to hijack TCP connection
- Some ways of choosing an initial sequence number:
  - Time to live: each packet has a deadline.
    - » If not delivered in X seconds, then is dropped
    - » Thus, can re-use sequence numbers if wait for all packets in flight to be delivered or to expire
  - Epoch #: uniquely identifies *which* set of sequence numbers are currently being used
    - » Epoch # stored on disk, Put in every message
    - » Epoch # incremented on crash and/or when run out of sequence #
  - Pseudo-random increment to previous sequence number
    - » Used by several protocol implementations

Use of TCP: Sockets

- Socket: an abstraction of a network I/O queue
  - Embodies one side of a communication channel
    - » Same interface regardless of location of other end
    - » Could be local machine (called "UNIX socket") or remote machine (called "network socket")
  - First introduced in 4.2 BSD UNIX: big innovation at time
    - » Now most operating systems provide some notion of socket
- Using Sockets for Client-Server (C/C++ interface):
  - On server: set up "server-socket"
    - » Create socket, Bind to protocol (TCP), local address, port
    - » Call listen(): tells server socket to accept incoming requests
    - » Perform multiple accept() calls on socket to accept incoming connection request
    - » Each successful accept() returns a new socket for a new connection; can pass this off to handler thread
  - On client:
    - » Create socket, Bind to protocol (TCP), remote address, port
    - » Perform connect() on socket to make connection
    - » If connect() successful, have socket connected to server

Socket Setup (Con't)



- Things to remember:
  - Connection requires 5 values:
    [Src Addr, Src Port, Dst Addr, Dst Port, Protocol]
  - Often, Src Port "randomly" assigned
    - » Done by OS during client socket setup
  - Dst Port often "well known"
    - » 80 (web), 443 (secure web), 25 (sendmail), etc
    - » Well-known ports from 0—1023

Socket Example (Java)

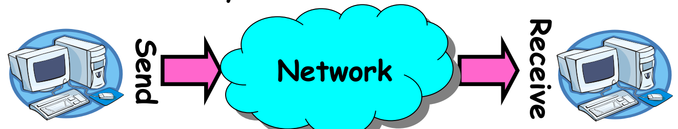
```
server:
     //Makes socket, binds addr/port, calls listen()
     ServerSocket sock = new ServerSocket(6013);
     while(true) {
       Socket client = sock.accept();
       PrintWriter pout = new
         PrintWriter(client.getOutputStream(),true);

       pout.println("Here is data sent to client!");
         …
       client.close();
     }

client:
     // Makes socket, binds addr/port, calls connect()
     Socket sock = new Socket("169.229.60.38",6013);
     BufferedReader bin =
       new BufferedReader(
         new InputStreamReader(sock.getInputStream));
     String line;
     while ((line = bin.readLine())!=null)
       System.out.println(line);
     sock.close();
```

Distributed Applications

- How do you actually program a distributed application?
  - Need to synchronize multiple threads, running on different machines
    - » No shared memory, so cannot use test&set



  - One Abstraction: send/receive messages
    - » Already atomic: no receiver gets portion of a message and two receivers cannot get same message
- Interface:
  - Mailbox (mbox): temporary holding area for messages
    - » Includes both destination location and queue
  - `Send(message,mbox)`
    - » Send message to remote mailbox identified by mbox
  - `Receive(buffer,mbox)`
    - » Wait until mbox has message, copy into buffer, and return
    - » If threads sleeping on this mbox, wake up one of them

Using Messages: Send/Receive behavior

- When should `send(message,mbox)` return?
  - When receiver gets message? (i.e. ack received)
  - When message is safely buffered on destination?
  - Right away, if message is buffered on source node?
- Actually two questions here:
  - When can the sender be sure that the receiver actually received the message?
  - When can sender reuse the memory containing message?
- Mailbox provides 1-way communication from T1→T2
  - T1→buffer→T2
  - Very similar to producer/consumer
    - » Send = V, Receive = P
    - » However, can't tell if sender/receiver is local or not!

Messaging for Producer-Consumer Style

- Using send/receive for producer-consumer style:

```
Producer:
   int msg1[1000];
   while(1) {
      prepare message;
      send(msg1,mbox);
   }
Consumer:
   int buffer[1000];
   while(1) {
      receive(buffer,mbox);
      process message;
   }
```

Send Message

Receive Message

- No need for producer/consumer to keep track of space in mailbox: handled by send/receive
  - One of the roles of the window in TCP: window is size of buffer on far end
  - Restricts sender to forward only what will fit in buffer

Messaging for Request/Response communication

- What about two-way communication?
  - Request/Response
    - » Read a file stored on a remote machine
    - » Request a web page from a remote web server
  - Also called: client-server
    - » Client ≡ requester, Server ≡ responder
    - » Server provides "service" (file storage) to the client
- Example: File service

```
Client: (requesting the file)
   char response[1000];

   send("read rutabaga", server_mbox);
   receive(response, client_mbox);

Consumer: (responding with the file)
   char command[1000], answer[1000];

   receive(command, server_mbox);
   decode command;
   read file into answer;
   send(answer, client_mbox);
```

Request File

Get Response

Receive Request

Send Response

Conclusion

- **Layering:** building complex services from simpler ones
- **Datagram:** an independent, self-contained network message whose arrival, arrival time, and content are not guaranteed
- Performance metrics
  - **Overhead:** CPU time to put packet on wire
  - **Throughput:** Maximum number of bytes per second
  - **Latency:** time until first bit of packet arrives at receiver
- ***Arbitrary Sized messages:***
  - Fragment into multiple packets; reassemble at destination
- **Ordered messages:**
  - Use sequence numbers and reorder at destination
- **Reliable messages:**
  - Use Acknowledgements
  - Want a window larger than 1 in order to increase throughput
- **TCP:** Reliable byte stream between two processes on different machines over Internet (read, write, flush)