Review: Networking

- **Network**: physical connection that allows two computers to communicate
  - Packet: sequence of bits carried over the network
- **Broadcast Network**: Shared Communication Medium
  - Transmitted packets sent to all receivers
  - Arbitration: act of negotiating use of shared medium
    - Ethernet: Carrier Sense, Multiple Access, Collision Detect
- **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.
- **Internet Protocol (IP)**: unreliable packet service
  - Used to route messages across globe
  - 32-bit destination addresses
- **Routing**: the process of forwarding packets hop-by-hop through routers to reach their destination
  - Internet has networks of many different scales
    - LANs, Autonomous Systems (AS), etc.
  - Different algorithms run at different scales
    - Border Gateway Protocol (BGP) at large scales
    - Variants of Distance Vector (DV) protocols at short scales

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: 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**:
  - Transport: TCP, UDP
  - Network: IP
  - Physical/Link: Ethernet, ATM
**Goals for Today**

- Networking
  - 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.

---

**Network Layering**

- Layering: building complex services from simpler ones
  - Each layer provides services needed for higher layers by utilizing services provided by lower layers
  - Our goal in the following is to show how to construct a secure, ordered, arbitrary-sized message service routed to anywhere:

<table>
<thead>
<tr>
<th>Physical Reality: Packets</th>
<th>Abstraction: Messages</th>
</tr>
</thead>
<tbody>
<tr>
<td>Limited Size</td>
<td>Arbitrary Size</td>
</tr>
<tr>
<td>Unordered (sometimes)</td>
<td>Ordered</td>
</tr>
<tr>
<td>Unreliable</td>
<td>Reliable</td>
</tr>
<tr>
<td>Machine-to-machine</td>
<td>Process-to-process</td>
</tr>
<tr>
<td>Only on local area net</td>
<td>Routed anywhere</td>
</tr>
<tr>
<td>Asynchronous</td>
<td>Synchronous</td>
</tr>
<tr>
<td>Insecure</td>
<td>Secure</td>
</tr>
</tbody>
</table>

---

**Basic Networking Limitations**

- The physical/link layer is pretty limited
  - Packets of limited size
    - Maximum Transfer Unit (MTU): often 200-1500 bytes
  - Packets can get lost or garbled
  - Hardware routing limited to physical link or switch
  - Physical routers crash/links get damaged
    - Famous Baltimore tunnel fire (July 2001): cut Internet half
  - Datagram: an independent, self-contained network message whose arrival, arrival time, and content are not guaranteed
  - Need resilient routing algorithms to send messages on wide area
    - Multi-hop routing mechanisms
    - Redundant links/Ability to route around failed links

**Handling Arbitrary Sized Messages:**

- Must deal with limited physical packet size
- Split big message into smaller ones (called fragments)
  - Must be reassembled at destination
- May happen on demand if packet routed through areas of reduced MTU (e.g., TCP)
- Checksum computed on each fragment or whole message

---

**Performance Considerations**

- Before continuing, 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 1.5 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 = 200 µ

- **E.g.: ATM cross-country**
  - Latency (assuming no routing): 3000miles * 5000ft/mile ÷ 15 milliseconds
  - How many bits could be in transit at same time?
    - 15ms * 155Mb/s = 290KB
  - In fact, Berkeley→MIT Latency ~ 90ms
    - Implies 1.7MB in flight if routers have wire-speed throughput

Roles for good performance:
- Local area: minimize overhead/improve bandwidth
- Wide area: keep pipeline full!

IP Packet Format

- **Internet Protocol (IP): Sends packets to arbitrary destination in network**
  - Deliver messages unreliable (“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" built on top of IP
    - Examples: ICMP(1), TCP(6), UDP (17), IPSEC(50,51)

- **IP Packet Format:**
  - *Time to Live (hops)*
  - *Type of protocol* (e.g., TCP, UDP, ICMP)
  - *Total length (16-bits)*
  - *Identification (16-bits)*
  - *Flags & fragmentation to split large messages*

Process-to-process communication: UDP

- **Process to process communication**
  - Basic routing gets packets from machine→machine
  - What we really want is routing from process→process
    - Example: ssh, email, ftp, web browsing
  - Several IP protocols include notion of a "port", which is a 16-bit identifier used in addition to IP addresses
    - A communication channel (connection) defined by 4 items: (source address, source port, dest address, dest port)

- **UDP: The Unreliable Datagram Protocol**
  - UDP layered on top of basic IP (IP Protocol 17)
    - Unreliable, unordered, user-to-user communication

  - Often used for high-bandwidth video streams
    - Many uses of UDP considered "anti-social" - none of the well-behaved aspects of (say) TCP/IP

Administrivia

- **Project 4 design document**
  - Due May 1st

- **MIDTERM II: April 26th**
  - All material from last midterm and up to Monday 4/24

- **Final Exam**
  - May 18th

- **Final Topics: Any suggestions?**
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 queueing 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...

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 only once
  - Can combine with ordering: every packet received by process at destination once and in order

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
- Cons: poor performance
  - Wire can hold multiple messages; want to fill up at (wire latency \( \times \) throughput)
  - Cons: 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
  » 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)

- **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 acked
    » 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)

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 ⇒ 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
  - Specifically TCP solution: "slow start"
    - Start sending slowly
    - If no timeout, slowly increase window size (throughput)
    - Timeout ⇒ congestion, so cut window size in half
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 TCP 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 a number of implementations now

General’s Paradox

• General’s paradox:
  - Constraints of problem:
    » Two generals, on separate mountains
    » Can only communicate via messengers
    » Messengers can be captured
  - Problem: need to coordinate attack
    » If they attack at different times, they all die
    » If they attack at same time, they win
  - Named after Custer, who died at Little Big Horn because he arrived a couple of days too early

• Can messages over an unreliable network be used to guarantee two entities do something simultaneously?
  - Remarkably, “no”, even if all messages get through

Two-Phase Commit

• Since we can’t solve the General’s Paradox (i.e. simultaneous action), let’s solve a related problem
  - Distributed transaction: Two machines agree to do something, or not do it, atomically

• Two-Phase Commit protocol does this
  - Use a persistent, stable log on each machine to keep track of whether commit has happened
    » If a machine crashes, when it wakes up it first checks its log to recover state of world at time of crash
  - Prepare Phase:
    » The global coordinator requests that all participants will promise to commit or rollback the transaction
    » Participants record promise in log, then acknowledge
    » If anyone votes to abort, coordinator writes “abort” in its log and tells everyone to abort; each records “abort” in log
  - Commit Phase:
    » After all participants respond that they are prepared, then the coordinator writes “commit” to its log
    » Then asks all nodes to commit; they respond with ack
    » After receive acks, coordinator writes “got commit” to log
    » Log can be used to complete this process such that all machines either commit or don’t commit

Two phase commit example

• Simple Example: A=ATM machine, B=The Bank
  - Phase 1:
    » A writes “Begin transaction” to log
    » A→B: OK to transfer funds to me?
    » Not enough funds:
    » B→A: transaction aborted; A writes “Abort” to log
    » Enough funds:
    » B: Write new account balance to log
    » B→A: OK, I can commit
  - Phase 2: A can decide for both whether they will commit
    » A: write new account balance to log
    » Write “commit” to log
    » Send message to B that commit occurred; wait for ack
    » Write “Got Commit” to log

• What if B crashes at beginning?
  - Wakes up, does nothing: A will timeout, abort and retry

• What if A crashes at beginning of phase 2?
  - Wakes up, sees transaction in progress; sends “abort” to B

• What if B crashes at beginning of phase 2?
  - B comes back up, look at log; when A sends it “Commit” message, it will say, oh, ok, commit
Distributed Decision Making Discussion

- **Two-Phase Commit: Blocking**
  - A site can get stuck in a situation where it cannot continue until some other site (usually the coordinator) recovers.
  - Example of how this could happen:
    » Participant site B writes a “prepared to commit” record to its log, sends a “yes” vote to the coordinator (site A) and crashes.
    » Site A crashes.
    » Site B wakes up, checks its log, and realizes that it has voted “yes” on the update. It sends a message to site A asking what happened. At this point, B cannot change its mind and decide to abort, because update may have committed.
    » B is blocked until A comes back.
  - Blocking is problematic because a blocked site must hold resources (locks on updated items, pagespinned in memory, etc) until it learns fate of update.
- **Alternative:** There are alternatives such as “Three Phase Commit” which don’t have this blocking problem.

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).
- **Two-phase commit:** distributed decision making.