

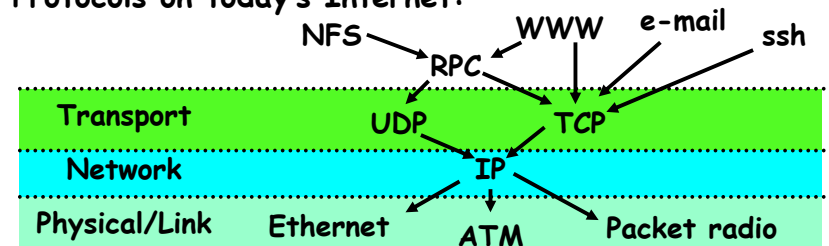
# CS162 Operating Systems and Systems Programming Lecture 23

## Networking III

November 19, 2008  
Prof. John Kubiatowicz  
<http://inst.eecs.berkeley.edu/~cs162>

## 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:



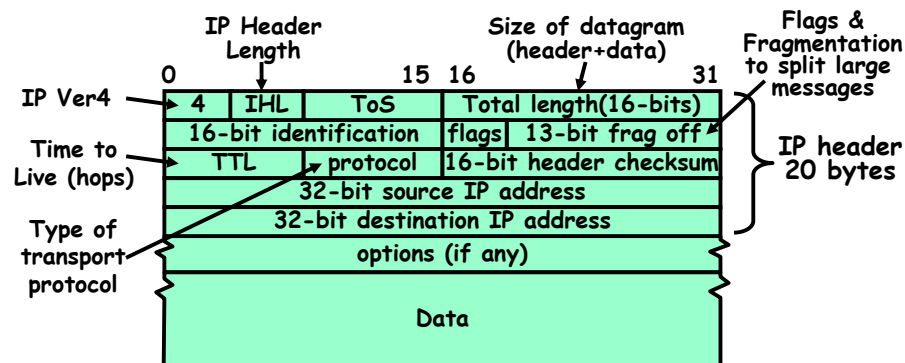
11/17/08

Kubiatowicz CS162 ©UCB Fall 2008

Lec 22.2

## Review: IP Packet Format

### • IP Packet Format:



11/17/08

Kubiatowicz CS162 ©UCB Fall 2008

Lec 22.3

## Goals for Today

- Networking
  - Continue discussion of reliable messaging
  - Sequence numbers for ordering
  - Acknowledgments for reliability
- TCP windowing
- Sockets
- Messages
  - Send/receive
  - One vs. two-way communication

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.

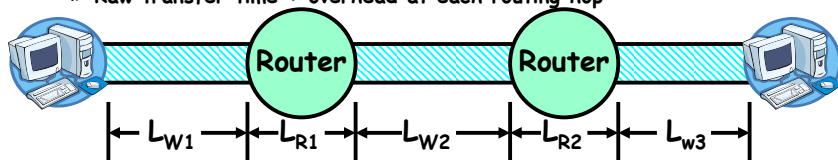
11/17/08

Kubiatowicz CS162 ©UCB Fall 2008

Lec 22.4

## 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 limited by slowest router or by congestion
  - **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?
- What is the End-to-end MTU? **Minimum across path**
- What is the End-to-end Throughput? **Minimum across path**

11/17/08

Kubiatowicz CS162 ©UCB Fall 2008

Lec 22.5

## Sample Computations

- E.g.: Ethernet within Soda
  - Latency: speed of light in wire is 1.5ns/foot, which implies latency in building < 1  $\mu$ s (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 $\mu$
- 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!

11/17/08

Kubiatowicz CS162 ©UCB Fall 2008

Lec 22.6

## 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...

11/17/08

Kubiatowicz CS162 ©UCB Fall 2008

Lec 22.7

## Reliable Message Delivery: the Problem

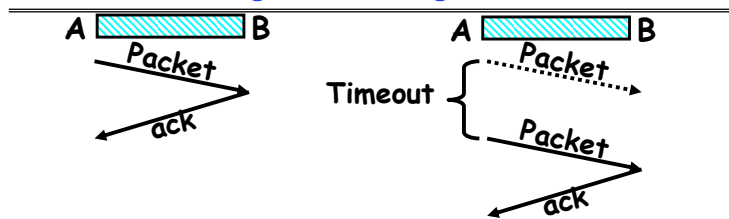
- **All physical networks can garble and/or drop packets**
  - Physical media: packet not transmitted/received
    - » If transmit close to maximum rate, get more throughput - even if some packets get lost
    - » If transmit at lowest voltage such that error correction just starts correcting errors, 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

11/17/08

Kubiatowicz CS162 ©UCB Fall 2008

Lec 22.8

## 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.

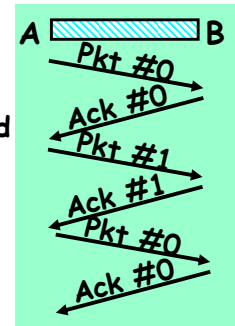
11/17/08

Kubiatowicz CS162 ©UCB Fall 2008

Lec 22.9

## 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



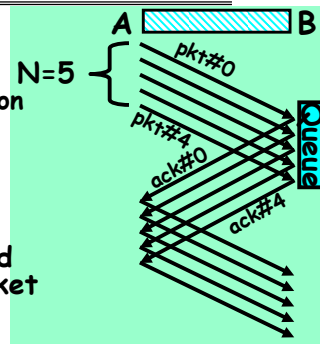
11/17/08

Kubiatowicz CS162 ©UCB Fall 2008

Lec 22.10

## Better messaging: Window-based acknowledgements

- Window based protocol:
  - 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



11/17/08

Kubiatowicz CS162 ©UCB Fall 2008

Lec 22.11

## Administrivia

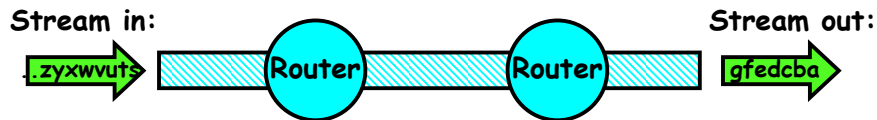
- Projects:
  - Project 4 design document due Monday, November 24th
- MIDTERM II: Wednesday Dec 3rd
  - Location: 10 Evans, 5:30pm - 8:30pm
  - Topics:
    - » All material from last midterm and up to Monday 12/1
    - » Lectures #13 - 26
    - » One cheat sheet (both sides)
- Final Exam
  - Thursday, Dec 18th, 8:00-11:00am
  - Topics: All Material except last lecture (freebie)
  - Two Cheat sheets.
- Final Topics: Any suggestions?
  - Please send them to me...

11/17/08

Kubiatowicz CS162 ©UCB Fall 2008

Lec 22.12

## Transmission Control Protocol (TCP)



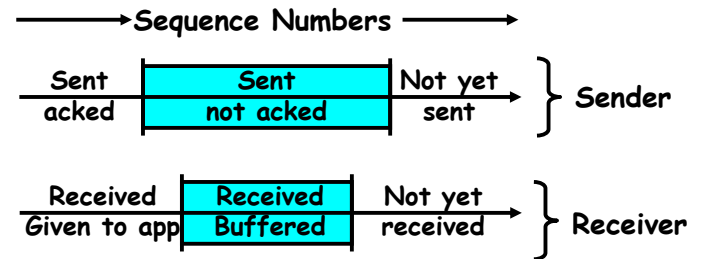
- **Transmission Control Protocol (TCP)**
  - TCP (IP Protocol 6) layered on top of IP
  - Reliable byte stream between two processes on different machines over Internet (read, write, flush)
- **TCP Details**
  - Fragments byte stream into packets, hands packets to IP
    - » IP may also fragment by 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"

11/17/08

Kubiatowicz CS162 ©UCB Fall 2008

Lec 22.13

## TCP Windows and Sequence Numbers



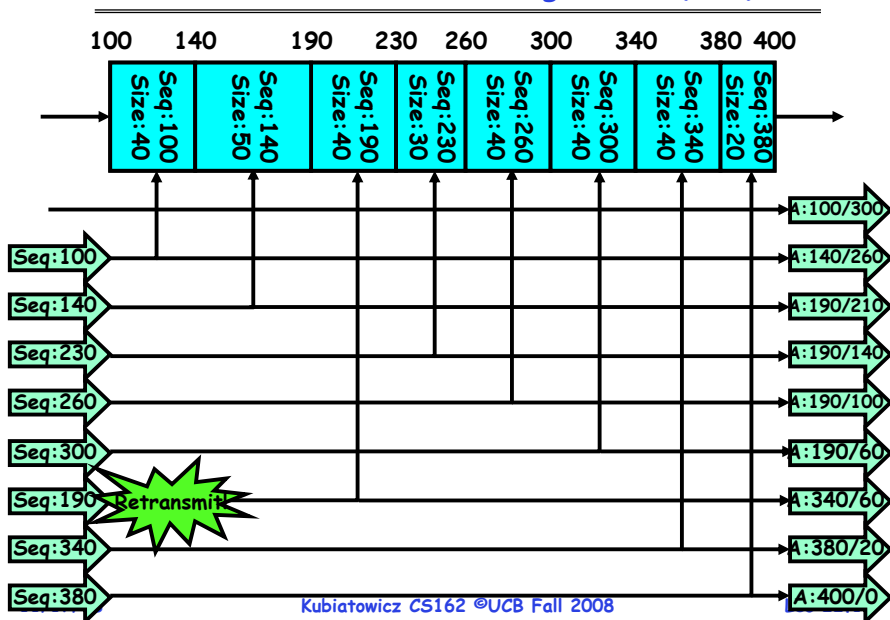
- **Sender has three regions:**
  - Sequence regions
    - » sent and ack'd
    - » Sent and not ack'd
    - » not yet sent
  - Window (colored region) adjusted by sender
- **Receiver has three regions:**
  - Sequence regions
    - » received and ack'd (given to application)
    - » received and buffered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

11/17/08

Kubiatowicz CS162 ©UCB Fall 2008

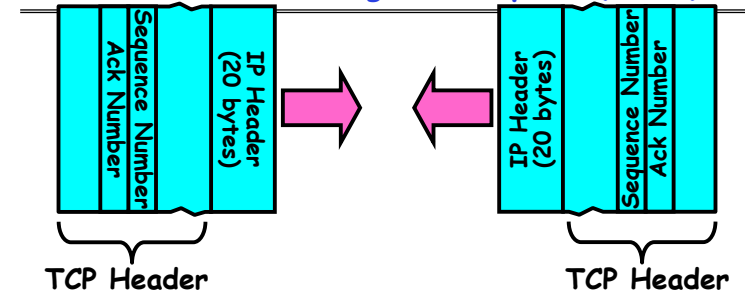
Lec 22.14

## Window-Based Acknowledgements (TCP)



Kubiatowicz CS162 ©UCB Fall 2008

## 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)

11/17/08

Kubiatowicz CS162 ©UCB Fall 2008

Lec 22.16

## 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"

11/17/08

Kubiatowicz CS162 ©UCB Fall 2008

Lec 22.17

## 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

11/17/08

Kubiatowicz CS162 ©UCB Fall 2008

Lec 22.18

## Use of TCP: Sockets

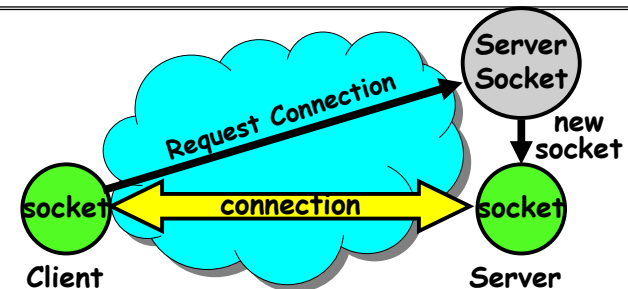
- **Socket**: an abstraction of a network I/O queue
  - Embodies one side of a communication channel
    - » Same interface regardless of location of other end
    - » Could be local machine (called "UNIX socket") or remote machine (called "network socket")
  - First introduced in 4.2 BSD UNIX: big innovation 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

11/17/08

Kubiatowicz CS162 ©UCB Fall 2008

Lec 22.19

## 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

11/17/08

Kubiatowicz CS162 ©UCB Fall 2008

Lec 22.20

## 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();
```

11/17/08

Kubiatowicz CS162 @UCB Fall 2008

Lec 22.21

## 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

11/17/08

Kubiatowicz CS162 @UCB Fall 2008

Lec 22.22

## 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!

11/17/08

Kubiatowicz CS162 @UCB Fall 2008

Lec 22.23

## Messaging for Producer-Consumer Style

- Using send/receive for producer-consumer style:

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

Send Message

```
Consumer:
int buffer[1000];
while(1) {
    receive(buffer, mbox);
    process 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

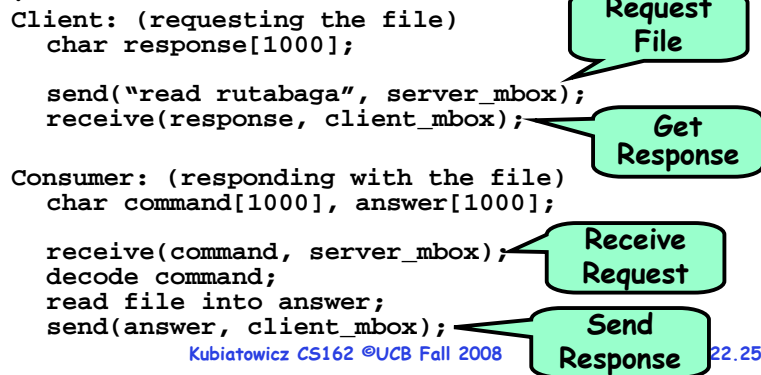
11/17/08

Kubiatowicz CS162 @UCB Fall 2008

Lec 22.24

## 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  $\equiv$  requester, Server  $\equiv$  responder
    - » Server provides "service" (file storage) to the client
- Example: File service



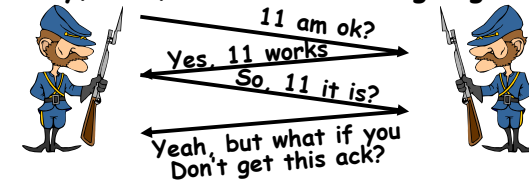
11/17/08

Kubiatowicz CS162 @UCB Fall 2008

22.25

## 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



11/17/08

Kubiatowicz CS162 @UCB Fall 2008

Lec 22.26

- No way to be sure last message gets 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

11/17/08

Kubiatowicz CS162 @UCB Fall 2008

Lec 22.27

## Two phase commit example

- Simple Example: A  $\equiv$  Wells Fargo Bank, B  $\equiv$  Bank of America
  - Phase 1: **Prepare** Phase
    - » A writes "Begin transaction" to log
    - A  $\rightarrow$  B: OK to transfer funds to me?
    - » Not enough funds:
      - B  $\rightarrow$  A: transaction aborted; A writes "Abort" to log
    - » Enough funds:
      - B: Write new account balance & promise to commit to log
      - B  $\rightarrow$  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 that there is a transaction in progress; sends "Abort" to B
- What if B crashes at beginning of phase 2?
  - B comes back up, looks at log; when A sends it "Commit" message, it will say, "oh, ok, commit"

11/17/08

Kubiatowicz CS162 @UCB Fall 2008

Lec 22.28

## Distributed Decision Making Discussion

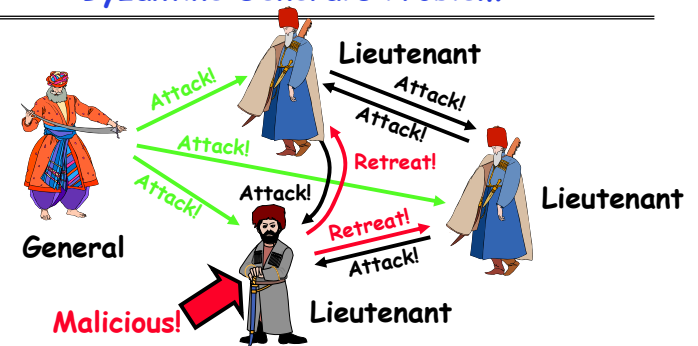
- Why is distributed decision making desirable?
  - Fault Tolerance!
  - A group of machines can come to a decision even if one or more of them fail during the process
    - » Simple failure mode called "failstop" (different modes later)
  - After decision made, result recorded in multiple places
- Undesirable feature of Two-Phase Commit: Blocking
  - One machine can be stalled until another site recovers:
    - » Site B writes "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, check 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 decide to abort, because update may have committed
    - » B is blocked until A comes back
  - A blocked site holds resources (locks on updated items, pages pinned in memory, etc) until learns fate of update
- Alternative: There are alternatives such as "Three Phase Commit" which don't have this blocking problem
- What happens if one or more of the nodes is malicious?
  - **Malicious:** attempting to compromise the decision making

11/17/08

Kubiatowicz CS162 ©UCB Fall 2008

Lec 22.29

## Byzantine General's Problem



- Byzantine General's Problem (n players):
  - One General
  - n-1 Lieutenants
  - Some number of these (f) can be insane or malicious
- The commanding general must send an order to his n-1 lieutenants such that:
  - IC1: All loyal lieutenants obey the same order
  - IC2: If the commanding general is loyal, then all loyal lieutenants obey the order he sends

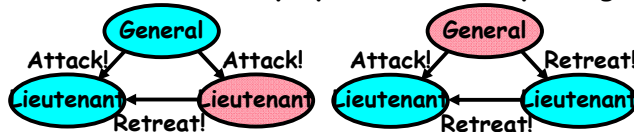
11/17/08

Kubiatowicz CS162 ©UCB Fall 2008

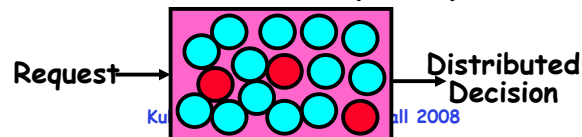
Lec 22.30

## Byzantine General's Problem (con't)

- Impossibility Results:
  - Cannot solve Byzantine General's Problem with  $n=3$  because one malicious player can mess up things



- With  $f$  faults, need  $n > 3f$  to solve problem
- Various algorithms exist to solve problem
  - Original algorithm has #messages exponential in  $n$
  - Newer algorithms have message complexity  $O(n^2)$ 
    - » One from MIT, for instance (Castro and Liskov, 1999)
- Use of BFT (Byzantine Fault Tolerance) algorithm
  - Allow multiple machines to make a coordinated decision even if some subset of them ( $< n/3$ ) are malicious



11/17/08

Kubiatowicz CS162 ©UCB Fall 2008

Lec 22.31

## Conclusion

- **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)
  - Uses window-based acknowledgement protocol
  - Congestion-avoidance dynamically adapts sender window to account for congestion in network
- **Two-phase commit:** distributed decision making
  - First, make sure everyone guarantees that they will commit if asked (prepare)
  - Next, ask everyone to commit
- **Byzantine General's Problem:** distributed decision making with malicious failures
  - One general, n-1 lieutenants: some number of them may be malicious (often "f" of them)
  - All non-malicious lieutenants must come to same decision
  - If general not malicious, lieutenants must follow general
  - Only solvable if  $n \geq 3f+1$

11/17/08

Kubiatowicz CS162 ©UCB Fall 2008

Lec 22.32