CS162
Operating Systems and Systems Programming
Lecture 22

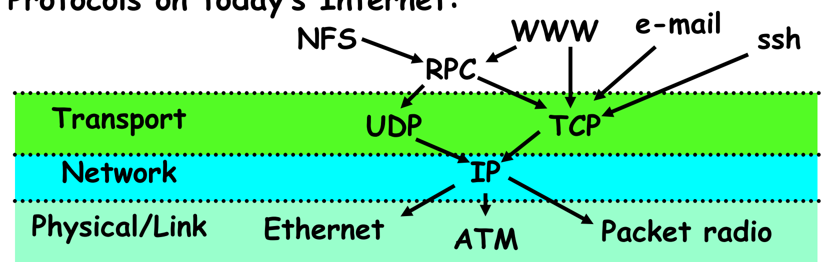
Distributed Systems, Networking, TCP/IP, RPC,VFS

April 15th, 2015

Prof. John Kubiatowicz
http://cs162.eecs.Berkeley.edu

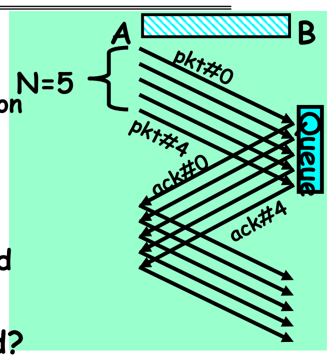
Recall: Network Protocols

- 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:

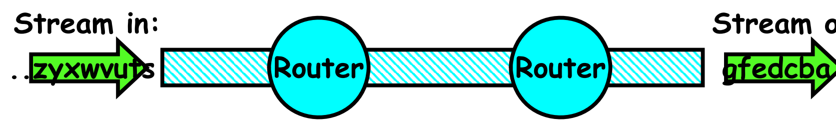


Recall: Window-based acknowledgements

- Windowing protocol (not quite 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
  - Ordering: Packets can be reordered at destination
- 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

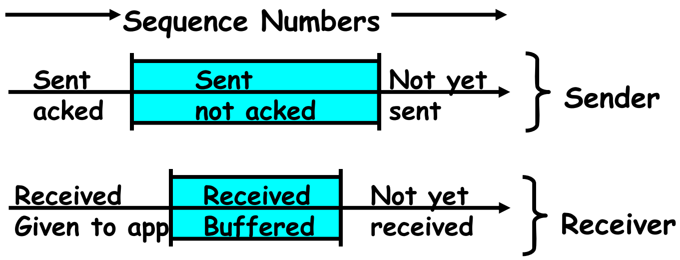


Transmission Control Protocol (TCP)



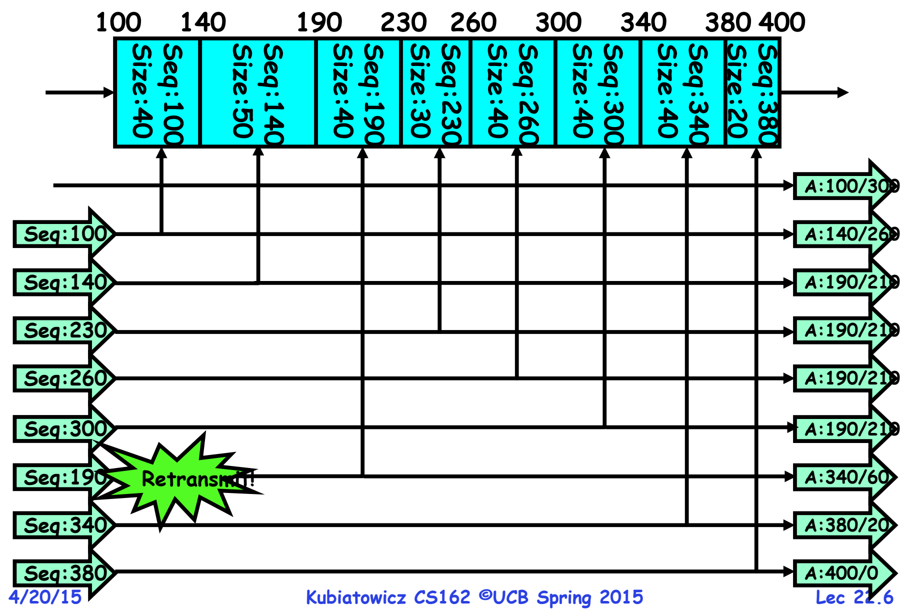
- Transmission Control Protocol (TCP)
  - TCP (IP Protocol 6) layered on top of IP
  - Reliable byte stream between two processes on different machines over Internet (read, write, flush)
- TCP Details
  - Fragments byte stream into packets, hands packets to IP
    - » IP may also fragment by itself
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

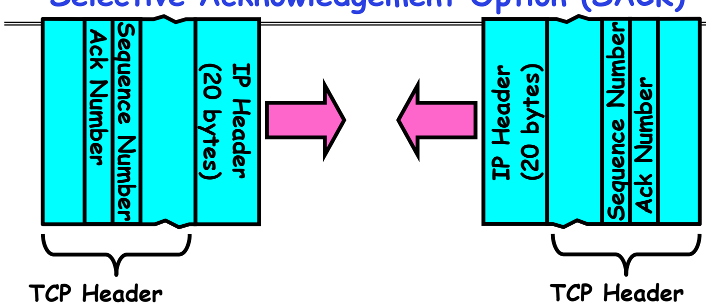


- Sender has three regions:
  - Sequence regions
    - » sent and ack'ed
    - » Sent and not ack'ed
    - » not yet sent
  - Window (colored region) adjusted by sender
- Receiver has three regions:
  - Sequence regions
    - » received and ack'ed (given to application)
    - » received and buffered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

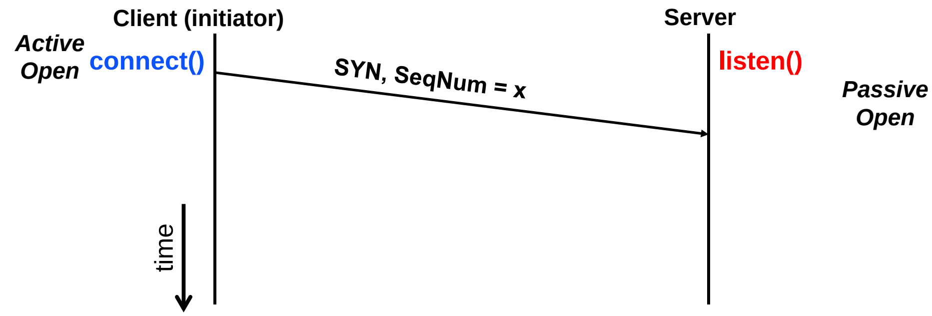
- Congestion
  - How long should timeout be for re-sending messages?
    - » Too long→wastes time if message lost
    - » Too short→retransmit even though ack will arrive shortly
  - Stability problem: more congestion ⇒ ack is delayed ⇒ unnecessary timeout ⇒ more traffic ⇒ more congestion
    - » Closely 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*"

Open Connection: 3-Way Handshaking

- Goal: agree on a set of parameters, i.e., the start sequence number for each side
  - Starting sequence number: sequence of first byte in stream
  - Starting sequence numbers are random

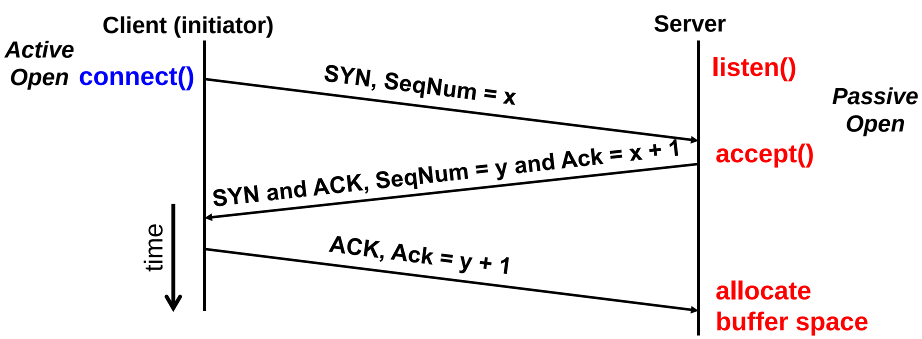
Open Connection: 3-Way Handshaking

- Server waits for new connection calling listen()
- Sender call connect() passing socket which contains server's IP address and port number
  - OS sends a special packet (SYN) containing a proposal for first sequence number, x



Open Connection: 3-Way Handshaking

- If it has enough resources, server calls accept() to accept connection, and sends back a SYN ACK packet containing
  - Client's sequence number incremented by one, (x + 1)
    - » Why is this needed?
  - A sequence number proposal, y, for first byte server will send

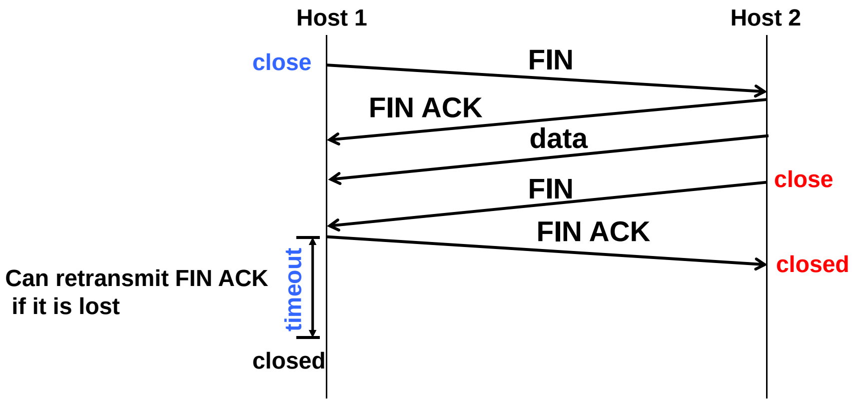


3-Way Handshaking (cont'd)

- Three-way handshake adds 1 RTT delay
- Why do it this way?
  - Congestion control: SYN (40 byte) acts as cheap probe
  - Protects against delayed packets from other connection (would confuse receiver)

Close Connection

- Goal: both sides agree to close the connection
- 4-way connection tear down



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

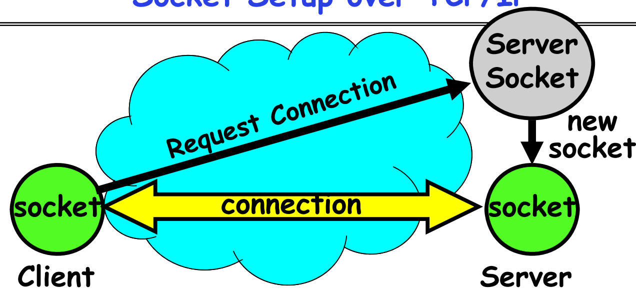
Administrivia

- Midterm II: Wednesday (4/22)
  - Time: 6:30PM – 9:30PM
  - Location: Dwinelle: 145/155
    - » Logins aa-ee, in Dwinelle 145
    - » Logins ef-nk, in Dwinelle 155
  - All topics from Midterm I, up to next Monday, including:
    - » Address Translation/TLBs/Paging
    - » I/O subsystems, Storage Layers, Disks/SSD
    - » Performance and Queueing Theory
    - » File systems
    - » Distributed systems, TCP/IP, RPC
    - » NFS/AFS, Key-Value Store
- Closed book, one page of notes – both sides
- Bring Calculator!

Use of TCP: Sockets

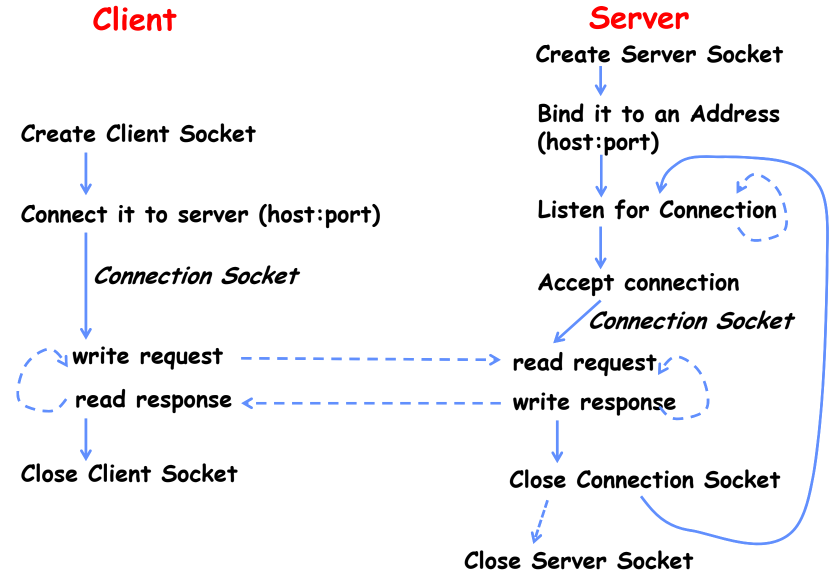
- Socket: an abstraction of a network I/O queue
  - Embodies one side of a communication channel
    - » Same interface regardless of location of other end
    - » Could be local machine (called "UNIX socket") or remote machine (called "network socket")
  - First introduced in 4.2 BSD UNIX: big innovation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



- Server Socket: Listens for new connections
  - Produces new sockets for each unique connection
- Things to remember:
  - Connection involves 5 values:
    [Client Addr, Client Port, Server Addr, Server Port, Protocol]
  - Often, Client Port "randomly" assigned
    - » Done by OS during client socket setup
  - Server Port often "well known"
    - » 80 (web), 443 (secure web), 25 (sendmail), etc
    - » Well-known ports from 0—1023

Recall: Sockets in concept



Recall: Client Protocol

```
char *hostname;
int sockfd, portno;
struct sockaddr_in serv_addr;
struct hostent *server;

server = buildServerAddr(&serv_addr, hostname, portno);

/* Create a TCP socket */
sockfd = socket(AF_INET, SOCK_STREAM, 0)

/* Connect to server on port */
connect(sockfd, (struct sockaddr *) &serv_addr, sizeof(serv_addr)
printf("Connected to %s:%d\n",server->h_name, portno);

/* 
cli

/* 
clo
```

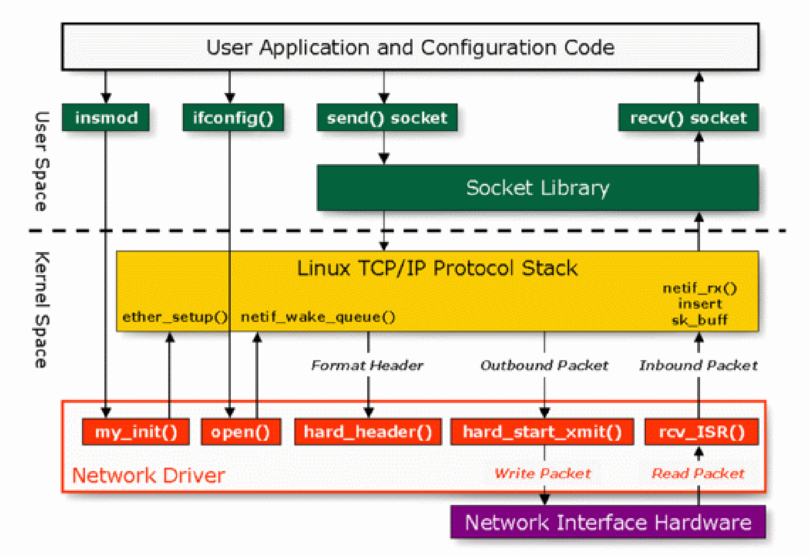
```
SOCK_STREAM
SOCK_DGRAM
SOCK_RAW
SOCK_SEQPACKET
SOCK_RDM
```

```
PF_LOCAL        Host-internal protocols, formerly called PF_UNIX,
PF_UNIX         Host-internal protocols, deprecated, use PF_LOCAL,
PF_INET         Internet version 4 protocols,
PF_ROUTE        Internal Routing protocol,
PF_KEY          Internal key-management function,
PF_INET6        Internet version 6 protocols,
PF_SYSTEM       System domain,
PF_NDRV         Raw access to network device
```

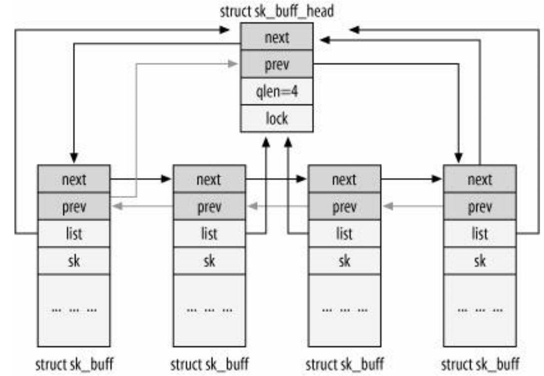

Recall: Server Protocol (v1)

```
/* Create Socket to receive requests*/
lstnsockfd = socket(AF_INET, SOCK_STREAM, 0);

/* Bind socket to port */
bind(lstnsockfd, (struct sockaddr *)&serv_addr,sizeof(serv_addr));
while (1) {
/* Listen for incoming connections */
   listen(lstnsockfd, MAXQUEUE);

/* Accept incoming connection, obtaining a new socket for it */
   consockfd = accept(lstnsockfd, (struct sockaddr *) &cli_addr,
                                    &clilen);

   server(consockfd);

   close(consockfd);
  }
close(lstnsockfd);
```

Linux Network Architecture

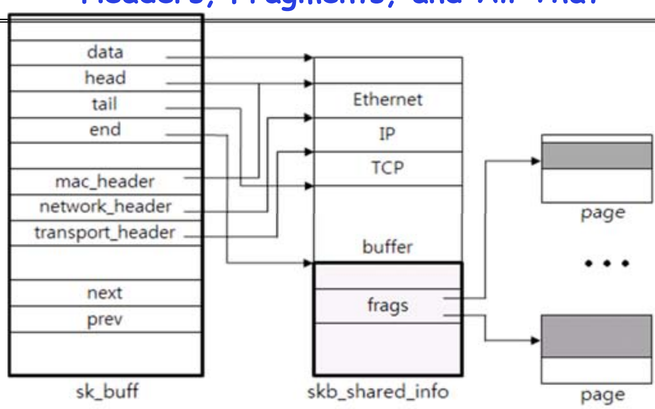


Network Details: sk_buff structure



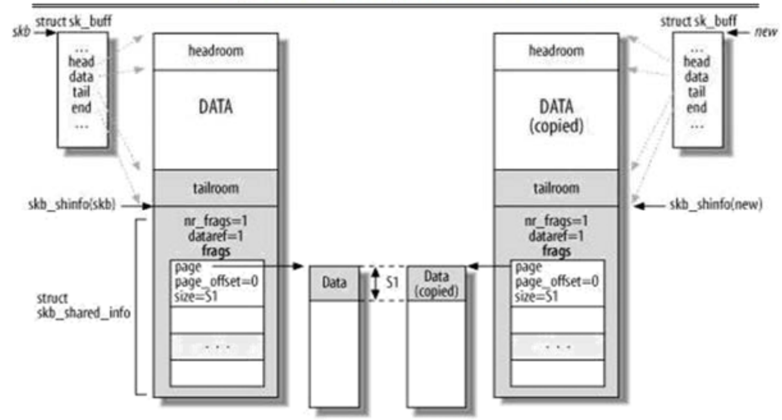
- Socket Buffers: sk_buff structure
  - The I/O buffers of sockets are lists of sk_buff
    - » Pointers to such structures usually called "skb"
  - Complex structures with lots of manipulation routines
  - Packet is linked list of sk_buff structures

Headers, Fragments, and All That



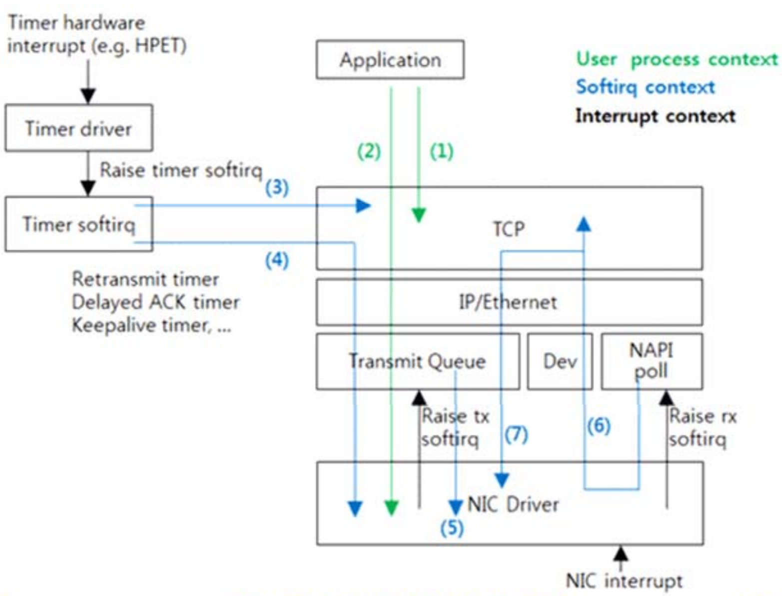
- The "linear region":
  - Space from skb->data to skb->end
  - Actual data from skb->head to skb->tail
  - Header pointers point to parts of packet
- The fragments (in skb_shared_info):
  - Right after skb->end, each fragment has pointer to pages, start of data, and length

Copies, manipulation, etc

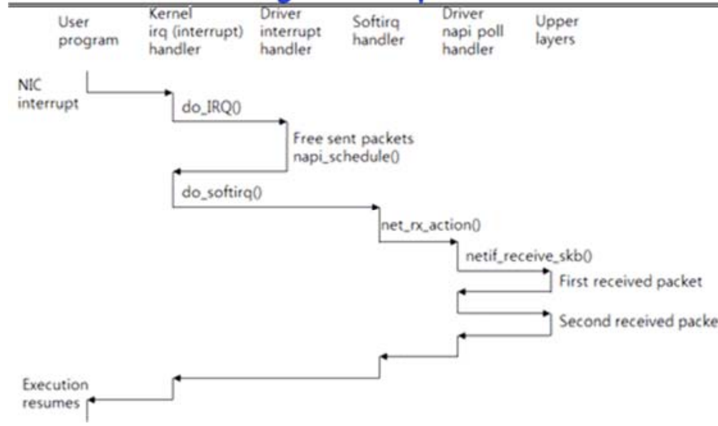


- Lots of sk_buff manipulation functions for:
  - removing and adding headers, merging data, pulling it up into linear region
  - Copying/cloning sk_buff structures

Network Processing Contexts



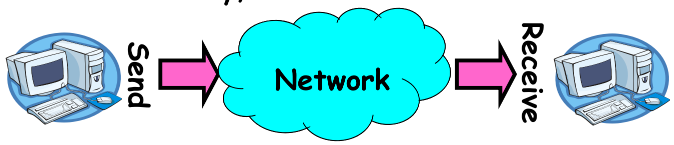
Avoiding Interrupts: NAPI



- New API (NAPI): Use polling to receive packets
  - Only some drivers actually implement this
- Exit hard interrupt context as quickly as possible
  - Do housekeeping and free up sent packets
  - Schedule soft interrupt for further actions
- Soft Interrupts: Handles receiption and delivery

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 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;
   }
```

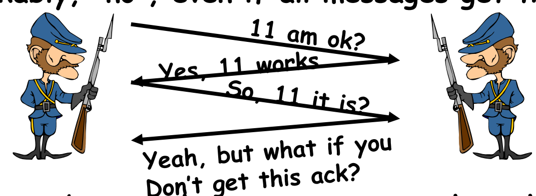
Send Message

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

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



  - No way to be sure last message gets through!

Two Phase (2PC) Commit

- Since we can't solve the General's Paradox (i.e. simultaneous action), let's solve a related problem
  - Distributed transaction: Two or more machines agree to do something, or not do it, atomically
- Two Phase Commit: High-level problem statement
  - If no node fails and all nodes are ready to commit, then all nodes COMMIT
  - Otherwise ABORT at all nodes
- Developed by Turing award winner Jim Gray (first Berkeley CS PhD, 1969)

2PC Algorithm

- One coordinator
- N workers (replicas)
- High level algorithm description
  - Coordinator asks all workers if they can commit
  - If all workers reply "VOTE-COMMIT", then coordinator broadcasts "GLOBAL-COMMIT",

    Otherwise coordinator broadcasts "GLOBAL-ABORT"
  - Workers obey the GLOBAL messages
- Use a persistent, stable log on each machine to keep track of what you are doing
  - If a machine crashes, when it wakes up it first checks its log to recover state of world at time of crash

Detailed Algorithm

Coordinator Algorithm

Coordinator sends **VOTE-REQ** to all workers

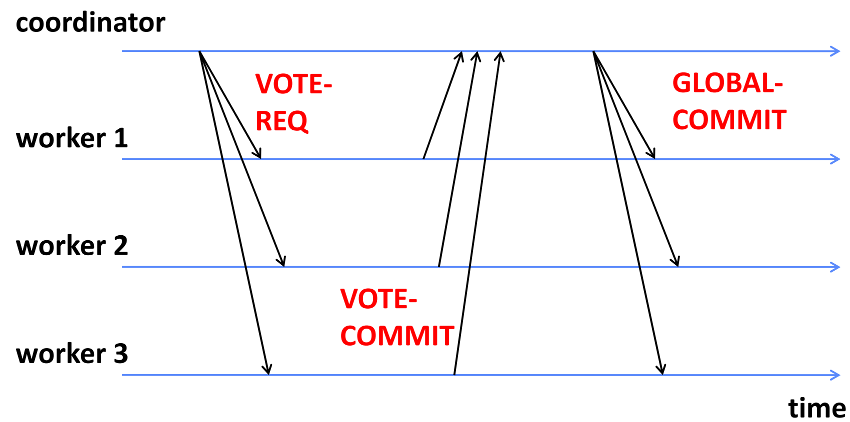
- If receive VOTE-COMMIT from all N workers, send GLOBAL-COMMIT to all workers
- If doesn't receive VOTE-COMMIT from all N workers, send GLOBAL-ABORT to all workers

Worker Algorithm

- Wait for **VOTE-REQ** from coordinator
- If ready, send **VOTE-COMMIT** to coordinator
- If not ready, send **VOTE-ABORT** to coordinator
  - And immediately abort

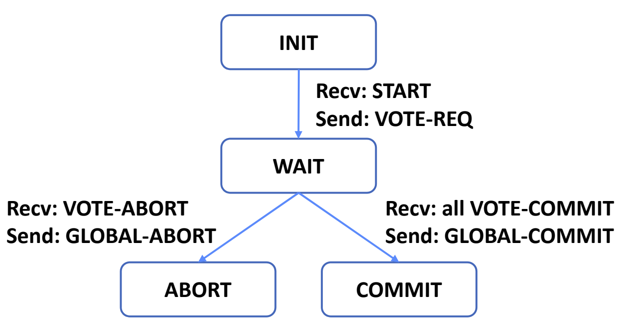
- If receive GLOBAL-COMMIT then commit
- If receive GLOBAL-ABORT then abort

Failure Free Example Execution

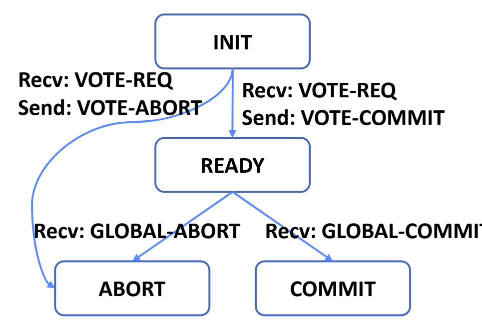


State Machine of Coordinator

- Coordinator implements simple state machine:

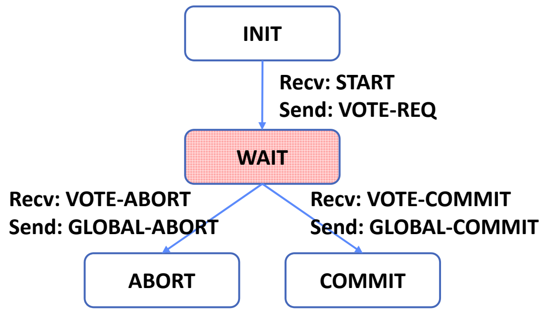


State Machine of Workers

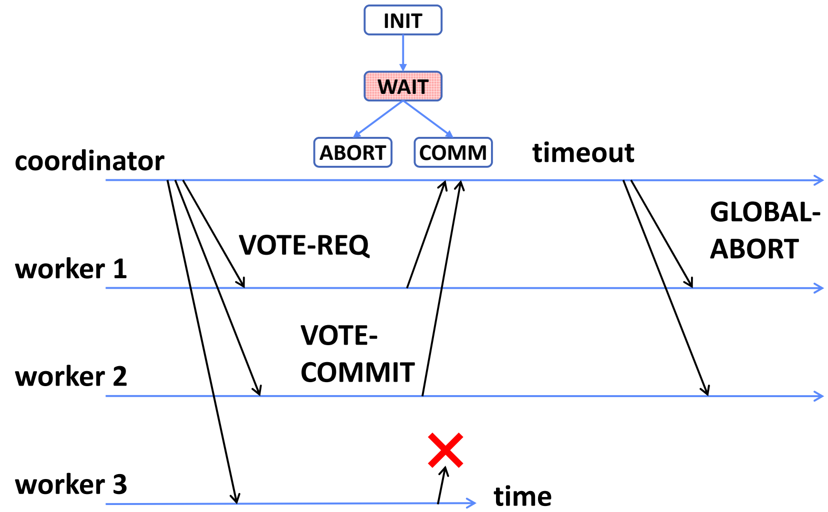


Dealing with Worker Failures

- How to deal with worker failures?
  - Failure only affects states in which the node is waiting for messages
  - Coordinator only waits for votes in "WAIT" state
  - In WAIT, if doesn't receive
  - N votes, it times out and sends
  - GLOBAL-ABORT

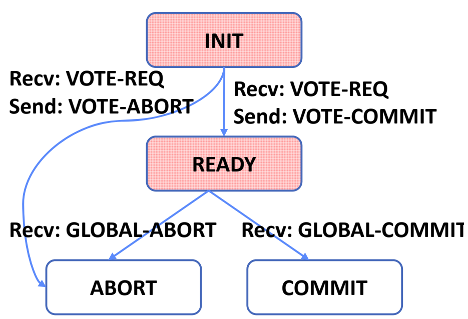


Example of Worker Failure

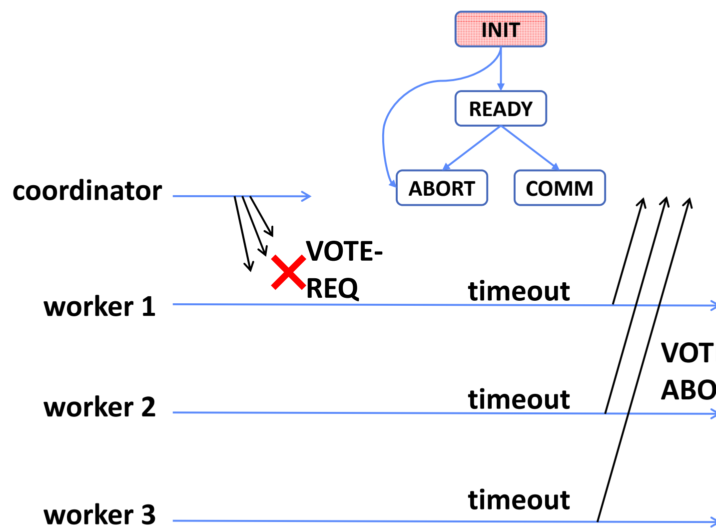


Dealing with Coordinator Failure

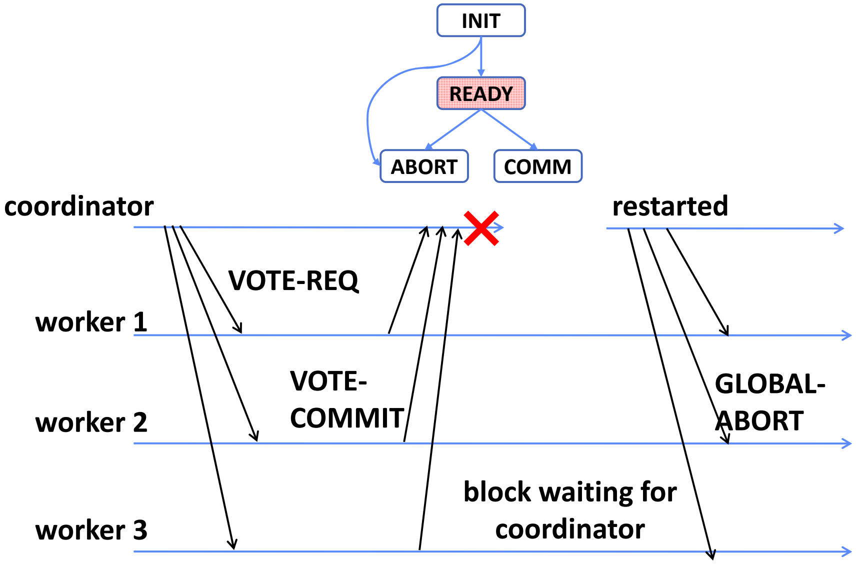
- How to deal with coordinator failures?
  - worker waits for VOTE-REQ in INIT
    - » Worker can time out and abort (coordinator handles it)
  - worker waits for GLOBAL-* message in READY
    - » If coordinator fails, workers must BLOCK waiting for coordinator to recover and send GLOBAL_* message



Example of Coordinator Failure #1



Example of Coordinator Failure #2

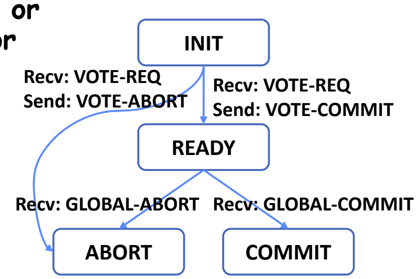


Durability

- All nodes use stable storage* to store which state they are in
- Upon recovery, it can restore state and resume:
  - Coordinator aborts in INIT, WAIT, or ABORT
  - Coordinator commits in COMMIT
  - Worker aborts in INIT, ABORT
  - Worker commits in COMMIT
  - Worker asks Coordinator in READY
- * - stable storage is non-volatile storage (e.g. backed by disk) that guarantees atomic writes.

Blocking for Coordinator to Recover

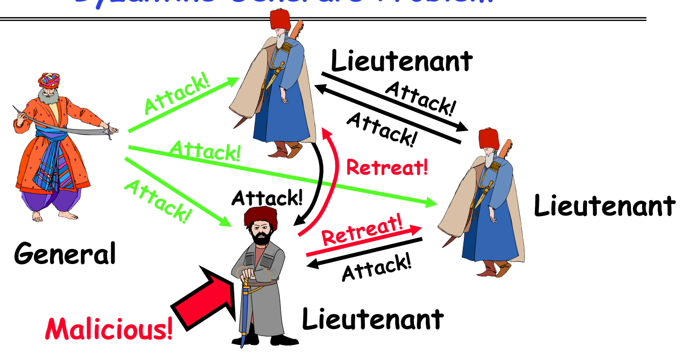
- A worker waiting for global decision can ask fellow workers about their state
  - If another worker is in ABORT or COMMIT state then coordinator must have sent GLOBAL-*
    » Thus, worker can safely abort or commit, respectively
  - If another worker is still in INIT state then both workers can decide to abort
  - If all workers are in ready, need to BLOCK (don't know if coordinator wanted to abort or commit)



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
- PAXOS: An alternative used by GOOGLE and others that does not have this blocking problem
- What happens if one or more of the nodes is malicious?
  - Malicious: attempting to compromise the decision making

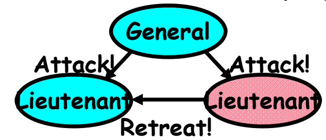
Byzantine General's Problem

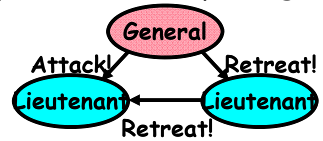


- Byazantine General's Problem (n players):
  - One General
  - n-1 Lieutenants
  - Some number of these (f) can be insane or malicious
- The commanding general must send an order to his n-1 lieutenants such that:
  - IC1: All loyal lieutenants obey the same order
  - IC2: If the commanding general is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

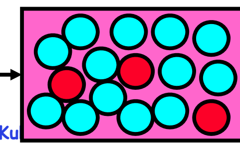
Byzantine General's Problem (con't)

- 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



Summary

- 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 $n \geq 3f+1$
- Remote Procedure Call (RPC): Call procedure on remote machine
  - Provides same interface as procedure
  - Automatic packing and unpacking of arguments without user programming (in stub)