CS162
Operating Systems and Systems Programming
Lecture 22

TCP Flow Control, Distributed Decision Making, RPC

November 13th, 2017

Prof. Ion Stoica

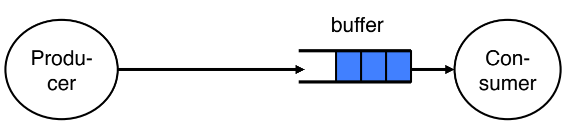
http://cs162.eecs.Berkeley.edu

Goals of Today's Lecture

- TCP flow control
- Two-Phase Commit

Flow Control

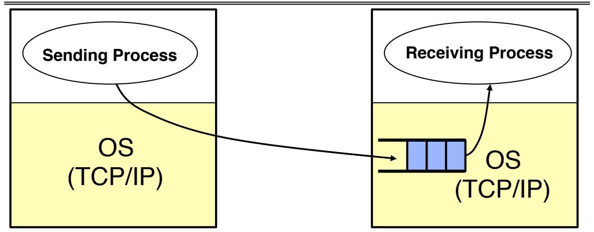
- Recall: Flow control ensures a fast sender does not overwhelm a slow receiver
- Example: Producer-consumer with bounded buffer (Lecture 5)
  - A buffer between producer and consumer
  - Producer puts items into buffer as long as buffer **not full**
  - Consumer consumes items from buffer



TCP Flow Control

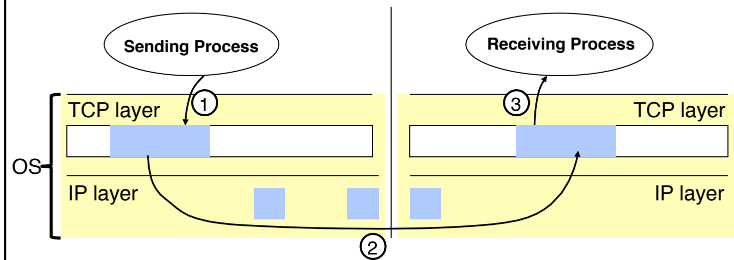
- TCP: sliding window protocol at byte (not packet) level
- Receiver tells sender how many more bytes it can receive without overflowing its buffer (i.e., AdvertisedWindow)
- The ack(nowledgement) contains sequence number N of next byte the receiver expects, i.e., receiver has received all bytes in sequence up to and including N-1

TCP Flow Control



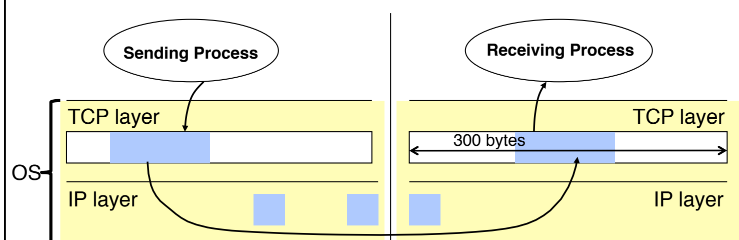
- TCP/IP implemented by OS (Kernel)
  - Cannot do context switching on sending/receiving every packet
    - » At 10Gbps, it takes 1.2 usec to send an 1500 bytes, and 80nsec to send an 100 byte packet
- Need buffers to match …
  - sending app with sending TCP
  - receiving TCP with receiving app

TCP Flow Control



- Three pairs of producer-consumer's
  - ① sending process → sending TCP
  - ② Sending TCP → receiving TCP
  - ③ receiving TCP → receiving process

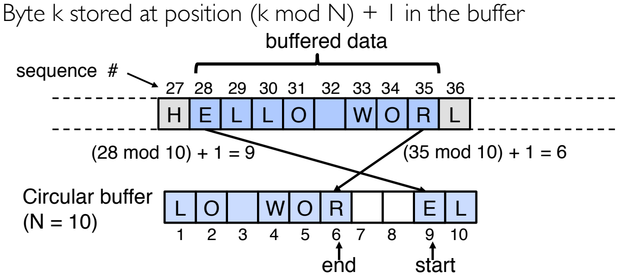
TCP Flow Control



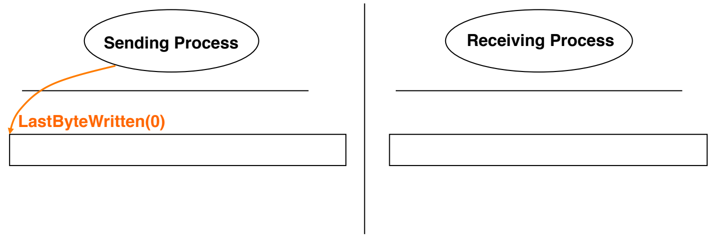
- Example assumptions:
  - Maximum IP packet size = 100 bytes
  - Size of the receiving buffer (MaxRcvBuf) = 300 bytes
- Recall, ack indicates the next expected byte in-sequence, not the last received byte
- Use circular buffers

Circular Buffer

- Assume
  - A buffer of size N
  - A stream of bytes, where bytes have increasing sequence numbers
    - » Think of stream as an unbounded array of bytes and of sequence number as indexes in this array
- Buffer stores at most N consecutive bytes from the stream
- Byte k stored at position (k mod N) + 1 in the buffer

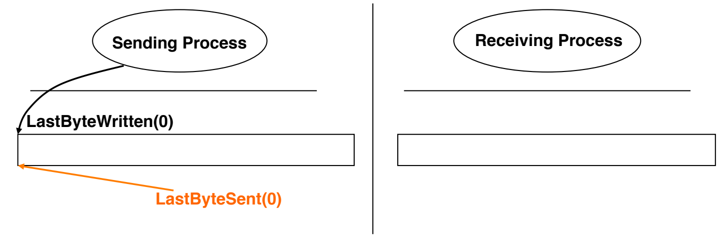


TCP Flow Control



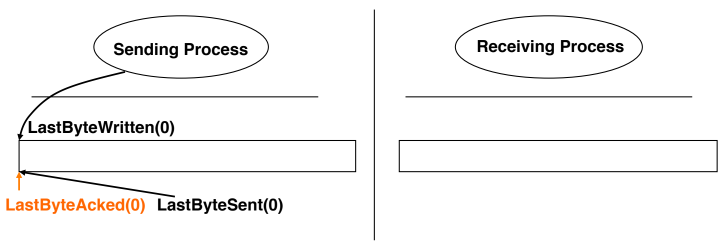
- LastByteWritten: last byte written by sending process

TCP Flow Control



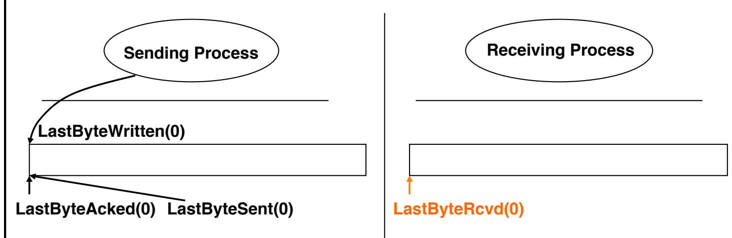
- LastByteWritten: last byte written by sending process
- LastByteSent: last byte sent by sender to receiver

TCP Flow Control



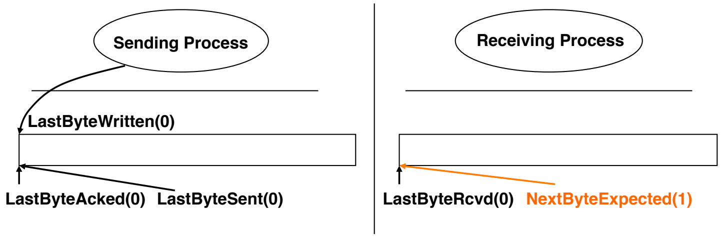
- LastByteWritten: last byte written by sending process
- LastByteSent: last byte sent by sender to receiver
- LastByteAcked: last ack received by sender from receiver

TCP Flow Control



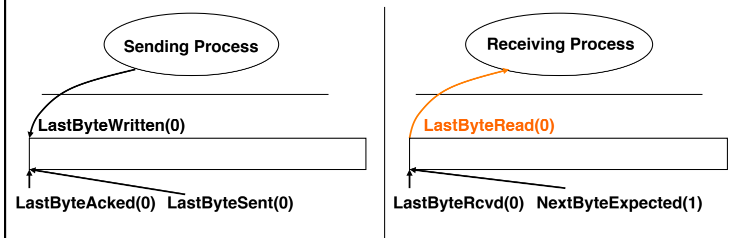
- LastByteWritten: last byte written by sending process
- LastByteSent: last byte sent by sender to receiver
- LastByteAcked: last ack received by sender from receiver
- LastByteRcvd: last byte received by receiver from sender

TCP Flow Control



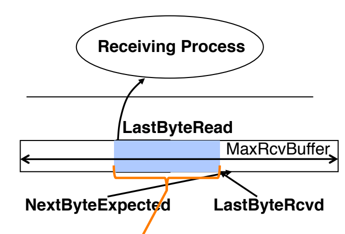
- LastByteWritten: last byte written by sending process
- LastByteSent: last byte sent by sender to receiver
- LastByteAcked: last ack received by sender from receiver
- LastByteRcvd: last byte received by receiver from sender
- NextByteExpected: last in-sequence byte expected by receiver

TCP Flow Control



- LastByteWritten: last byte written by sending process
- LastByteSent: last byte sent by sender to receiver
- LastByteAcked: last ack received by sender from receiver
- LastByteRcvd: last byte received by receiver from sender
- NextByteExpected: last in-sequence byte expected by receiver
- LastByteRead: last byte read by the receiving process

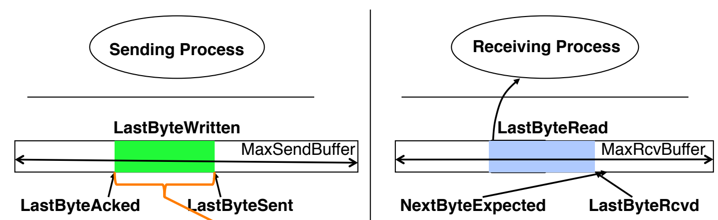
TCP Flow Control



- AdvertisedWindow: number of bytes TCP receiver can receive

AdvertisedWindow = MaxRcvBuffer – (LastByteRcvd – LastByteRead)

TCP Flow Control



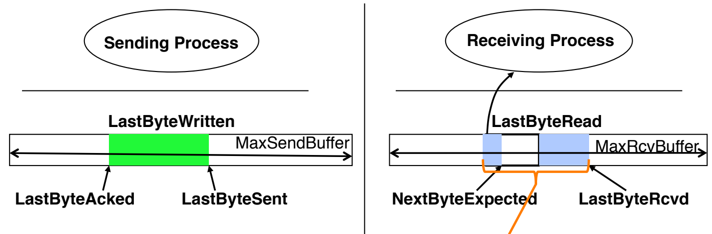
- AdvertisedWindow: number of bytes TCP receiver can receive

AdvertisedWindow = MaxRcvBuffer – (LastByteRcvd – LastByteRead)

- SenderWindow: number of bytes TCP sender can send

SenderWindow = AdvertisedWindow – (LastByteSent – LastByteAcked)

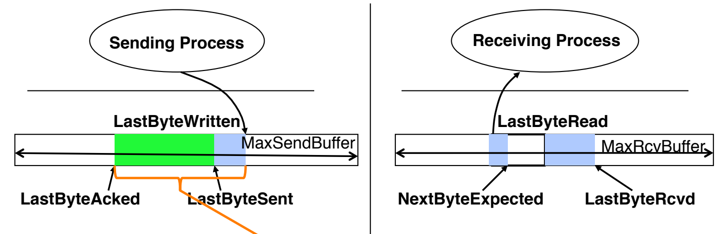
TCP Flow Control



- Still true if receiver missed data….

AdvertisedWindow = MaxRcvBuffer – (LastByteRcvd – LastByteRead)

TCP Flow Control



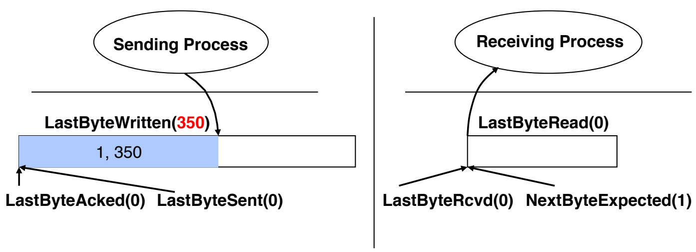
- Still true if receiver missed data….

AdvertisedWindow = MaxRcvBuffer – (LastByteRcvd – LastByteRead)

- WriteWindow: number of bytes sending process can write

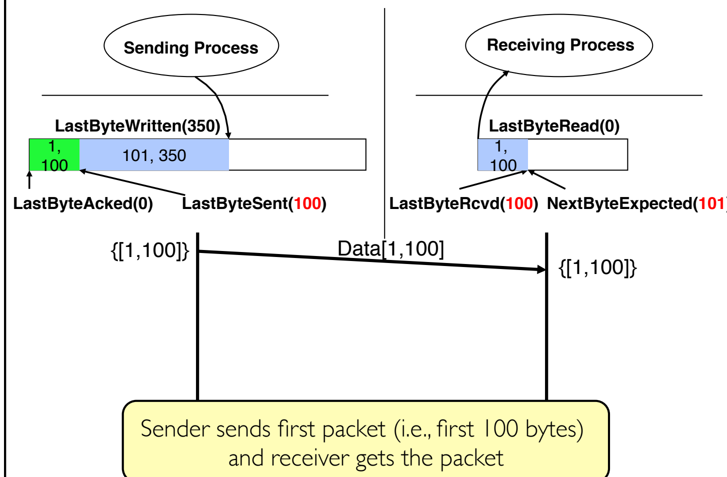
WriteWindow = MaxSendBuffer – (LastByteWritten – LastByteAcked)

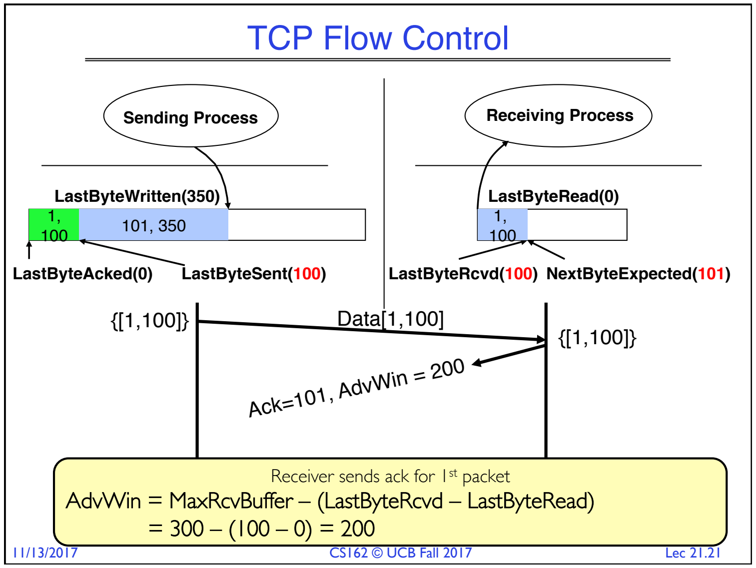
TCP Flow Control

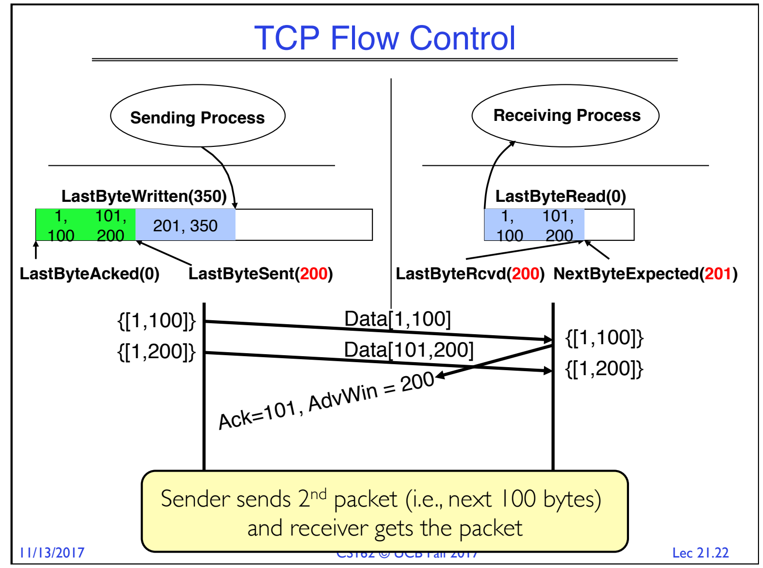


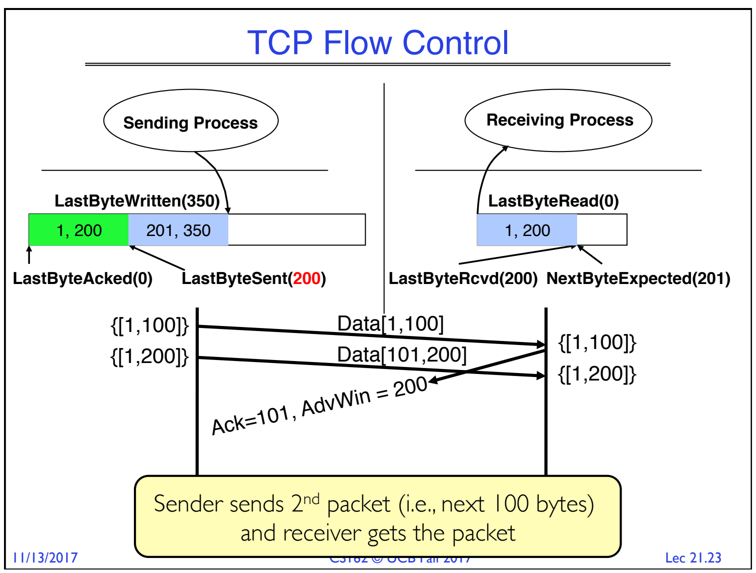
- Sending app sends 350 bytes
- Recall:
  - We assume IP only accepts packets no larger than 100 bytes
  - MaxRcvBuf = 300 bytes, so initial Advertised Window = 300 byets

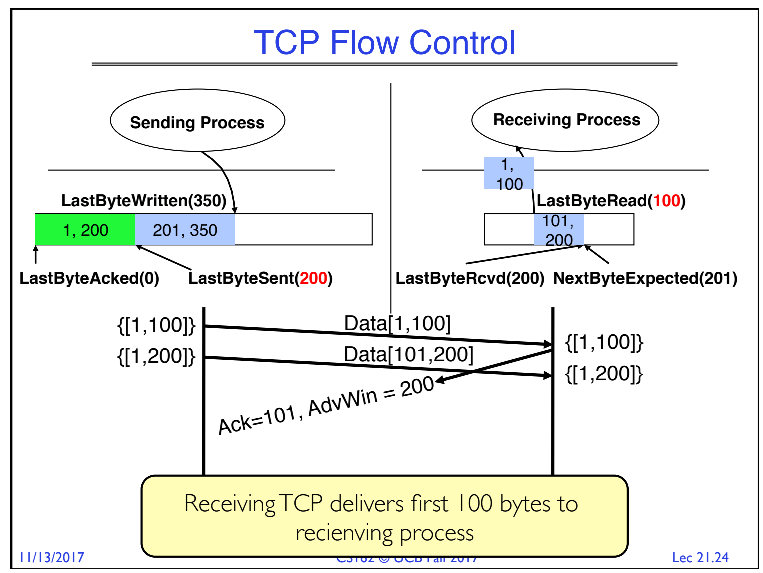
TCP Flow Control



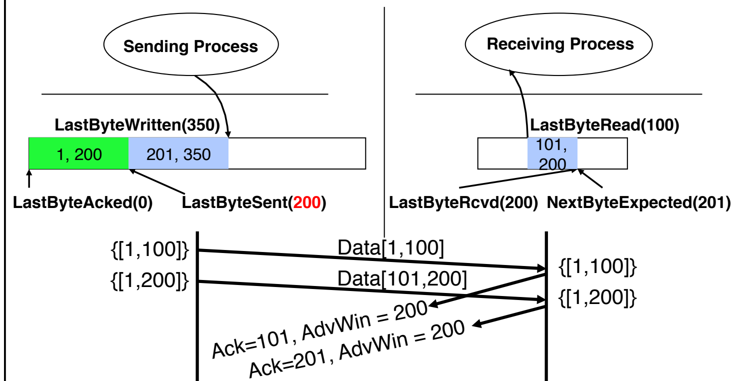

TCP Flow Control
Sending Process
Receiving Process
LastByteWritten(350)
1, 100
101, 350
LastByteAcked(0)
LastByteSent(100)
LastByteRead(0)
1, 100
LastByteRcvd(100)
NextByteExpected(101)
{[1,100]}
Data[1,100]
{[1,100]}
Ack=101, AdvWin = 200
Receiver sends ack for 1st packet
AdvWin = MaxRcvBuffer – (LastByteRcvd – LastByteRead)
= 300 – (100 – 0) = 200
11/13/2017
CS162 © UCB Fall 2017
Lec 21.21


TCP Flow Control
Sending Process
Receiving Process
LastByteWritten(350)
1, 100
101, 200
201, 350
LastByteAcked(0)
LastByteSent(200)
LastByteRead(0)
1, 100
101, 200
LastByteRcvd(200)
NextByteExpected(201)
{[1,100]}
{[1,200]}
Data[1,100]
Data[101,200]
{[1,100]}
{[1,200]}
Ack=101, AdvWin = 200
Sender sends 2nd packet (i.e., next 100 bytes)
and receiver gets the packet
11/13/2017
Lec 21.22


TCP Flow Control
Sending Process
Receiving Process
LastByteWritten(350)
1, 200
201, 350
LastByteAcked(0)
LastByteSent(200)
LastByteRead(0)
1, 200
LastByteRcvd(200)
NextByteExpected(201)
{[1,100]}
{[1,200]}
Data[1,100]
Data[101,200]
{[1,100]}
{[1,200]}
Ack=101, AdvWin = 200
Sender sends 2nd packet (i.e., next 100 bytes)
and receiver gets the packet
11/13/2017
Lec 21.23


TCP Flow Control
Sending Process
Receiving Process
1, 100
LastByteWritten(350)
1, 200
201, 350
LastByteAcked(0)
LastByteSent(200)
LastByteRead(100)
101, 200
LastByteRcvd(200)
NextByteExpected(201)
{[1,100]}
{[1,200]}
Data[1,100]
Data[101,200]
{[1,100]}
{[1,200]}
Ack=101, AdvWin = 200
Receiving TCP delivers first 100 bytes to
recienving process
11/13/2017
Lec 21.24

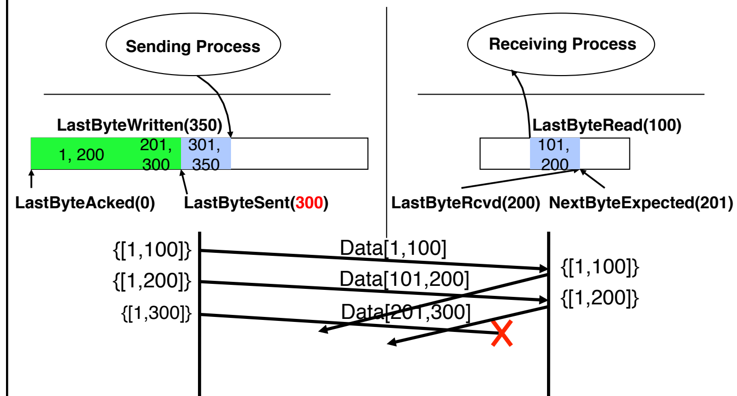
TCP Flow Control



Receiver sends ack for 2nd packet

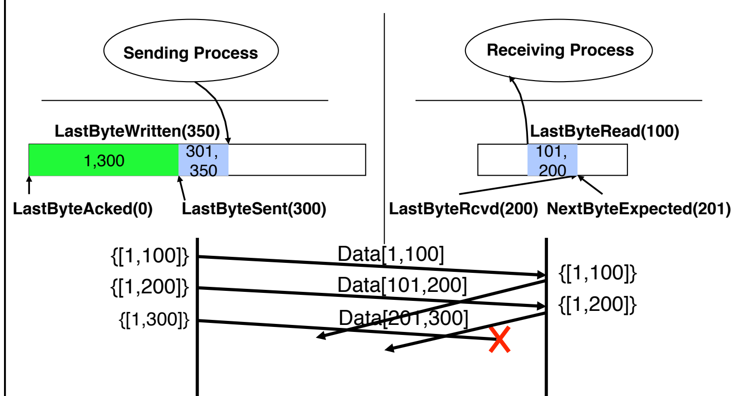
AdvWin = MaxRcvBuffer – (LastByteRcvd – LastByteRead)
= 300 – (200 – 100) = 200

TCP Flow Control



Sender sends 3rd packet (i.e., next 100 bytes) and the packet is lost

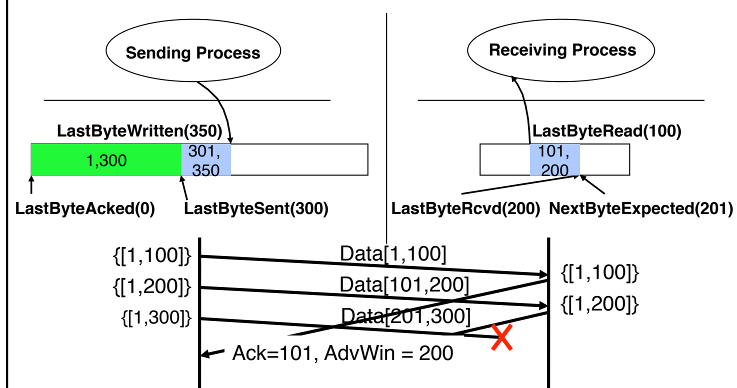
TCP Flow Control



Sender stops sending as window full

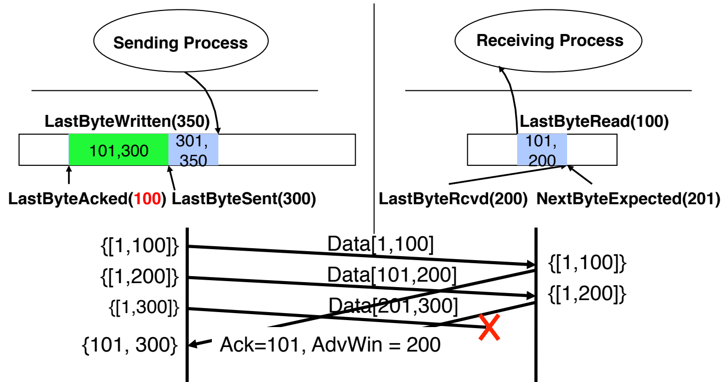
SndWin = AdvWin – (LastByteSent – LastByteAcked)
= 300 – (300 – 0) = 0

TCP Flow Control



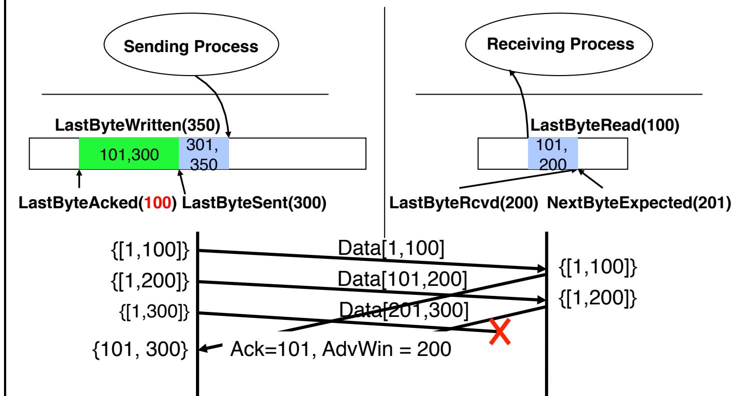
- Sender gets ack for 1st packet
- AdvWin = 200

TCP Flow Control



- Ack for 1st packet (ack indicates next byte expected by receiver)
- Receiver no longer needs first 100 bytes

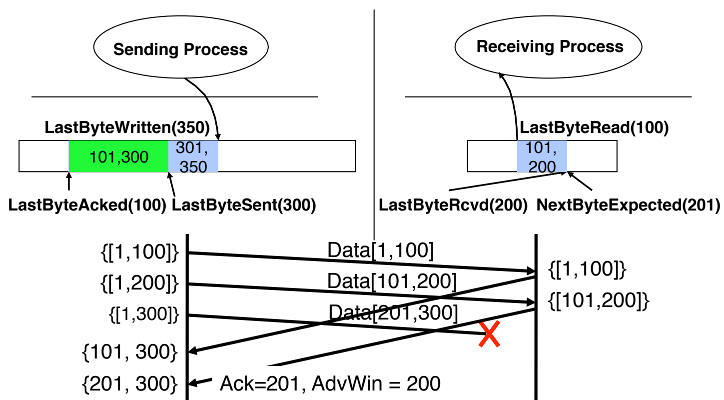
TCP Flow Control



Sender still cannot send as window full

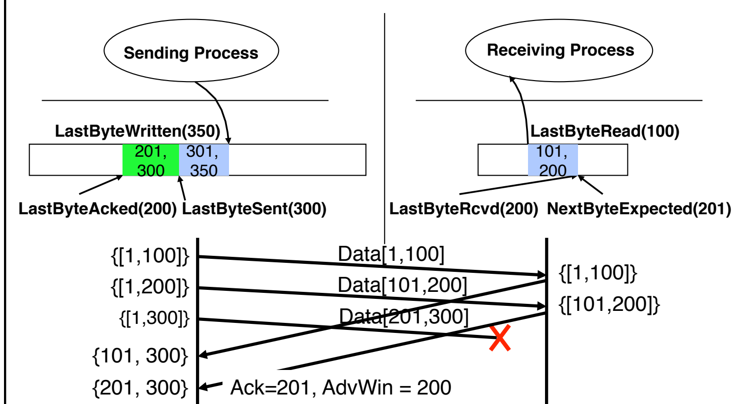
SndWin = AdvWin – (LastByteSent – LastByteAcked)
= 200 – (300 – 100) = 0

TCP Flow Control



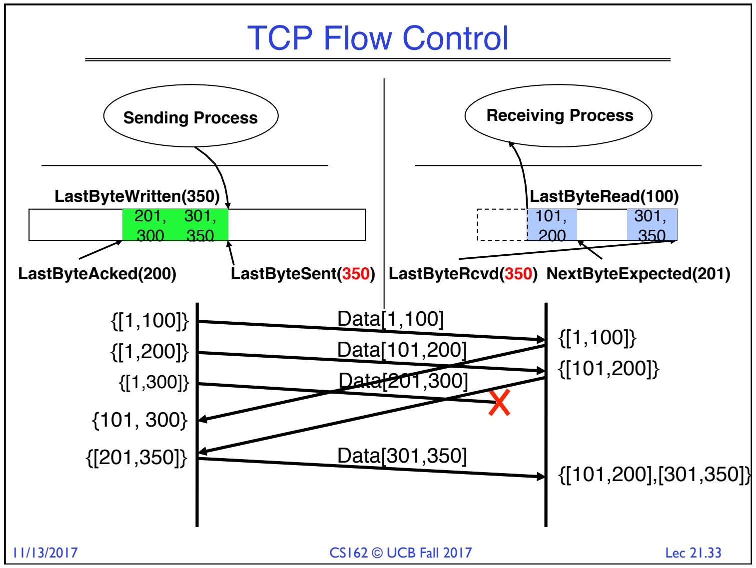
- Receiver gets ack for 2nd packet
- AdvWin = 200 bytes

TCP Flow Control

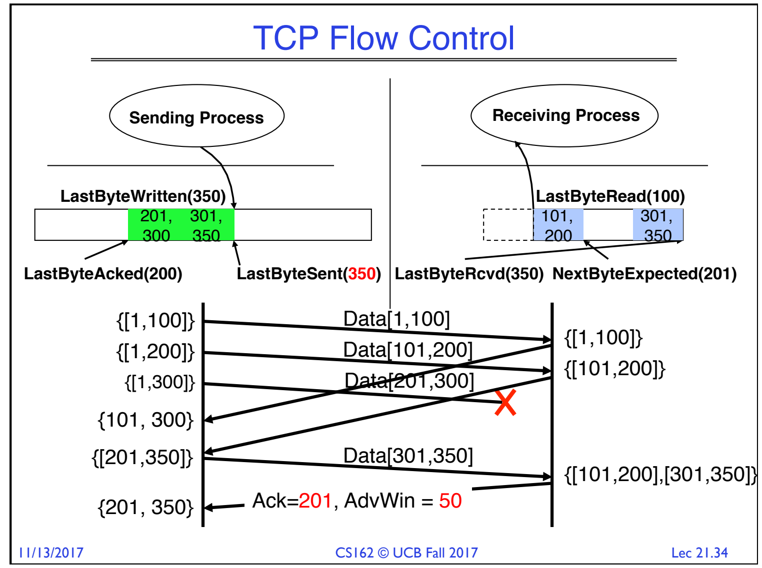


Sender can now send new data!

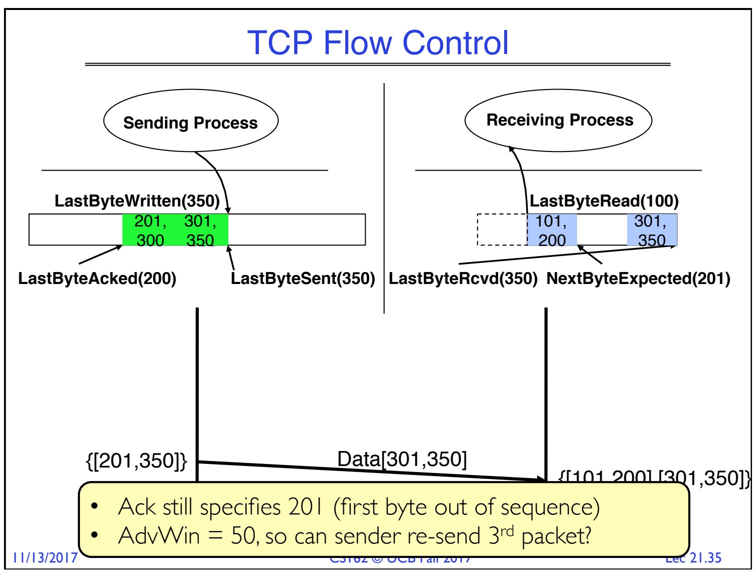
SndWin = AdvWin – (LasByteSent – LastByteAcked) = 100



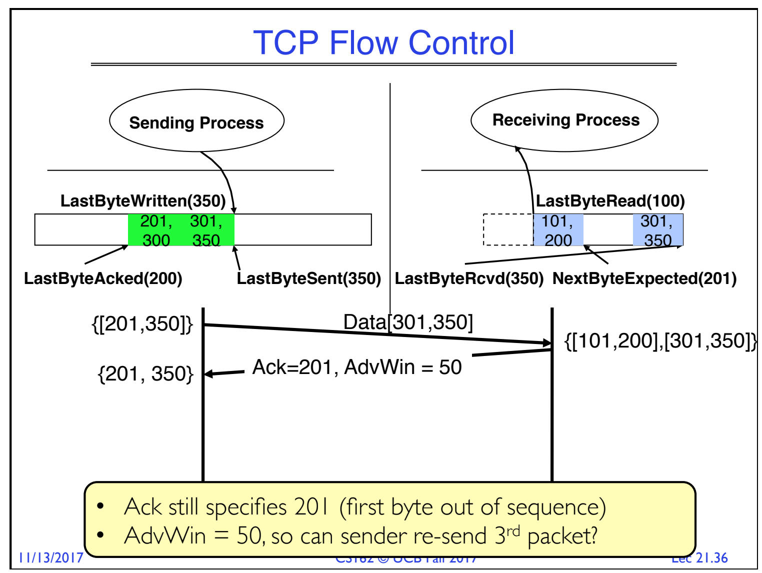
TCP Flow Control
Sending Process
Receiving Process
LastByteWritten(350)
201, 300
301, 350
LastByteAcked(200)
LastByteSent(350)
LastByteRead(100)
101, 200
301, 350
LastByteRcvd(350)
NextByteExpected(201)
{[1,100]}
{[1,200]}
{[1,300]}
{101, 300}
{[201,350]}
Data[1,100]
Data[101,200]
Data[201,300]
Data[301,350]
{[1,100]}
{[101,200]}
{[101,200],[301,350]}
11/13/2017
CS162 © UCB Fall 2017
Lec 21.33



TCP Flow Control
Sending Process
Receiving Process
LastByteWritten(350)
201, 300
301, 350
LastByteAcked(200)
LastByteSent(350)
LastByteRead(100)
101, 200
301, 350
LastByteRcvd(350)
NextByteExpected(201)
{[1,100]}
{[1,200]}
{[1,300]}
{101, 300}
{[201,350]}
{201, 350}
Data[1,100]
Data[101,200]
Data[201,300]
Data[301,350]
Ack=201, AdvWin = 50
{[1,100]}
{[101,200]}
{[101,200],[301,350]}
11/13/2017
CS162 © UCB Fall 2017
Lec 21.34

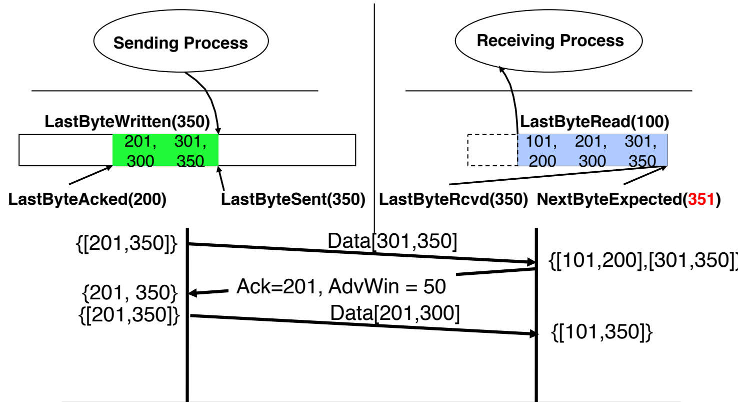


TCP Flow Control
Sending Process
Receiving Process
LastByteWritten(350)
201, 300
301, 350
LastByteAcked(200)
LastByteSent(350)
LastByteRead(100)
101, 200
301, 350
LastByteRcvd(350)
NextByteExpected(201)
{[201,350]}
Data[301,350]
{[101,200],[301,350]}
• Ack still specifies 201 (first byte out of sequence)
• AdvWin = 50, so can sender re-send 3rd packet?
11/13/2017
Lec 21.35



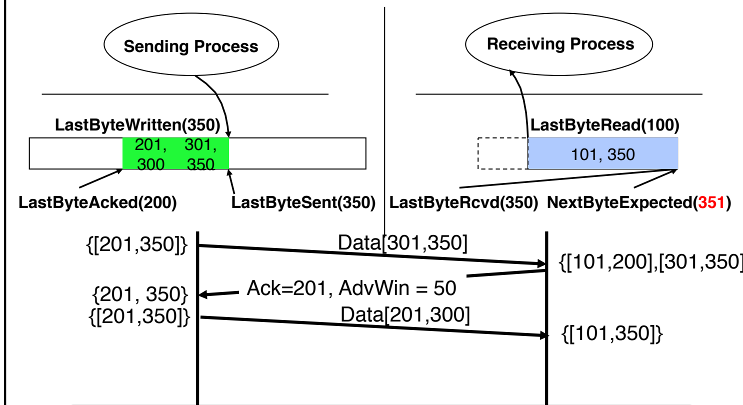
TCP Flow Control
Sending Process
Receiving Process
LastByteWritten(350)
201, 300
301, 350
LastByteAcked(200)
LastByteSent(350)
LastByteRead(100)
101, 200
301, 350
LastByteRcvd(350)
NextByteExpected(201)
{[201,350]}
{201, 350}
Data[301,350]
Ack=201, AdvWin = 50
{[101,200],[301,350]}
• Ack still specifies 201 (first byte out of sequence)
• AdvWin = 50, so can sender re-send 3rd packet?
11/13/2017
Lec 21.36

TCP Flow Control



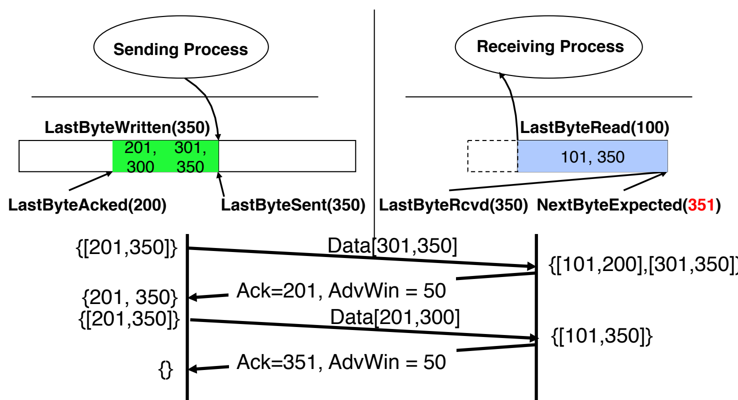
Yes! Sender can re-send 2nd packet since it's in existing window – won't cause receiver window to grow

TCP Flow Control



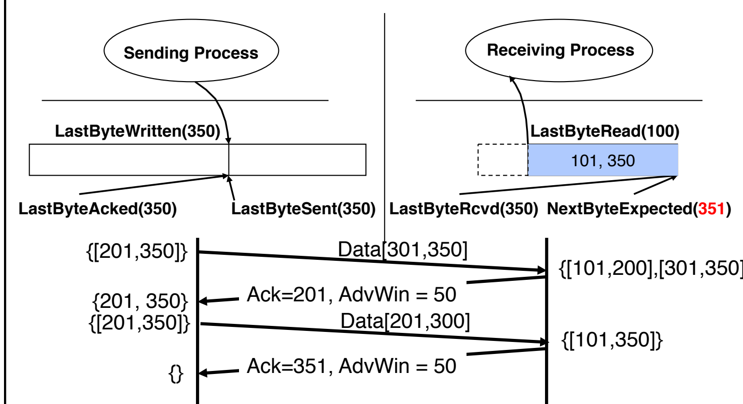
Yes! Sender can re-send 2nd packet since it's in existing window – won't cause receiver window to grow

TCP Flow Control



- Sender gets 3rd packet and sends Ack for 351
- AdvWin = 50

TCP Flow Control



Sender DONE with sending all bytes!

Discussion

- Why not have a huge buffer at the receiver (memory is cheap!)?
- Sending window (SndWnd) also depends on network congestion
  - **Congestion control**: ensure that a fast sender doesn't overwhelm a router in the network (discussed in detail in cs168)
- In practice there is another set of buffers in the protocol stack, at the **link layer** (i.e., Network Interface Card)

Administrivia

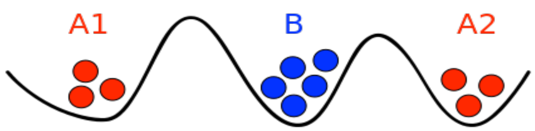
- Midterm 3 coming up on **Wen 11/29 6:30-8PM**
  - All topics up to and including Lecture 24
    - » Focus will be on Lectures 17 – 24 and associated readings, and Projects 3
    - » But expect 20-30% questions from materials from Lectures 1-16
  - Closed book
  - 2 sides hand-written notes both sides

BREAK

Goals of Today's Lecture

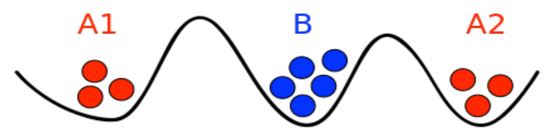
- TCP flow control
- Two-Phase Commit

General's Paradox

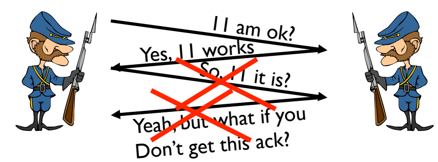


- Constraints of problem:
  - Two generals, on separate mountains
  - Can only communicate via messengers
  - Messengers can be captured
- Problem: need to coordinate attack
  - If they attack at different times, they all die
  - If they attack at same time, they win
- Named after Custer, who died at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

General's Paradox



- 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 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 protocol: Developed by Turing Award winner Jim Gray (first Berkeley CS PhD, 1969)

Two-Phase Commit Protocol

- Persistent stable log on each machine: 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 used to guarantee that all machines either commit or don't

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

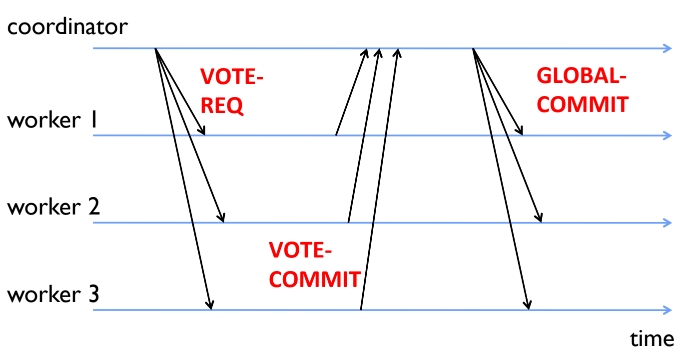
- If receive **VOTE-COMMIT** from all N workers, send **GLOBAL-COMMIT** to all workers
- If doesn't receive **VOTE-COMMIT** from all N workers, send **GLOBAL-ABORT** to all workers

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

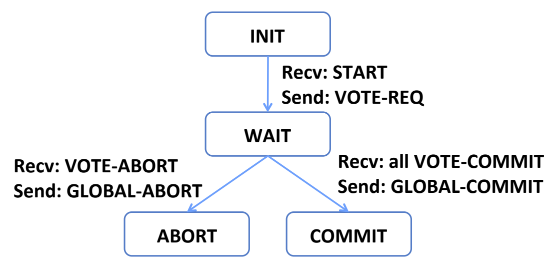
- If receive **GLOBAL-COMMIT** then commit
- If receive **GLOBAL-ABORT** then abort

Failure Free Example Execution

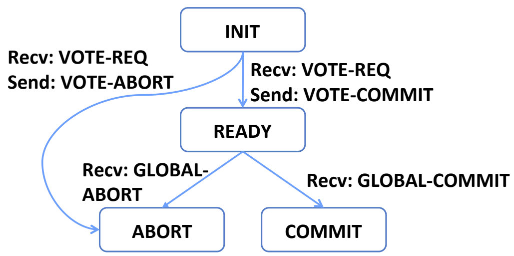


State Machine of Coordinator

- Coordinator implements simple state machine:

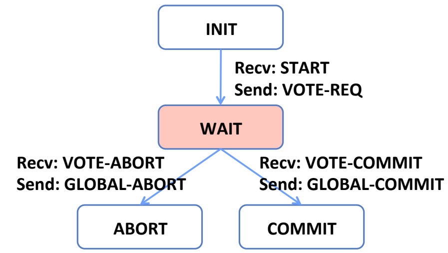


State Machine of Workers

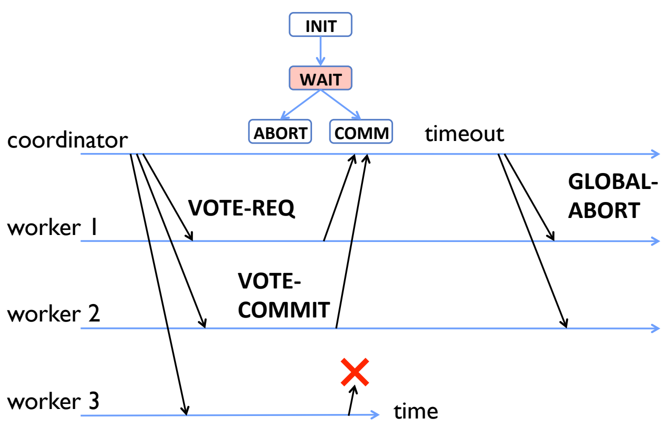


Dealing with Worker Failures

- Failure only affects states in which the coordinator 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

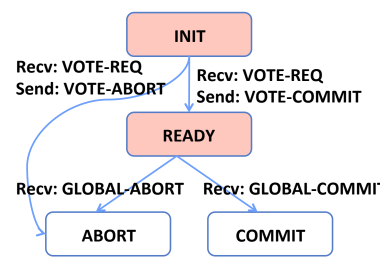


Example of Worker Failure

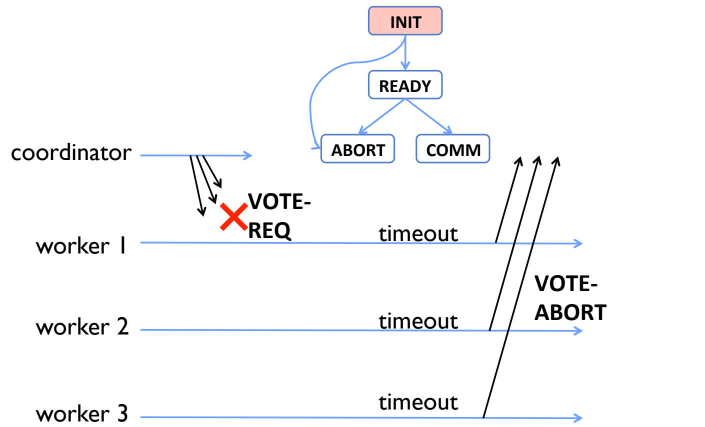


Dealing with Coordinator Failure

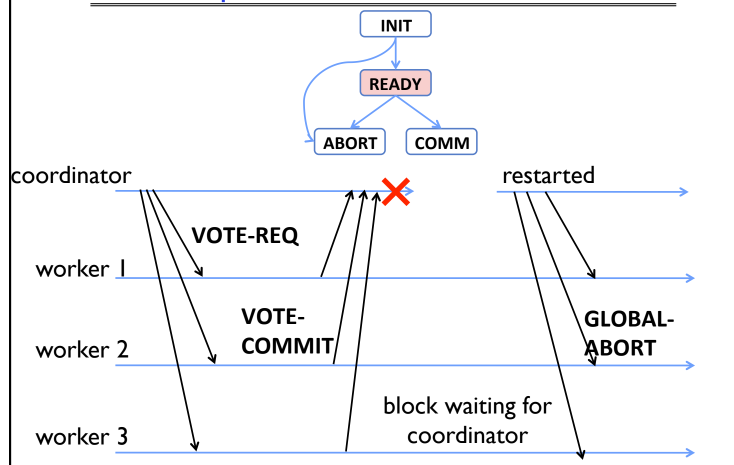
- Worker waits for VOTE-REQ in INIT
  - Worker can time out and abort (coordinator handles it)
- Worker waits for GLOBAL-* message in READY
  - If coordinator fails, workers must **BLOCK** waiting for coordinator to recover and send GLOBAL_* message



Example of Coordinator Failure #1



Example of Coordinator Failure #2

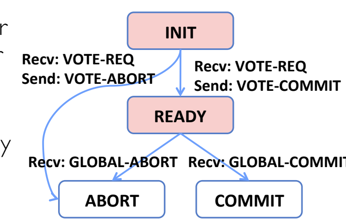


Durability

- All nodes use stable storage to store current state
  - stable storage is non-volatile storage (e.g. backed by disk) that guarantees atomic writes.
- 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

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 (1/2)

- 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

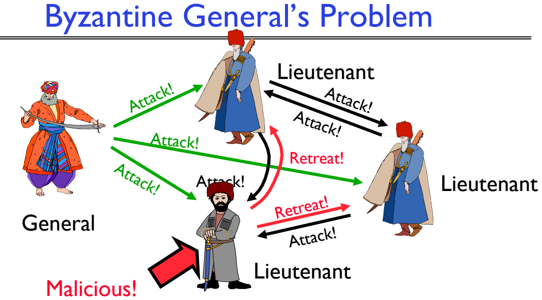
Distributed Decision Making Discussion (2/2)

- 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

- PAXOS: An alternative used by Google and others that does not have this blocking problem
  - Develop by Leslie Lamport (Turing Award Winner)
- What happens if one or more of the nodes is malicious?
  - Malicious: attempting to compromise the decision making

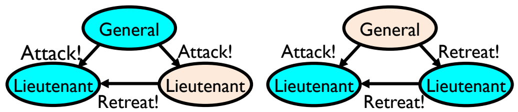
Byzantine General's Problem



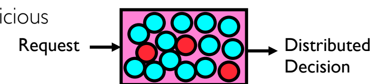
- Byazantine General's Problem (n players):
  - One General and n-1 Lieutenants
  - Some number of these (*f*) can be insane or malicious
- The commanding general must send an order to his n-1 lieutenants such that the following Integrity Constraints apply:
  - IC1: All loyal lieutenants obey the same 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 flow control
  - Ensures a fast sender does not overwhelm a slow receiver
  - Receiver tells sender how many more bytes it can receive without overflowing its buffer (i.e., AdvertisedWindow)
  - The ack(nowledgement) contains sequence number N of next byte the receiver expects, i.e., receiver has received all bytes in sequence up to and including N-1
- Two-phase commit: distributed decision making
  - First, make sure everyone guarantees they will commit if asked (prepare)
  - Next, ask everyone to commit