EECS 482
Introduction to Operating Systems

Winter 2018

Harsha V. Madhyastha

The Design and Implementation of a Log-Structured File System

MENDEL ROSENBLUM and JOHN K. OUSTERHOUT
University of California at Berkeley

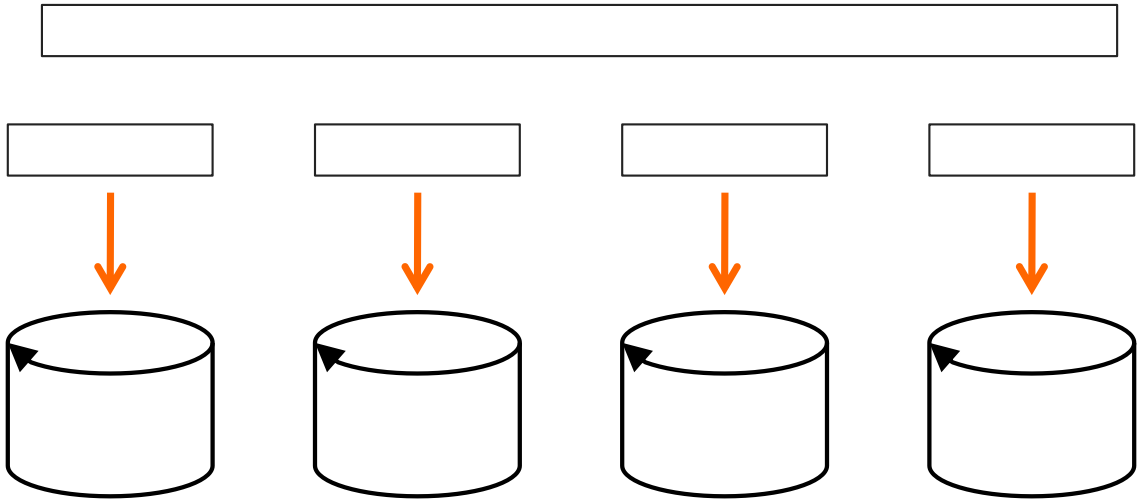
This paper presents a new technique for disk storage management called a *log-structured file system*. A log-structured file system writes all modifications to disk sequentially in a log-like structure, thereby speeding up both file writing and crash recovery. The log is the only structure on disk; it contains indexing information so that files can be read back from the log efficiently.

RAID

- Redundant Array of Inexpensive Disks (RAID)
  - Sits in between hardware and the file system

- Idea: Use many disks in parallel to increase storage bandwidth, improve reliability
  - Files are striped across disks
  - Each stripe portion is read/written in parallel
  - Bandwidth increases with more disks

RAID



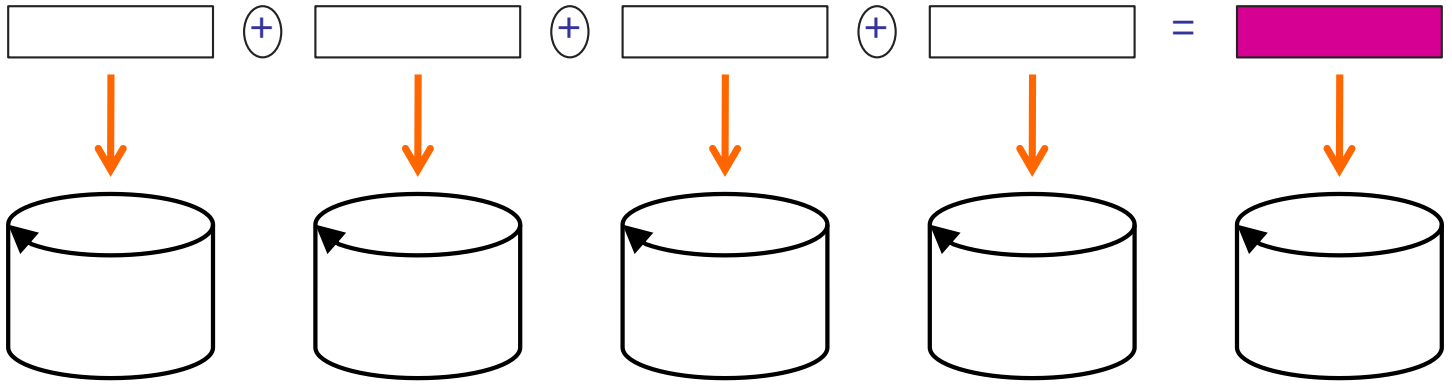
RAID Challenges

- Small files (small writes less than a full stripe)
  - Need to read entire stripe, update with small write, then write entire stripe out to disks

- Reliability
  - More disks increase chance of failure (MTBF)
  - Example:
    - » Say 1 disk has 10% chance of failing in one year
    - » With 10 disks, chance of *any* 1 disk failing in one year is 1 – (1 – 0.1)^10 = 65%!

RAID with parity

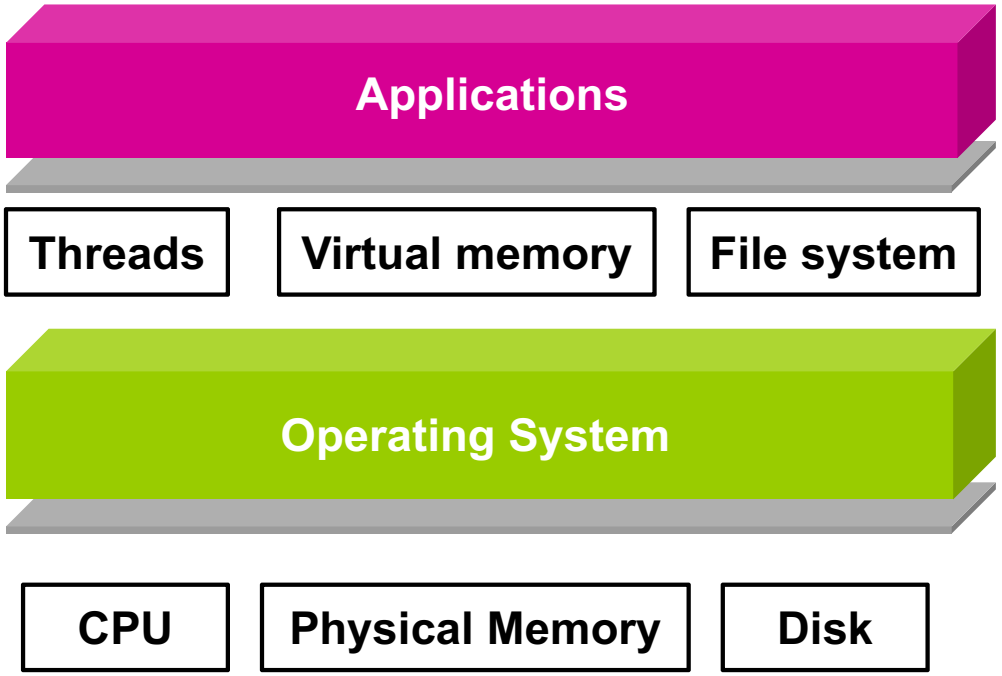
- Improve reliability by storing redundant parity
  - In each stripe, use one block to store parity data
    - » XOR of all data blocks in stripe
  - Can recover any data block from all others + parity
  - Introduces overhead, but disks are “inexpensive”



RAID Levels

- RAID 0: Striping
  - Good performance but no reliability
- RAID 1: Mirroring
  - Maintain full copy of all data
  - Good read performance, but 100% overhead for storage and writes
- RAID 5: Floating parity
  - Parity blocks for different stripes written to different disks
  - No single parity disk → no bottleneck at that disk

OS Abstractions



- Next few lectures:
  - Abstraction of network (EECS 489)
  - Distributed systems (EECS 491)

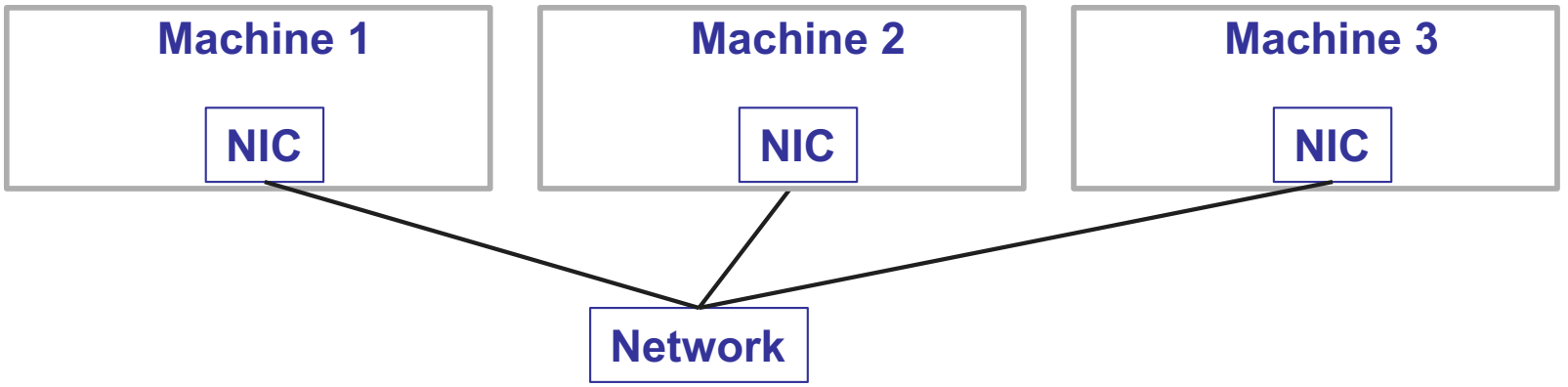
OS abstraction of network

- Hardware reality:
  - One network interface card shared by all processes
    → Machine-to-machine communication
  - Network is unreliable
  - Unordered delivery of finite-sized messages

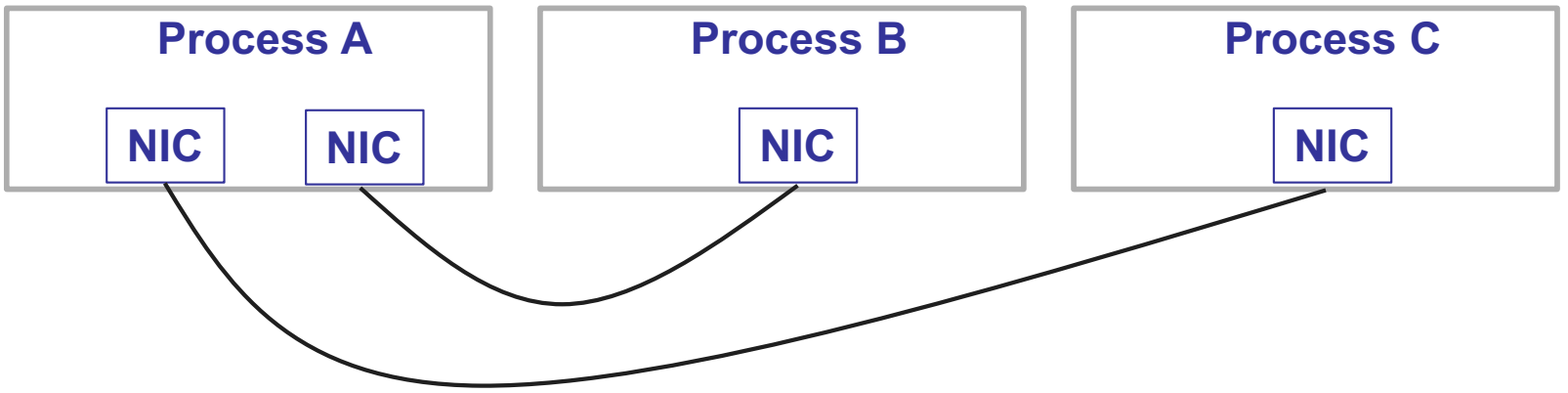
- OS abstraction?
  - Process-to-process communication
  - Reliable and ordered delivery of byte stream

OS abstraction of network

- Hardware reality



- OS abstraction

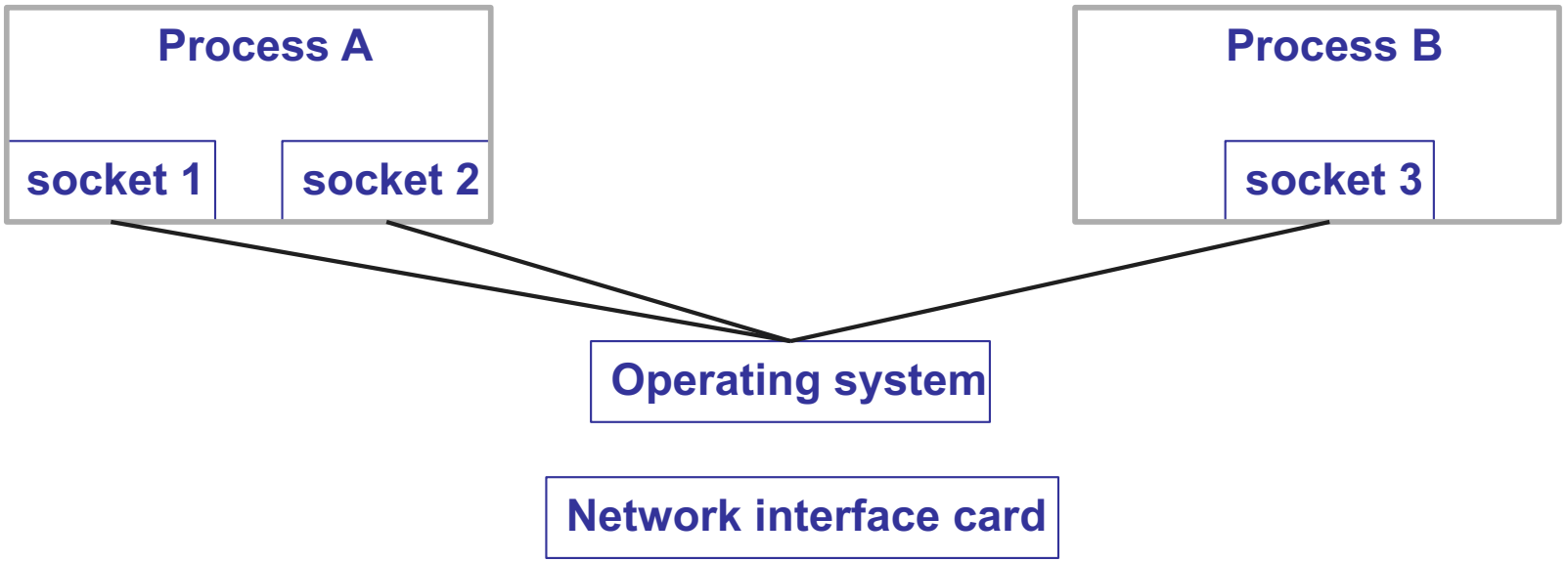


Inter-machine to inter-process

- Every process thinks it has its own:
  - Multiprocessor (threads)
  - Memory (address space)
  - Network interface cards (sockets)

- Socket
  - Virtual network interface card
  - Endpoint for communication
  - NIC named by MAC address; socket named by "port number" (via bind)
  - Programming interface: BSD sockets (discussion)

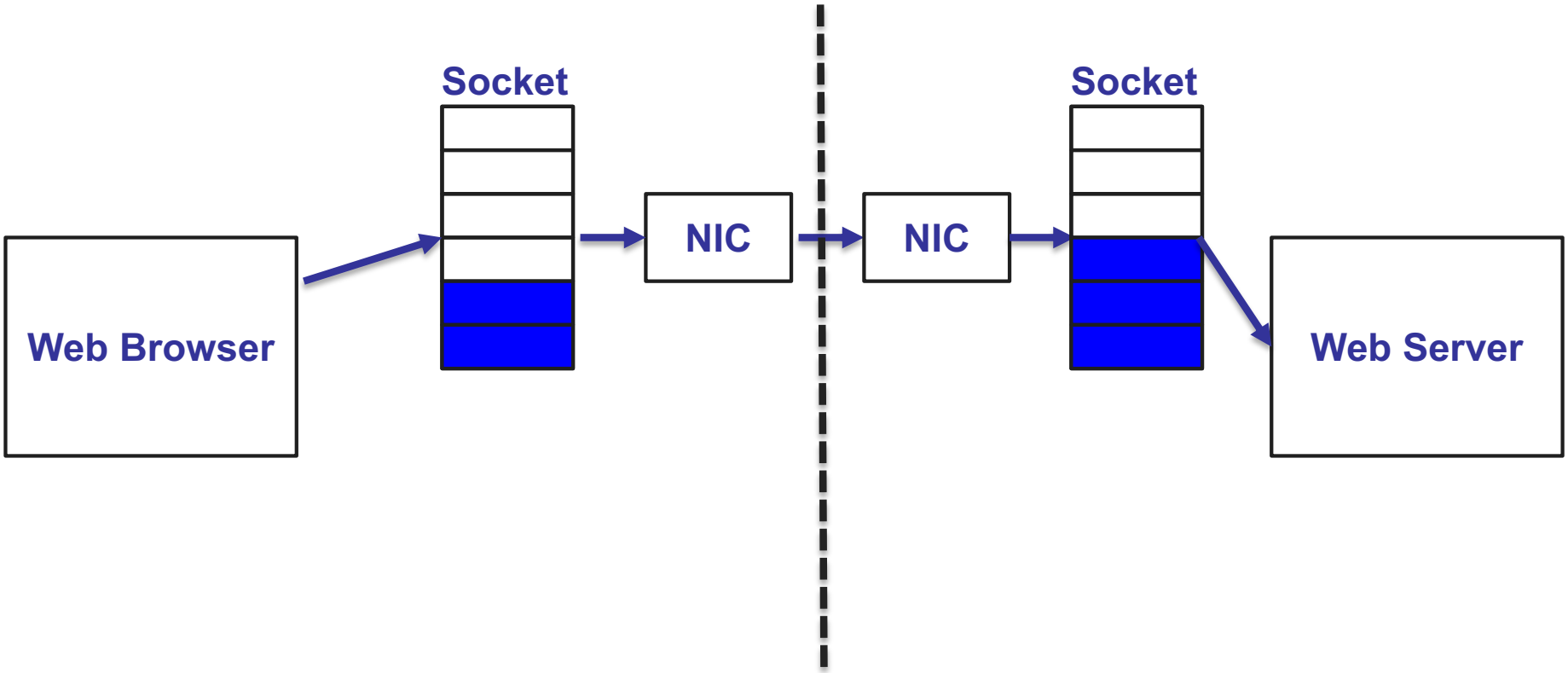
OS virtualizes NIC



- UDP (user datagram protocol): IP + sockets
- TCP (transmission control protocol): IP + sockets + reliable, ordered bytestreams

Socket as a bounded buffer

- How do you send data when network busy?
- How do you receive data when process busy?



TCP

- What guarantees does internet provide for packet delivery?
  - chirp, chirp, chirp... (none)
  - Packets can be:
    - » Dropped/lost
    - » Mutilated/corrupted
    - » Duplicated
    - » Delayed/delivered out of order
- TCP must provide guarantees on top of this

Ordered messages

- Hardware reality: messages can be re-ordered
  - Sender: A, B
  - Receiver: B, A
- Application interface: messages recvd in order sent
- How to detect reordering of messages?
  - Assign sequence numbers
- Ordering of messages per ”connection”
  - TCP: process opens connection (via `connect`), sends sequence of data, then closes connection
  - Sequence number specific to a socket-to-socket connection

Ordered messages

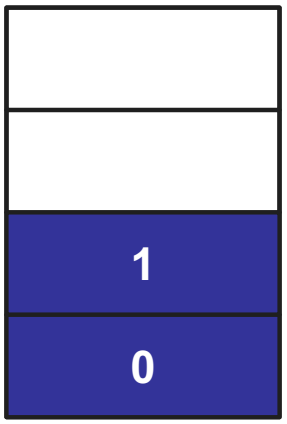
- Example:
  - Sender sends 0, 1, 2, 3, 4, …
  - Receiver receives 0, 1, 3, 2, 4, …

- How should receiver deal with reordering?
  - Drop 3, Deliver 2, Deliver 4
  - Deliver 3, Drop 2, Deliver 4
  - Save 3, Deliver 2, Deliver 3, Deliver 4

Ordered messages

- Example:
  - Sender sends 0, 1, 2, 3, 4, …
  - Receiver receives 0, 1, 3, 2, 4, …

- How should receiver deal with reordering?
  - Drop 3, Deliver 2, Deliver 4
  - Deliver 3, Drop 2, Deliver 4
  - Save 3, Deliver 2, Deliver 3, Deliver 4



Ordered messages

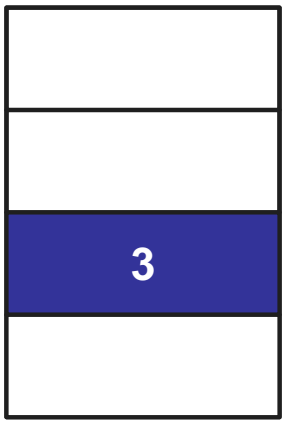
- Example:
  - Sender sends 0, 1, 2, 3, 4, …
  - Receiver receives 0, 1, 3, 2, 4, …

- How should receiver deal with reordering?
  - Drop 3, Deliver 2, Deliver 4
  - Deliver 3, Drop 2, Deliver 4
  - Save 3, Deliver 2, Deliver 3, Deliver 4

Ordered messages

- Example:
  - Sender sends 0, 1, 2, 3, 4, …
  - Receiver receives 0, 1, 3, 2, 4, …

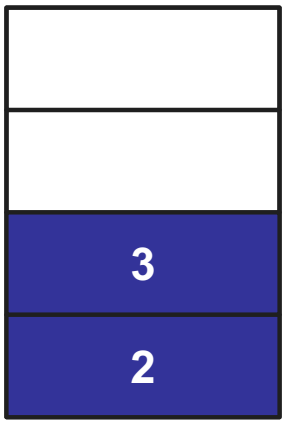
- How should receiver deal with reordering?
  - Drop 3, Deliver 2, Deliver 4
  - Deliver 3, Drop 2, Deliver 4
  - Save 3, Deliver 2, Deliver 3, Deliver 4



Ordered messages

- Example:
  - Sender sends 0, 1, 2, 3, 4, …
  - Receiver receives 0, 1, 3, 2, 4, …

- How should receiver deal with reordering?
  - Drop 3, Deliver 2, Deliver 4
  - Deliver 3, Drop 2, Deliver 4
  - Save 3, Deliver 2, Deliver 3, Deliver 4



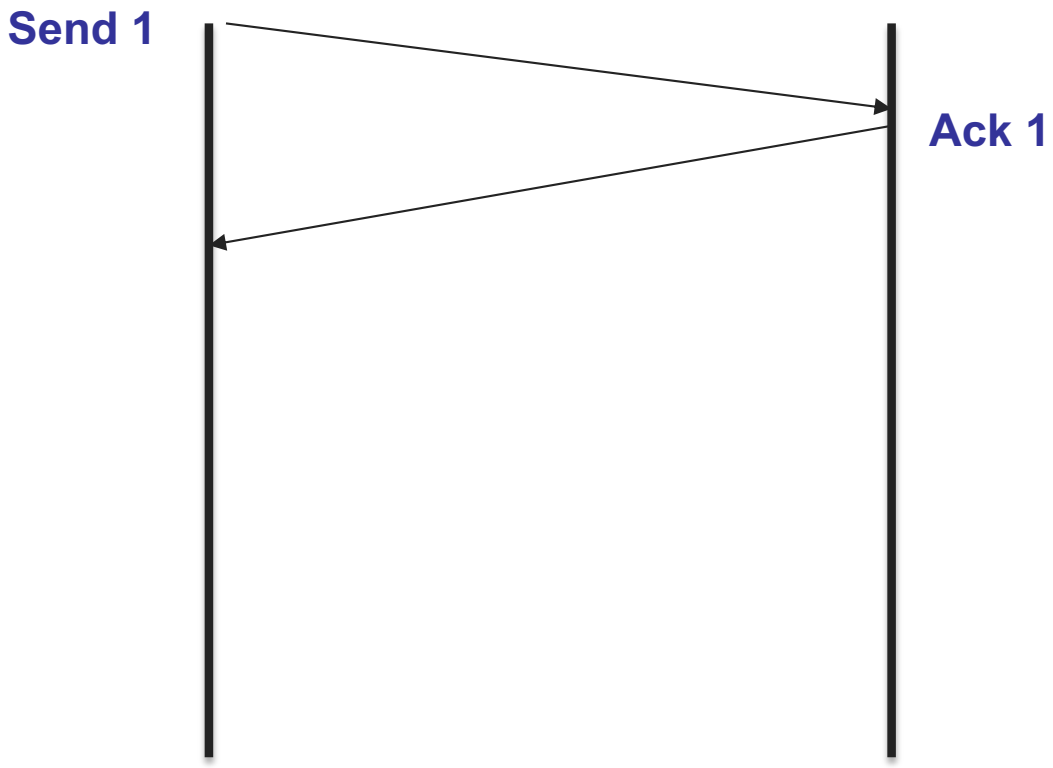
Reliable messages

- Hardware interface: Messages can be dropped, duplicated, or corrupted
- Application interface: Each message is delivered exactly once (without corruption)

- How to fix a dropped message?
  - Have the sender re-send it
- How does sender know message was dropped?
  - Have receiver ACK messages; resend after timeout
  - Downside of dealing with drops this way?

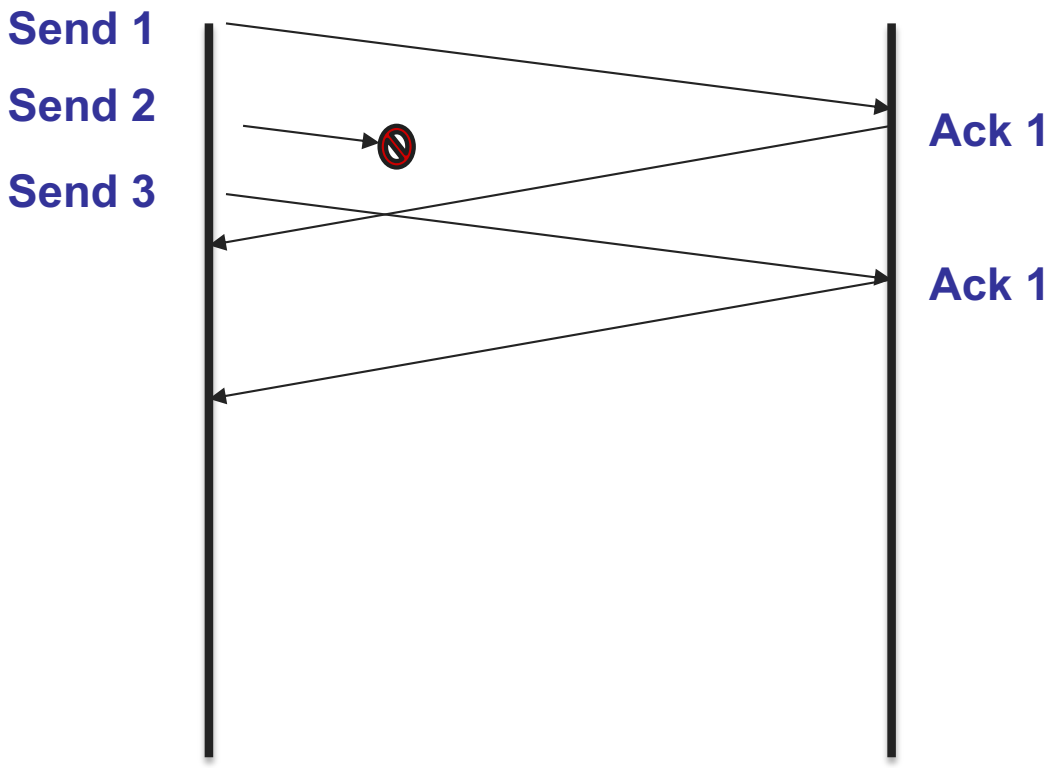
Fast retransmission

- Resend if receive 3 duplicate acks



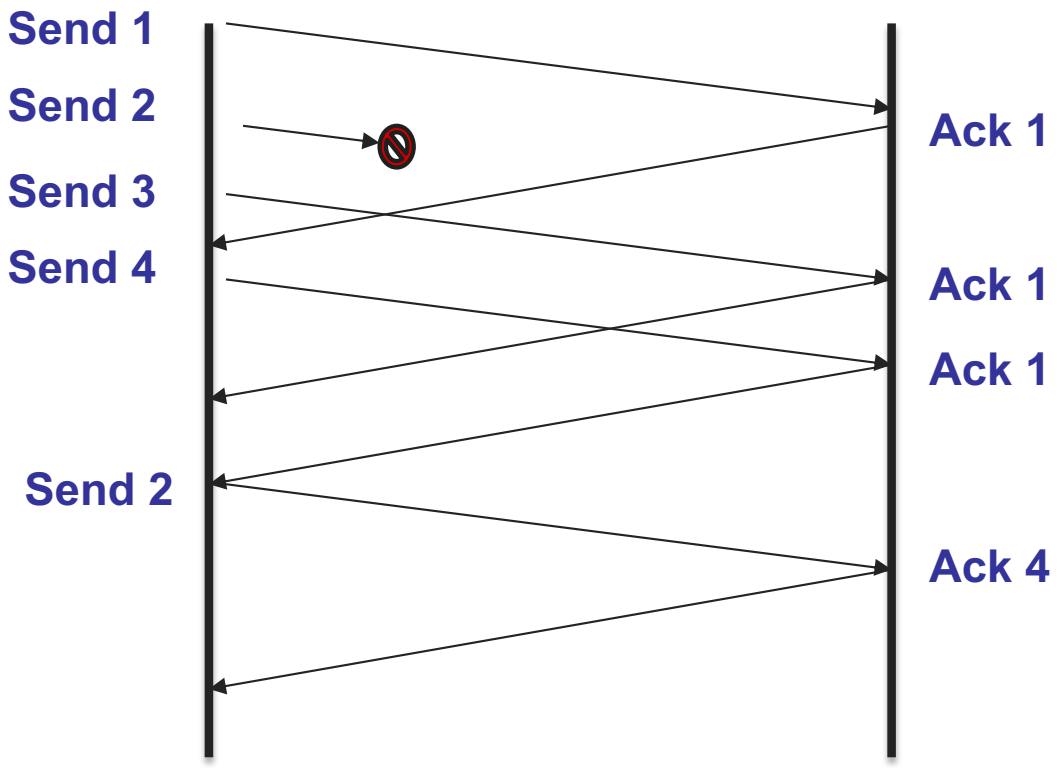
Fast retransmission

- Resend if receive 3 duplicate acks



Fast retransmission

- Resend if receive 3 duplicate acks



Reliable messages

- How to deal with duplicate messages?
  - Detect by sequence #s and drop duplicates
- How to deal with corrupted messages?
  - Add redundant information (e.g., checksum)
  - Fix by dropping corrupted message
- Transformations:
  - Corrupted messages → dropped messages
  - Potential dropped messages → potential duplicates
  - Solve duplicates by sequence #/dropping

Byte streams

- Hardware interface: Send/receive messages
- Application interface: Abstraction of data stream

- TCP: Sender sends messages of arbitrary size, which are combined into a single stream
- Implementation
  - Break up stream into fragments
  - Sends fragments as distinct messages/packets
  - Reassembles fragments at destination

Message boundaries

- TCP has no message boundaries (unlike UDP)
  - Example: Sender sends 100 bytes, then 50 bytes; Receiver could receive 1-150 bytes
- Receiver must loop until all bytes received
- How to know # of bytes to receive?
  - Convention (e.g., specified by protocol)
  - Specified in header
  - End-of-message delimiter
  - Sender closes connection

Project 4

- Start ASAP: Due in less than 3 weeks
- Project 3 scores on day project due
  - If first submit before 3/14: mean 77, median 85
  - If first submit on or after 3/14: mean 66, median 75
- Read man pages
- Things to keep in mind:
  - Encrypted data is not a C-string
  - File data can contain NULL character
  - Data received over network is untrusted