

Based on slides by Harsha V. Madhyastha

EECS 482 Introduction to Operating Systems

Spring/Summer 2020

Lecture 21: Sockets and tcp

Nicole Hamilton

[https://web.eecs.umich.edu/~nham/
nham@umich.edu](https://web.eecs.umich.edu/~nham/nham@umich.edu)

Agenda

1. RAID.
2. Sockets.
3. Project 4.

Agenda

1. RAID.
2. Sockets.
3. Project 4.

RAID

Redundant Array of Inexpensive Disks (RAID)

Invented by David Patterson at Berkeley.

Sits in between hardware and the file system.

Idea: Use many disks in parallel to increase storage bandwidth, improve reliability.

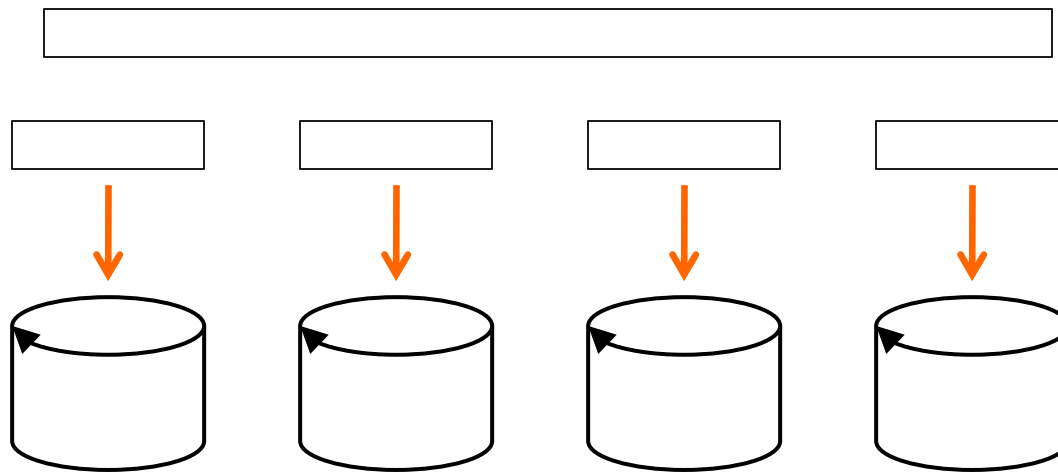
Implemented in the hardware.

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

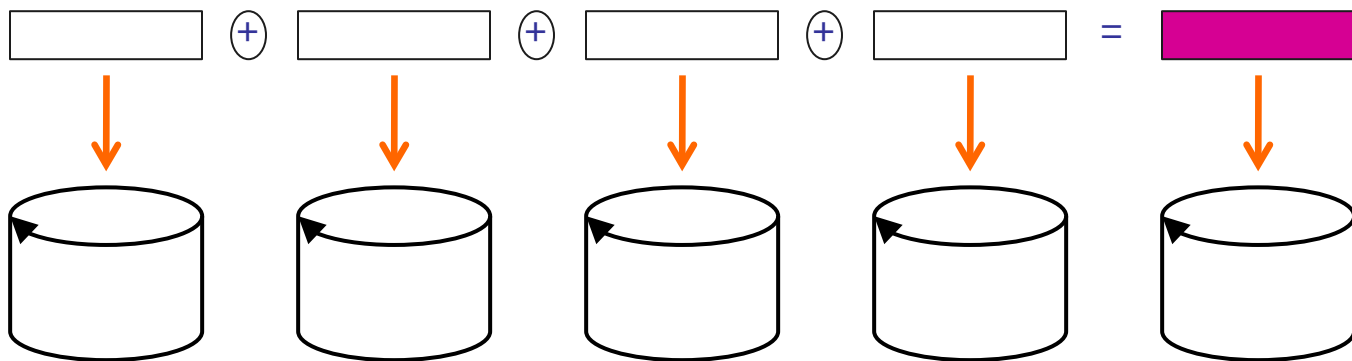
Improve reliability by storing redundant parity.

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

Improved bandwidth but decreased reliability.

RAID 1: Mirroring

Maintain full copy of all data.

Recover if a drive fails.

RAID 10: Mirroring and striping

Supported by many PC motherboards.

Improved bandwidth and recoverability if a drive fails.

RAID 5: Floating parity

Parity blocks for different stripes written to different disks.

No single parity disk → no bottleneck at that disk.

Status

Create

Manage

Intel® Optane™ Memory

Performance

Preferences

Help



Current Status

Your system is functioning normally.

Create

Create a volume by combining available disks to enhance your storage system.

[Create a custom volume](#)

Manage

Click on any element in the storage system view to manage its properties.

The Windows* write-cache buffer flushing policy can be enabled for all RAID array drives to ensure data integrity or disabled to improve data performance. Click the Help icon for more information on setting the write-cache buffer flushing policy based on your needs.

Intel® Optane™ Memory

Intel® Optane™ memory status: disabled. [Enable](#)

Storage System View

SATA_Array_0000

932 GB

932 GB

932 GB

932 GB



Desktop
Type: RAID 10
1,863 GB

 SATA SSD (1,863 GB)
1,863 GB SATA disk (3,726 GB)
3,726 GB

[More help on this page](#)

Status

Create

Manage

Intel® Optane™ Memory

Performance


Preferences

Help



Access help and support

Manage Disk

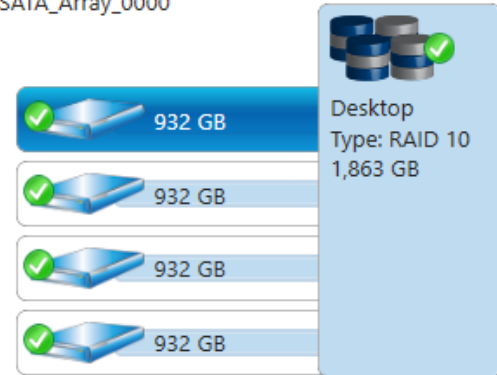
Controller 0, Port 0
Port location: Internal
Status: Normal
Type: SATA hard disk
Usage: Array disk
Size: 953,870 MB 
Serial number: Z1DD6SE6
Model: ST1000DX001-1CM162
Firmware: CC43

▼ Advanced


Password protected: No
Disk data cache: Enabled
Native command queuing: Yes
SATA transfer rate: 6 Gb/s
Physical sector size: 4096 Bytes
Logical sector size: 512 Bytes

Storage System View

SATA_Array_0000



Desktop
Type: RAID 10
1,863 GB

 SATA SSD (1,863 GB)
1,863 GB

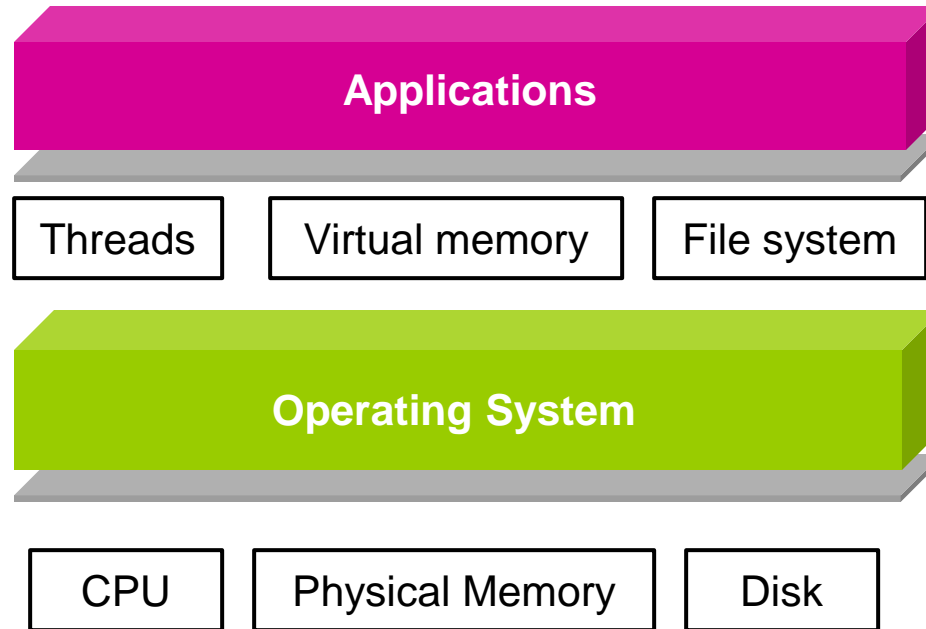
 SATA disk (3,726 GB)
3,726 GB

[More help on this page](#)

Agenda

1. RAID.
2. Sockets.
3. Project 4.

OS Abstractions



Next few lectures:

Abstraction of network, a preview of EECS 489

Distributed systems, a preview of EECS 491

OS abstraction of network

Hardware reality

One network interface controller (NIC) enabling machine-to-machine communication shared by all processes.

Network is unreliable, may lose or garble packets, deliver duplicates or out-of-order.

Unordered delivery of finite-sized messages.

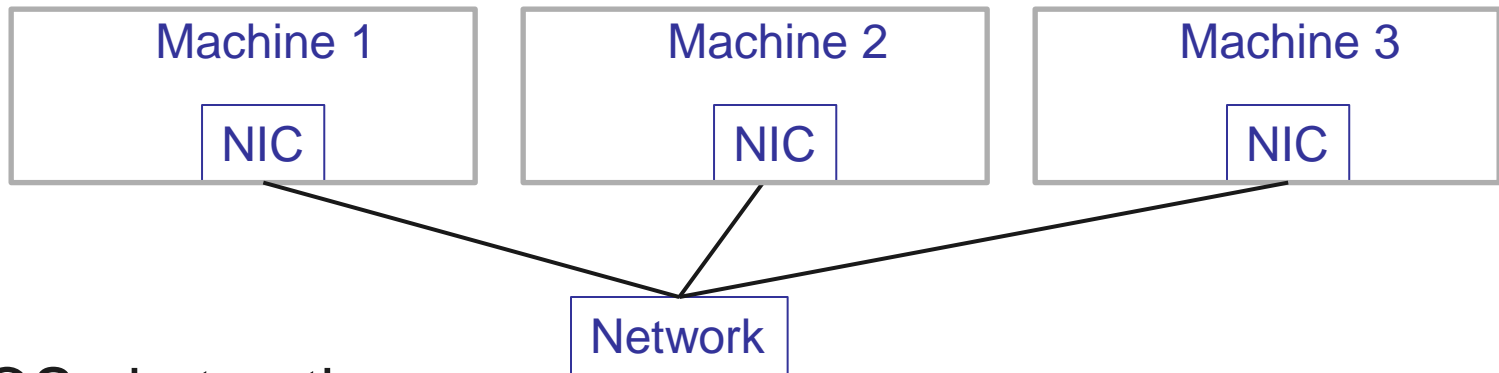
OS abstraction

Process-to-process communication.

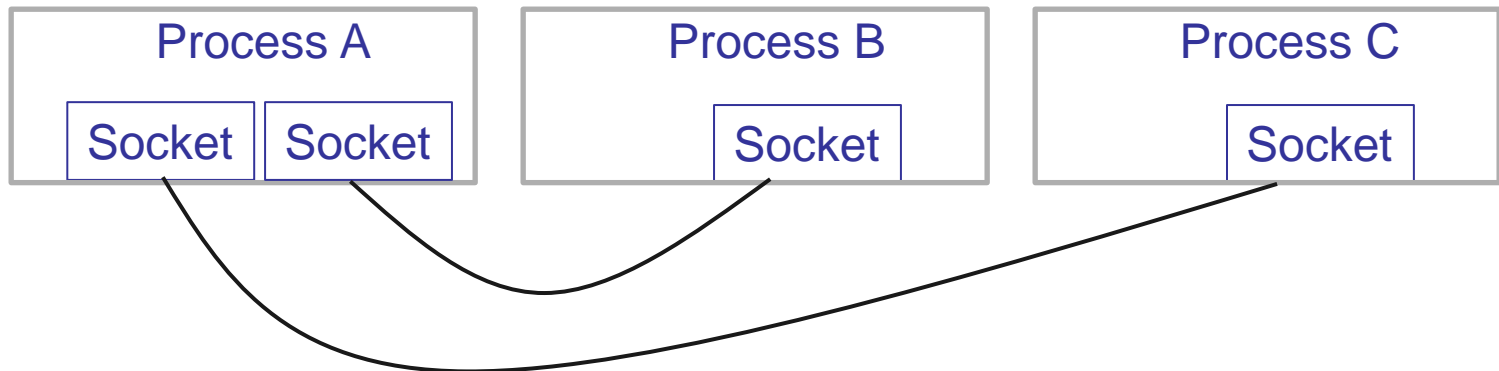
Reliable and ordered delivery of a byte stream.

OS abstraction of network

Hardware reality



OS abstraction



Inter-machine to inter-process

Every process thinks it has its own:

Processor (threads)

Memory (address space)

Network interface (sockets)

Socket

Virtual network interface controller.

Endpoint for communication.

NIC named by MAC/IP address; socket named by “port number”
(via bind)

Programming interface: BSD sockets (Friday’s lab)

Analogy

Computer is like an apartment building.

NIC has a well-known address (MAC/IP address).

Contains many mailboxes.

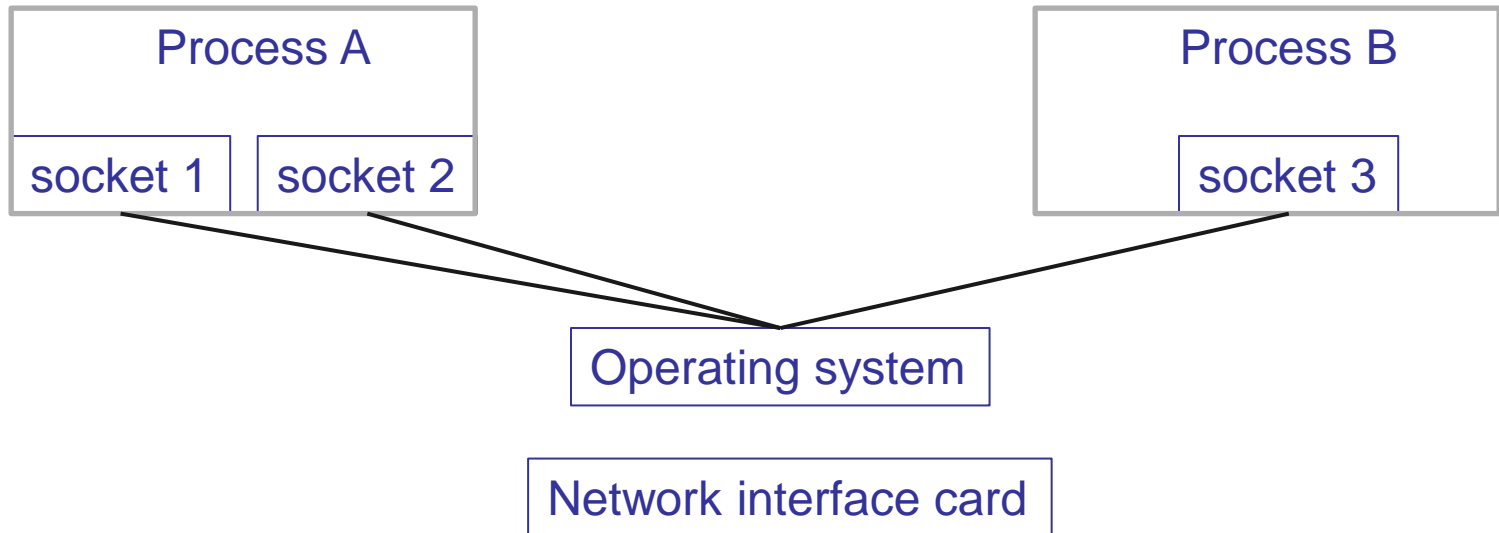
Each labeled with apt. no. (port number).

You put outgoing mail in box, mailman picks up.

Mailman puts mail in box, you pickup later.

Mailbox has a fixed size (can't overstuff).

OS virtualizes NIC



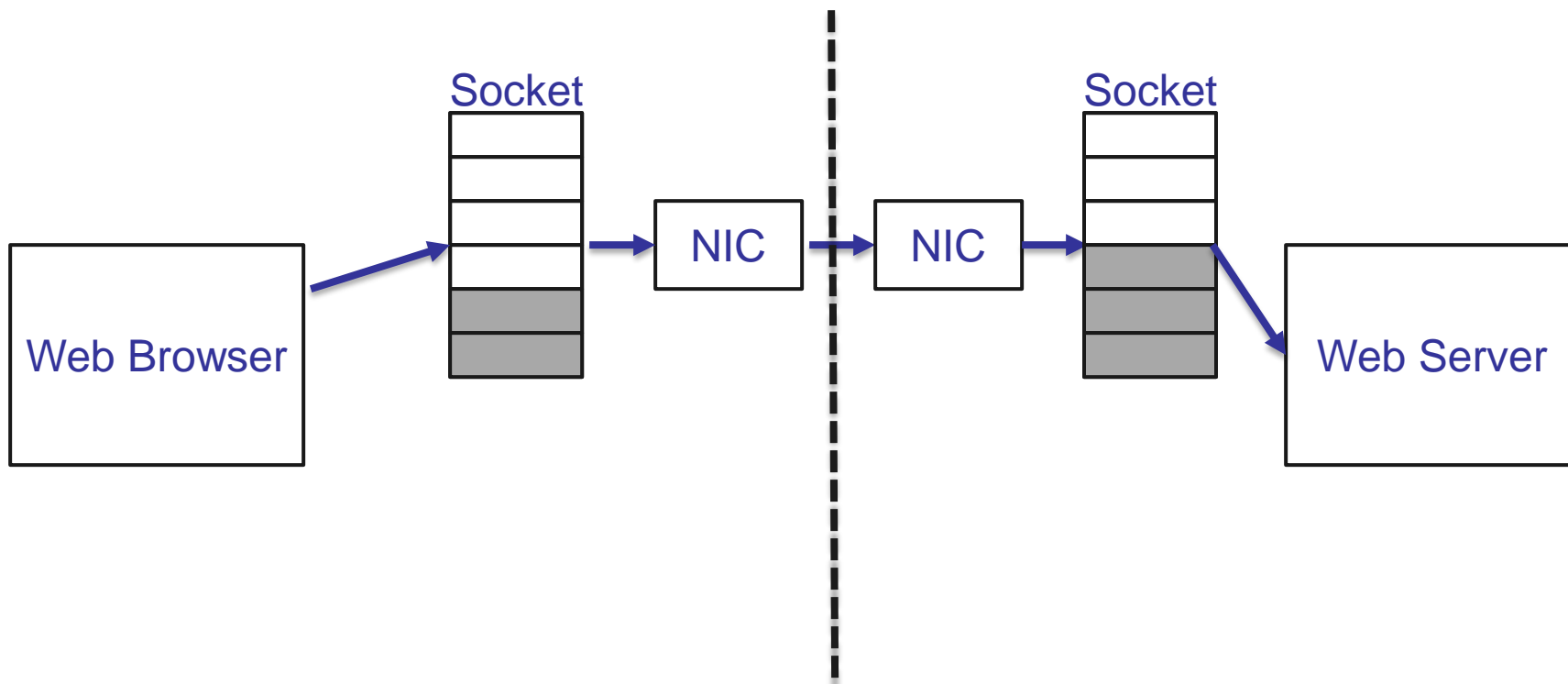
UDP (user datagram protocol): Unreliable IP + sockets.

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, ...

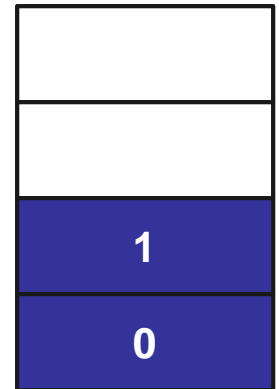
Receiver receives 0, 1, 3, 2, 4, ...

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, ...

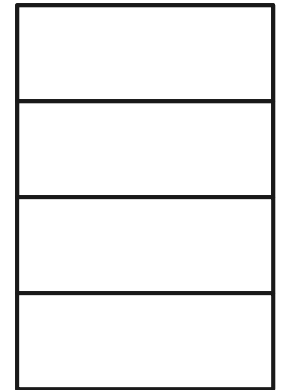
Receiver receives 0, 1, 3, 2, 4, ...

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, ...

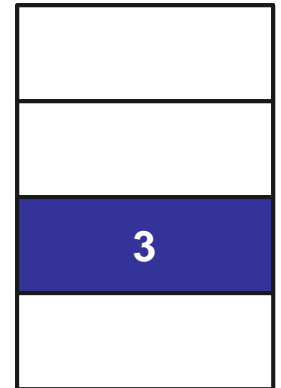
Receiver receives 0, 1, 3, 2, 4, ...

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, ...

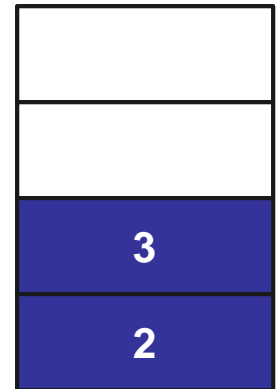
Receiver receives 0, 1, 3, 2, 4, ...

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?

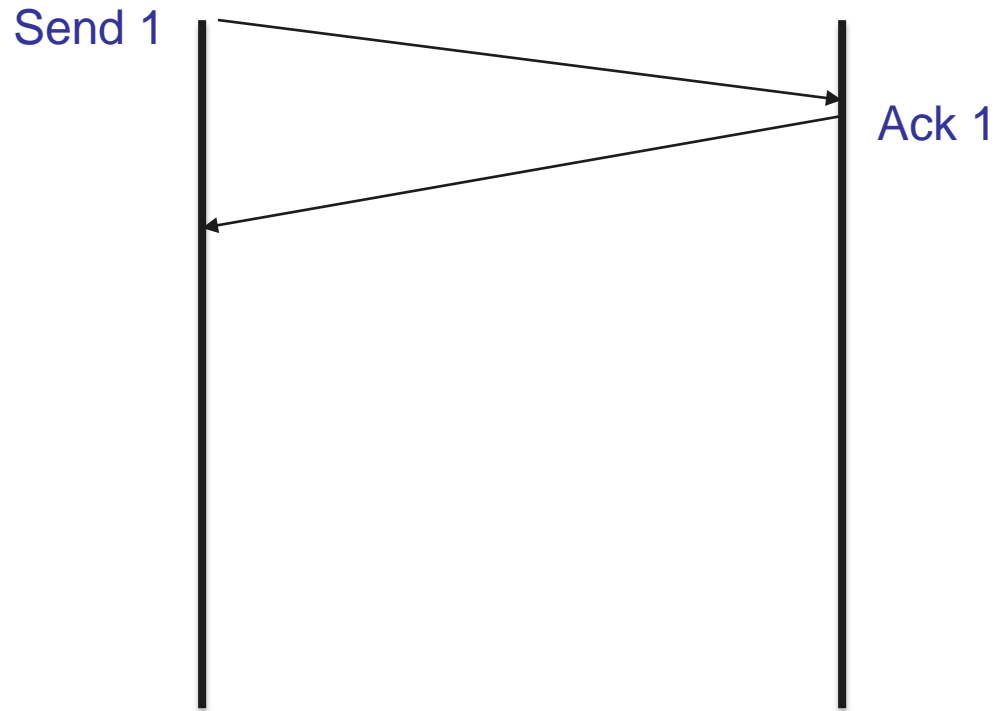
Have receiver ACK messages; resend after timeout. Learns the timeout by observing responses to previous messages.

Downside of dealing with drops this way?

Could send duplicate messages, wait too long before resending.

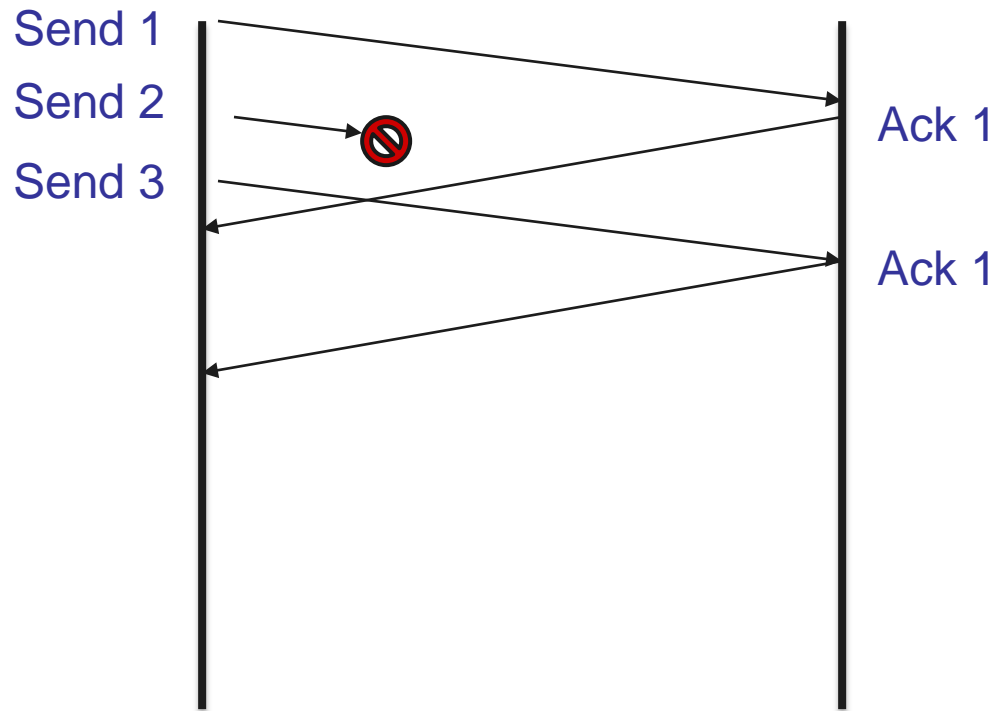
Fast retransmission

Resend if receive 3 duplicate acks



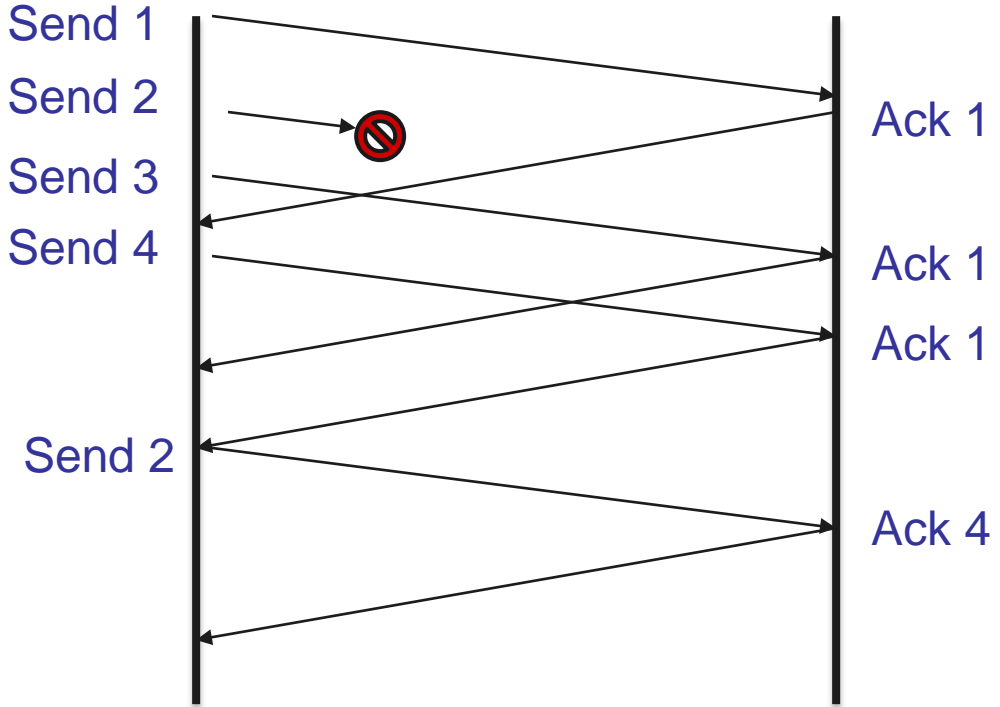
Fast retransmission

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 numbers and drop duplicates.

How to deal with corrupted messages?

Add redundant information (e.g., checksum).

Fix by dropping corrupted message.

Basic strategy

Treat corrupted messages as dropped messages.

Potential for dropped and duplicated messages.

Solve duplicates by sequence numbering and 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.

- Send fragments as distinct messages/packets.

- Reassemble 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.

```
rc = recv( s, buf, size, 0 );
```

Blocks until error or can return >0 bytes.

May return up to size bytes.

Returns 0 if connection closed, -1 on error.

Message boundaries

Call to `recv` blocks until at least 1 byte received.

How to know number of bytes to receive?

- Convention (e.g., specified by protocol).

- Specified in header.

- End-of-message delimiter.

- Sender closes connection.

OS abstraction of network

Hardware reality	Abstraction
Multiple computers connected via a network	Single computer
Machine-to-machine communication	Process-to-process communication
Unreliable and unordered delivery of finite messages	Reliable and ordered delivery of byte stream

Client-server

Common way to structure a distributed application:

Server provides some centralized service.

Client makes request to server, waits for response.

Example: **Web server**

Server stores and returns web pages.

Clients run Web browsers, make GET/POST requests.

Example: **Producer/consumer relationship with a server**

Server manages state associated with coke machine.

Clients call `client_produce()` or `client_consume()`, which send request to the server and return when done.

Client requests block at the server until they are satisfied.

Client-server example

```
client_produce( )  
  {  
  send produce request to server;  
  receive response;  
  }
```

```
server( )  
  {  
  while ( true )  
  {  
  receive request;  
  if ( produce request )  
    add coke to machine;  
  else  
    take coke out of machine;  
  send response;  
  }  
  }
```

Problems?

Only single client at a time.

What if machine is full or empty?

Machine can deadlock.

Client-server example

```
client_produce( )
{
  send produce request to server;
  receive response;
}
```

Problems?

Still possible to deadlock.

```
server( )
{
  while ( true )
  {
    receive request;
    if ( produce request )
    {
      while ( machine is full )
        wait;
      add coke to machine;
      signal;
    }
    else
      ...
    send response;
  }
}
```

Example with threads

```
server( )
{
  while ( true )
  {
    receive request;
    if ( produce request )
      create thread that calls
        server_produce( );
    else
      create thread that calls
        server_consume( );
  }
}
```

```
server_produce( )
{
  lock;
  while ( machine is full )
    wait;
  put coke in machine;
  signal;
  send response;
  unlock;
}
```

Two modifications necessary to reduce blocking on slow operations?

Example with threads

```
server( )
{
  while ( true )
  {
    wait for new connection
      from client;
    create thread that calls
      handle_request( );
  }
}
```

```
handle_request( )
{
  receive request;
  produce request ?
  server_produce( ) :
  server_consume( );
}
```

```
server_produce( )
{
  lock;
  while ( machine is full )
    wait;
  put coke in machine;
  signal;
  unlock;
  send response;
}
```

Optimizations

How to lower overhead of creating threads?

Maintain pool of worker threads.

There are other ways to structure the server

Basic goal: Account for “slow” operations.

Examples:

Polling (via `select`)

Signals

Next time ...

Remote Procedure Call

Agenda

1. RAID.
2. Sockets.
3. **Project 4.**

Project 4: due August 17

Start ASAP: Due in only 19 days.

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.

Friday's lab: Sockets!

Project 4

Use **assertions** to catch errors early

No. of free disk blocks matches file system contents?

Are you unlocking a lock that you hold?

Verify initial file system is not malformed

Use **showfs** to verify that contents of file system match your expectations

Test cases: cover all states of data structures

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

```
rc = recv( s, buf, size, 0 );
```

Blocks until error or can return >0 bytes

May return up to size bytes

Returns 0 if connection closed, -1 on error

Looping to get n bytes

Here's how to receive
n bytes.

```
char buf[ n ];
int bytes_rcvd = 0;
while ( bytes_rcvd < n )
{
    rc = recv ( s, buf + bytes_rcvd,
               n - bytes_rcvd, 0 );
    if ( rc <= 0 )
    {
        perror( "read" );
        break;
    }
    bytes_rcvd += rc;
}
```

Determining boundaries

How to know # of bytes to receive?

- Convention (e.g., specified by protocol)

- Specified in fixed-size header (length field)

- Sentinel: end-of-message delimiter

 - Careful about reading too many bytes!

- Sender closes connection

 - Loop until recv returns 0