EE 565: Computer Communication Networks I

Lecture 7

Physical Layer 4 + Project 2

Winter Quarter 2022

Today’s Lecture: Physical Layer

- Physical Layer
  - Digital networking
  - Modulation
  - Characterization of Communication Channels
  - Fundamental Limits in Digital Transmission
  - Modems and Digital Modulation
  - Line Coding
  - Properties of Media and Digital Transmission Systems
  - Error Detection and Correction

Today’s Lecture: Physical Layer

- Physical Layer
  - Digital networking
  - Modulation
  - Characterization of Communication Channels
  - Fundamental Limits in Digital Transmission
  - Modems and Digital Modulation
  - Line Coding
  - Properties of Media and Digital Transmission Systems
  - Error Detection and Correction

Error Control

- Channels introduce errors in digital communications
- Applications require certain reliability level
  - Data applications require error-free transfer
  - Voice & video applications tolerate some errors
- Error control may be needed to meet application requirement
- Error control ensures a data stream is transmitted to a certain level of accuracy despite errors
- Two basic approaches:
  - Error ***detection*** & retransmission (ARQ)
  - Forward error ***correction*** (FEC)

Single Parity Code

- Information (7 bits): (0, 1, 0, 1, 1, 0, 0)
- Mini Quiz: Parity Bit?
  (“True” → 1, ”False” → 0)

  $b_8 = 0 + 1 + 0 + 1 + 1 + 0 = 1$

  Codeword (8 bits): (0, 1, 0, 1, 1, 0, 0, 1)

- If single error in bit 3? (0, 1, 1, 1, 1, 0, 0, 1)
  - # of 1’s =5, odd => Error detected

- If errors in bits 3 and 5? (0, 1, 1, 1, 0, 0, 0, 1)
  - # of 1’s =4, even => Error not detected

Two-Dimensional Parity Check

- More parity bits to improve coverage
- Arrange information as columns
- Add single parity bit to each column
- Add a final “parity” column
- Used in early error control systems

$$
\begin{array}{ccccc|c}
1 & 0 & 0 & 1 & 0 & 0 \\
0 & 1 & 0 & 0 & 0 & 1 \\
1 & 0 & 0 & 1 & 0 & 0 \\
1 & 1 & 0 & 1 & 1 & 0 \\
\hline
1 & 0 & 0 & 1 & 1 & 1
\end{array}
$$

Last column consists of check bits for each row

Bottom row consists of check bit for each column

Error-detecting capability

$$\begin{array}{ccccc|c}
1 & 0 & 0 & 1 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 1 \\
1 & 0 & 0 & 1 & 0 & 0 \\
1 & 1 & 0 & 1 & 1 & 0 \\
\hline
1 & 0 & 0 & 1 & 1 & 1
\end{array}$$

One error

$$\begin{array}{ccccc|c}
1 & 0 & 0 & 1 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 1 \\
1 & 0 & 0 & 1 & 0 & 0 \\
1 & 0 & 0 & 1 & 1 & 0 \\
\hline
1 & 0 & 0 & 1 & 1 & 1
\end{array}$$

Two errors

$$\begin{array}{ccccc|c}
1 & 0 & 0 & 1 & 0 & 0 \\
0 & 0 & 0 & 1 & 0 & 1 \\
1 & 0 & 0 & 1 & 0 & 0 \\
1 & 0 & 0 & 1 & 1 & 0 \\
\hline
1 & 0 & 0 & 1 & 1 & 1
\end{array}$$

Three errors

$$\begin{array}{ccccc|c}
1 & 0 & 0 & 1 & 0 & 0 \\
0 & 0 & 0 & 1 & 0 & 1 \\
1 & 0 & 0 & 1 & 0 & 0 \\
1 & 0 & 0 & 0 & 1 & 0 \\
\hline
1 & 0 & 0 & 1 & 1 & 1
\end{array}$$

Four errors

1, 2, or 3 errors can always be detected; Not all patterns >4 errors can be detected

Arrows indicate failed check bits

Checksum Calculation

The checksum $\mathbf{b}_L$ is calculated as follows:

- Treating each 16-bit word as an integer, find

$$\mathbf{x} = \mathbf{b}_0 + \mathbf{b}_1 + \mathbf{b}_2 + \ldots + \mathbf{b}_{L-1} \text{ modulo } 2^{16}-1$$

- The checksum is then given by:

$$\mathbf{b}_L = -\,\mathbf{x} \quad \text{modulo } 2^{16}-1$$

Thus, the headers must satisfy the following ***pattern*** at the receiver:

$$\mathbf{0} = \mathbf{b}_0 + \mathbf{b}_1 + \mathbf{b}_2 + \ldots + \mathbf{b}_{L-1} + \mathbf{b}_L \text{ modulo } 2^{16}-1$$

- The checksum calculation is carried out in software using one’s complement arithmetic

Internet Checksum Example

Use Modulo Arithmetic

- Assume 4-bit words
- Use mod 2^4-1 (= 15) arithmetic
- $\underline{b}_0$=1100 = 12
- $\underline{b}_1$=1010 = 10
- $\underline{b}_0$+$\underline{b}_1$=12+10=7 mod15
- $\underline{b}_2$ = -7 = 8 mod15
- Therefore
- $\underline{b}_2$=1000

Use Binary Arithmetic

- Note 16 =1 mod15
- So: 10000 = 0001 mod15
- leading bit wraps around

$$
\begin{aligned}
b_0 + b_1 &= 1100+1010 \\
&=10110 \\
&=10000+0110 \\
&=0001+0110 \\
&=0111 \\
&=7
\end{aligned}
$$

Take 1's complement

b_2 = -0111 =1000

Polynomial Codes

- Polynomials instead of vectors for codewords
- Polynomial arithmetic instead of check sums
- Implemented using shift-register circuits
- Also called *cyclic redundancy check (CRC)*
- Most data communications standards use polynomial codes for error detection
    - Have very simple hardware implementations
- Polynomial codes also basis for powerful error-correction methods

Binary Polynomial Arithmetic

- Binary vectors map to polynomials

$$(i_{k-1}, i_{k-2}, \ldots, i_2, i_1, i_0) \to i_{k-1}x^{k-1} + i_{k-2}x^{k-2} + \ldots + i_2x^2 + i_1x^1 + i_0$$

Addition:

$$\begin{aligned}(x^7 + x^6 + 1) + (x^6 + x^5) &= x^7 + x^6 + x^6 + x^5 + 1 \\ &= x^7 + (1+1)x^6 + x^5 + 1 \\ &= x^7 + x^5 + 1 \quad \text{since} \quad 1+1 = 0 \bmod 2\end{aligned}$$

Multiplication:

$$\begin{aligned}(x+1)(x^2 + x + 1) &= x(x^2 + x + 1) + 1(x^2 + x + 1) \\ &= (x^3 + x^2 + x) + (x^2 + x + 1) \\ &= x^3 + 1\end{aligned}$$

Binary Polynomial Division

- Division with Decimal Numbers

$$
\begin{array}{rl}
 & \ 34 \quad \text{quotient} \\
\text{divisor} \quad 35 &)\ 1222 \quad \text{dividend} \\
 & \ \underline{105} \\
 & \ \ \ 172 \\
 & \ \ \ \underline{140} \\
 & \ \ \ \ \ 32 \quad \text{remainder}
\end{array}
$$

dividend = quotient x divisor + remainder

$1222 = 34 \times 35 + 32$

- Polynomial Division

$$
\begin{array}{rl}
 & \ x^3 + x^2 + x \qquad = q(x) \ \text{quotient} \\
\text{divisor} \quad x^3 + x + 1 &)\ x^6 + x^5 \qquad \text{dividend} \\
 & \ \underline{x^6 + \ x^4 + x^3} \\
 & \ \ x^5 + x^4 + x^3 \\
 & \ \ \underline{x^5 + \ x^3 + x^2} \\
 & \ \ x^4 + \ x^2 \\
 & \ \ \underline{x^4 + \ x^2 + x} \\
 & \ \ x \qquad = r(x) \ \text{remainder}
\end{array}
$$

Note: Degree of r(x) is less than degree of divisor

Polynomial Coding

- k information bits define polynomial of degree k-1

$$i(x) = i_{k-1}x^{k-1} + i_{k-2}x^{k-2} + \ldots + i_2x^2 + i_1x + i_0$$

$$g(x) = x^{n-k} + g_{n-k-1}x^{n-k-1} + \ldots + g_2x^2 + g_1x + 1$$

- Find *remainder polynomial* of at most degree n-k-1

$$\begin{array}{r} q(x) \\ g(x)\,\overline{)\,x^{n-k}\, i(x)} \\ r(x) \end{array}$$

$$x^{n-k}i(x) = q(x)g(x) + r(x)$$

- Define the *codeword polynomial* of degree n-1

$$\boxed{\underbrace{b(x)}_{\text{n bits}} = \underbrace{x^{n-k}i(x)}_{\text{k bits}} + \underbrace{r(x)}_{\text{n-k bits}}}$$

Quiz Q: Find codeword if k=4, n-k=3

And: Generator polynomial: $g(x) = x^3 + x + 1$

Information: (1,1,0,0) $i(x) = x^3 + x^2$

Encoding: $x^3 i(x) = x^6 + x^5$

Quiz Q: Find codeword if k=4, n-k=3

And: Generator polynomial: $g(x) = x^3 + x + 1$

Information: (1,1,0,0) $\quad i(x) = x^3 + x^2$

Encoding: $x^3 i(x) = x^6 + x^5$

$$
\begin{array}{r l}
 & x^3 + x^2 + x \\
\cline{2-2}
x^3 + x + 1\ ) & x^6 + x^5 \\
 & x^6 + \ x^4 + x^3 \\
\cline{2-2}
 & \ x^5 + x^4 + x^3 \\
 & \ x^5 + \ x^3 + x^2 \\
\cline{2-2}
 & \ x^4 + \ x^2 \\
 & \ x^4 + \ x^2 + x \\
\cline{2-2}
 & \ x
\end{array}
$$

Quiz Q: Find codeword if k=4, n-k=3

And: Generator polynomial: $g(x) = x^3 + x + 1$

Information: (1,1,0,0) $\quad i(x) = x^3 + x^2$

Encoding: $x^3 i(x) = x^6 + x^5$

$$
\begin{array}{r}
x^3 + x^2 + x \\
x^3 + x + 1\ )\ \overline{x^6 + x^5 } \\
x^6 + \ x^4 + x^3 \\
\hline
x^5 + x^4 + x^3 \\
x^5 + \ x^3 + x^2 \\
\hline
x^4 + \ x^2 \\
x^4 + \ x^2 + x \\
\hline
x
\end{array}
$$

$$
\begin{array}{r}
1110 \\
1011\ )\ \overline{1100000} \\
1011 \\
\hline
1110 \\
1011 \\
\hline
1010 \\
1011 \\
\hline
010
\end{array}
$$

Transmitted codeword:

$b(x) = x^6 + x^5 + x$

$\underline{b} = (1,1,0,0,0,1,0)$

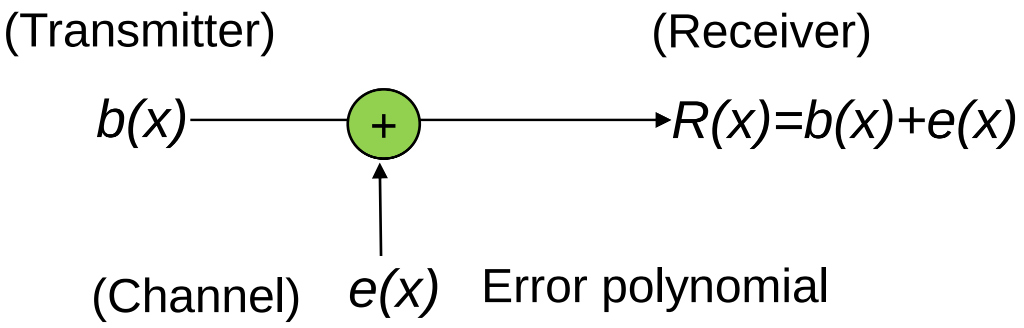
The *Pattern* in Polynomial Coding

- All codewords satisfy the following **pattern**:

$$b(x) = x^{n-k}i(x) + r(x) = q(x)g(x) + r(x) + r(x) = q(x)g(x)$$

- All codewords are a multiple of *g(x)!*
- Receiver should divide received n-tuple by g(x) and check if remainder is zero
- If remainder is non-zero, then received n-tuple is not a codeword

Undetectable error patterns



- $e(x)$ has 1's in error locations & 0's elsewhere
- Receiver divides the received polynomial $R(x)$ by $g(x)$
- Undetectable error: If $e(x)$ is a multiple of $g(x)$, that is, $e(x)$ is a non-zero codeword, then
- $R(x) = b(x) + e(x) = q(x)g(x) + q'(x)g(x)$
- *The set of undetectable error polynomials is the set of nonzero code polynomials*
- *Choose the generator polynomial so that selected error patterns can be detected.*

Designing good polynomial codes

- Select generator polynomial so that likely error patterns are not multiples of $g(x)$
- *Detecting Single Errors*
  - $e(x) = x^i$ for error in location $i+1$
  - If $g(x)$ has more than 1 term, it cannot divide x^i
- *Detecting Double Errors*
  - $e(x) = x^i + x^j = x^i(x^{j-i}+1)$ *where* $j>i$
  - If $g(x)$ has more than 1 term, it cannot divide x^i
  - If g(x) is a *primitive* polynomial, it cannot divide x^m+1 for all $m<2^{n-k}-1$ (Need to keep codeword length less than $2^{n-k}-1$)
  - Primitive polynomials can be found by consulting coding theory books

Standard Generator Polynomials

CRC = cyclic redundancy check

- CRC-8: $= x^8 + x^2 + x + 1$ ATM
- CRC-16: $= x^{16} + x^{15} + x^2 + 1$
  $= (x+1)(x^{15} + x + 1)$ Bisync
- CCITT-16: $= x^{16} + x^{12} + x^5 + 1$ HDLC, XMODEM, V.41
- CCITT-32: IEEE 802, DoD, V.42

$$= x^{32} + x^{26} + x^{23} + x^{22} + x^{16} + x^{12} + x^{11} + x^{10} + x^8 + x^7 + x^5 + x^4 + x^2 + x + 1$$

Hamming Codes

- Class of *error-correcting* codes
- Capable of correcting all *single-error* patterns
- Provably optimal for 1-bit errors
- Very less redundancy, e.g. 1-bit error proof – adds O(log n) bits of redundancy for n bit sequences

m=3 Hamming Code

- Information bits are b_1, b_2, b_3, b_4
- Equations for parity checks b_5, b_6, b_7

$$
\begin{aligned}
b_5 &= b_1 \phantom{{}+b_2} + b_3 + b_4 \\
b_6 &= b_1 + b_2 \phantom{{}+b_3} + b_4 \\
b_7 &= + b_2 + b_3 + b_4
\end{aligned}
$$

- There are 2^4=16 codewords
- (0,0,0,0,0,0,0) is a codeword

My "simple" proof of optimality

Assume you got the following 7 bit sequences and make the following checks:

$$b_5 = b_1 + b_3 + b_4$$
$$b_6 = b_1 + b_2 + b_4$$
$$b_7 = + b_2 + b_3 + b_4$$

Case	b_5 match	b_6 match	b_7 match
No error			
b_1 flipped			
b_2 flipped			
b_3 flipped			
b_4 flipped			
b_5 flipped			
b_6 flipped			
b_7 flipped			

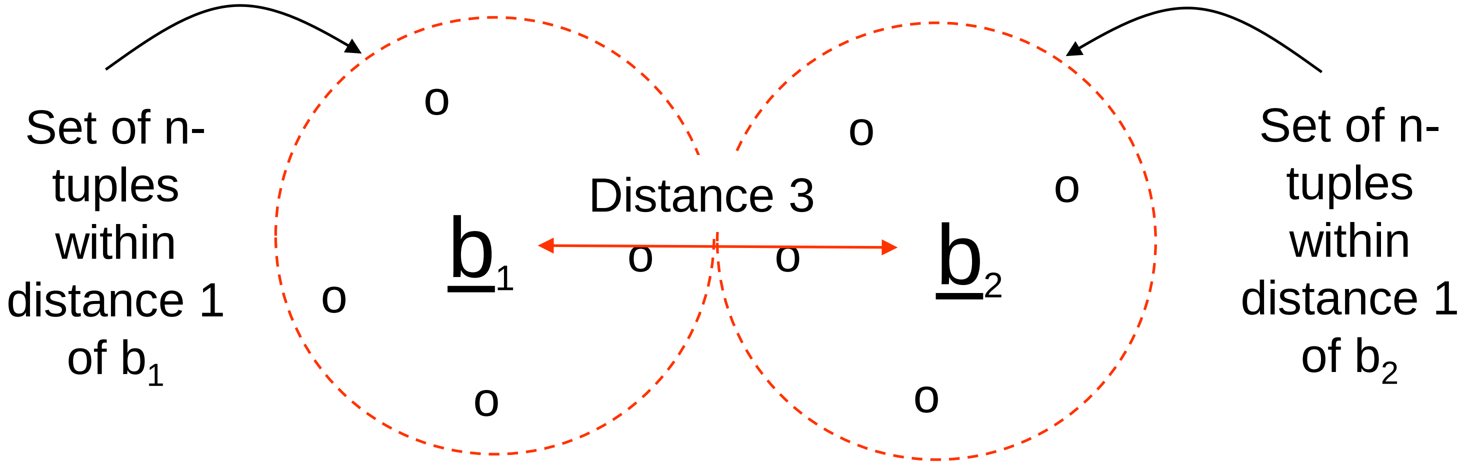
My “simple” proof of optimality

Assume you got the following 7 bit sequences and make the following checks:

$$b_5 = b_1 \qquad + b_3 + b_4$$
$$b_6 = b_1 + b_2 \qquad + b_4$$
$$b_7 = \qquad + b_2 + b_3 + b_4$$

Case	b_5 match	b_6 match	b_7 match
No error	✔	✔	✔
b_1 flipped	X	X	✔
b_2 flipped	✔	X	X
b_3 flipped	X	✔	X
b_4 flipped	X	X	X
b_5 flipped	X	✔	✔
b_6 flipped	✔	X	✔
b_7 flipped	✔	✔	X

Why is Hamming a "good code"?

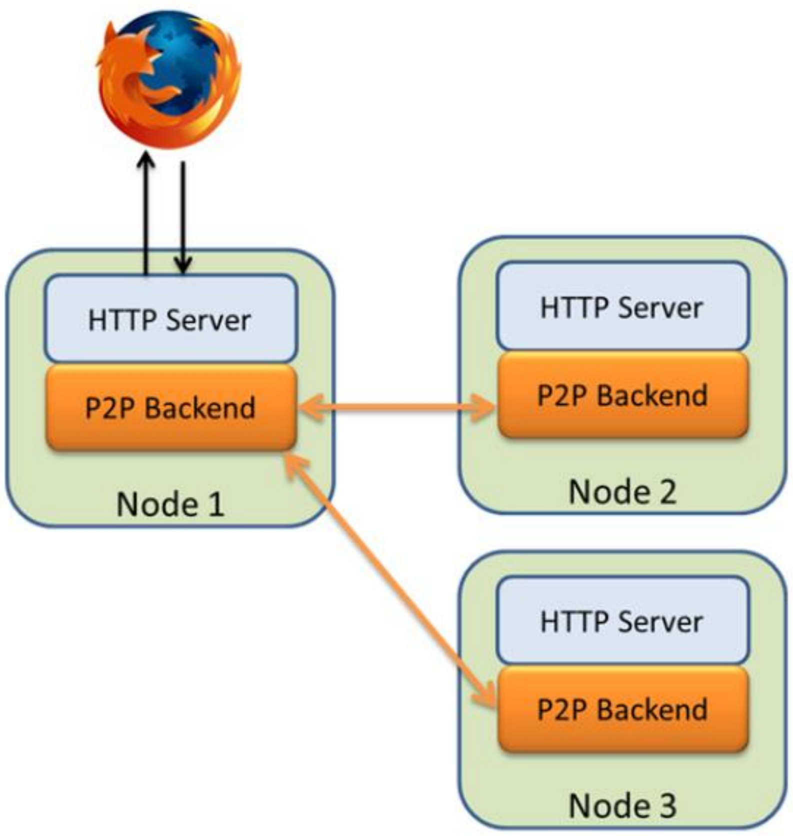


- Two valid bit sequences have a minimum distance of 3 bit flips
- Spheres of distance 1 around each codeword do not overlap
- If a single error occurs, the resulting n-tuple will be in a unique sphere around the original codeword
- Thus, receiver can correct erroneous reception back to original codeword

Project 2

- Develop a UDP backend

Develop a backend



- Hopefully, you have developed an operational frontend by now ☺
- Let's get to more complicated things…
- Develop a transport layer mechanism to implement a backend protocol
- Based on UDP
- Share content with friends (Similar to clearing each other's doubts on Canvas/Discord)

What's new?

- First need an extra port for backend protocol

  java vodServer 8345 -> java vodServer 8345 8346

- Support new URIs!

Add Content URI

/peer/add?path=<contentpath>&host=<IP/Hostname>&port=<backendport>&rate=<kbps>

- This URI let the node knows that the specified remote content with the given path could be found on another node with the given IP address or hostname and port.

http://localhost:8345/peer/add?path=content/video.ogg&host=pi.ece.uw.edu&port=8346&rate=1600

- This URI tells the node that the file “content/video.ogg” could be found on pi.ece.uw.edu. The content path is relative to the directory where we start the peer. The back-end port is 8346. The average bit rate for the content is 1600 kilobits per second.

View Content URI

/peer/view/<contentpath>

- This URI could be called by a web browser to retrieve the remote content using the specified path.

- After we enter this URIs (use the browser)

http://localhost:8345/peer/add?path=content/video.ogg&host=pi.ece.cmu.edu&port=8346

Then this URI should (immediately) play the video.

http://localhost:8345/peer/view/content/video.ogg

Configuration URI

/peer/config?rate=<bytes/s>

- This URI specified the bandwidth limit (in bytes per second) that the backend transport can use for content transfer.

http://localhost:8345/peer/config?rate=8192

- Set a bandwidth limit on the backend transport to 8192 bytes/s.

Status URI [Optional]

/peer/status

- This URI displays the average rate till now for the transfers on this backend port and IP address.

http://localhost:8345/peer/status

- Displays the average rate till now for the transfers on this backend port and IP address

Requirements

- ***Multiplex:*** You are allowed to use only one port for the backend transport (include sending & receiving). All backend traffic (even simultaneous transfers) should go through this port.
- ***Robustness:*** Can reliably transfer the content, that is it can endure at least 5% packet loss
- ***Bandwidth Limit:*** Can maintain the bandwidth specified in /peer/config option (can stay within 10% of the maximum bandwidth)
- ***Simultaneous transfer***: Can simultaneously transfer multiple (at least five) content requests at the same time.
- ***Content chunking***: When the same content is found on multiple peers (by sending multiple /peer/add requests with the same content path), the backend should be able to retrieve different parts of the content from different peers.

Summary: How do I start?

- Start simple: UDP echoserver + client..
  - Transfer a file (you already know how to!)
- Create a header to send appropriate packets, multiplex, add sequence #, field for ACKs, etc.
- Implement flow control (window, ACKs)
- +Congestion control (updates to window)

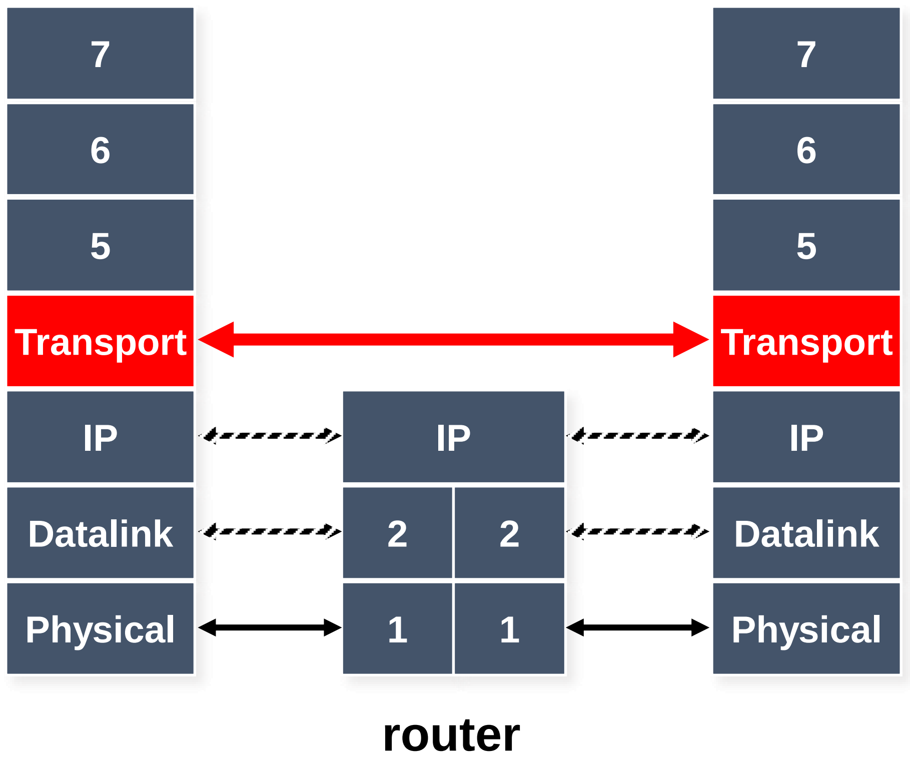
- Note: Also do URI parsing, frontend-backend comm., etc..

Outline

- What is the transport layer?
- Create/Destroy connection
- Flow control + Error recovery
- Congestion control

Transport Protocols

- Lowest level end-to-end protocol (only TX, RX involved)
- Routers don't participate!



What does it do?

- **Demultiplexing**: If there are 3 flows, how do I not mix them up? → Port #
- **Error detection:** If a packet is received incorrectly, how do I know? → checksum
- **Error recovery:** If an error happens, retransmit → ACKs+retransmit
- **Message boundaries** → Length field
- **In-order Delivery:** → Sequence #

Wait, I thought the MAC does this already?

→ *Only per-link, transport is end-to-end*

What does it do?

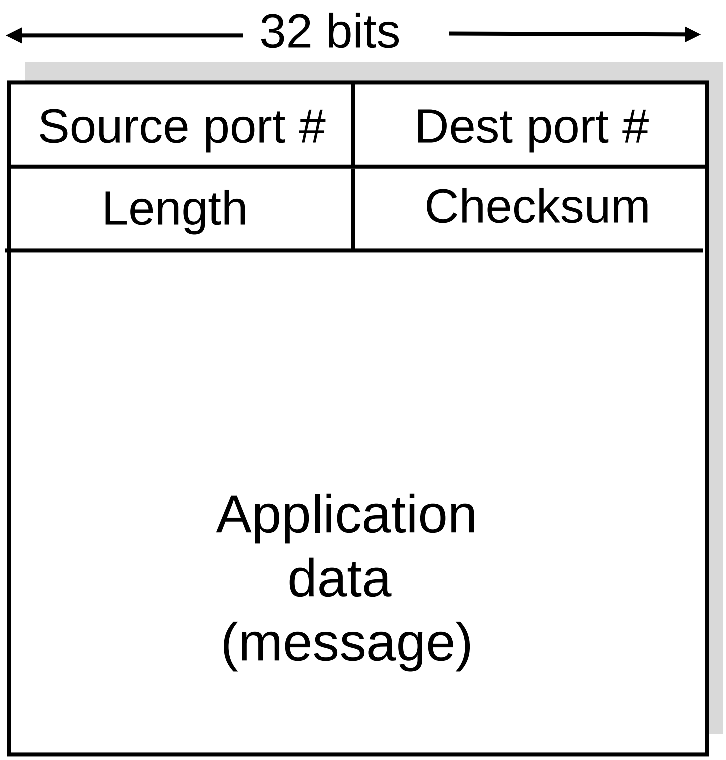
UDP

- **Demultiplexing**: If there are 3 flows, how do I not mix them up? → Port #
- **Error detection:** If a packet is received incorrectly, how do I know? → checksum
- **Error recovery:** If an error happens, retransmit → ACKs+retransmit
- **Message boundaries** → Length field
- **In-order Delivery:** → Sequence #

TCP

UDP: User Datagram Protocol [RFC 768]

- "No frills," "bare bones" Internet transport protocol
- Demultiplexing based on ports
- Optional checksum
  - One's complement add (weak)



UDP segment format

UDP doesn’t give a …!

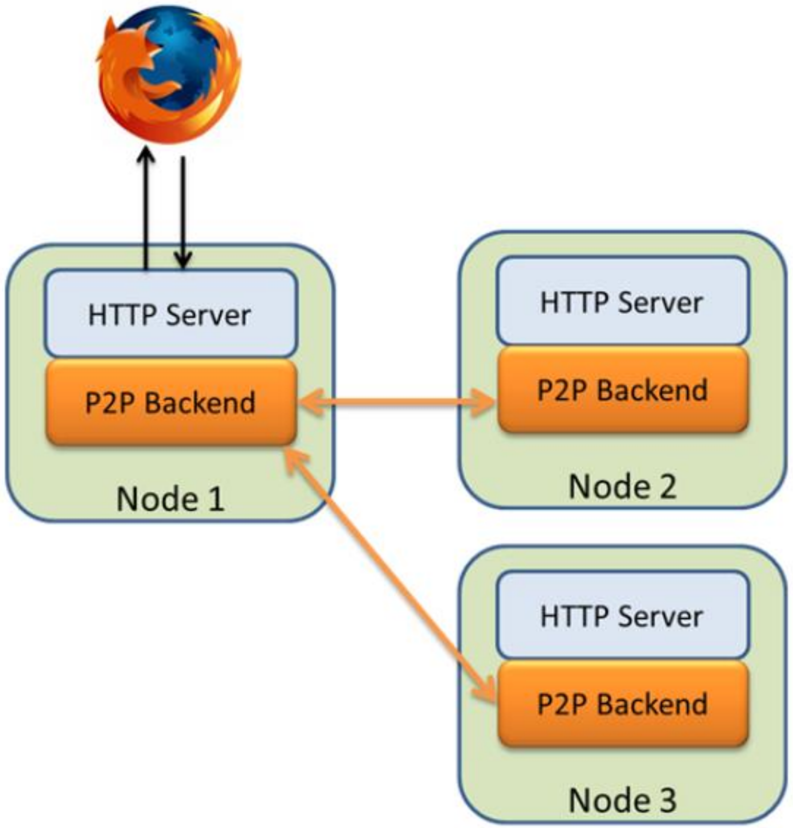
- Loss … Not its problem?
- Reordering… Doesn’t deal with it!

- Then what does it do?
  - Multiplexing + (Optional) Checksum
- Why is it useful?
  - Simple tasks (send 1 packet, e.g. beacon)
  - Building block for more complex protocols (e.g. project-2!)

Project-2

- **Main task:** you need to develop a reliable transport protocol over UDP between backends (today)

- *Other stuff*: URI parsing, coordinating between HTTP server & backend (libraries)



Ok.. What do I need to build?

- "Something like TCP over UDP.. (you can get away with less.. TCP-lite)"

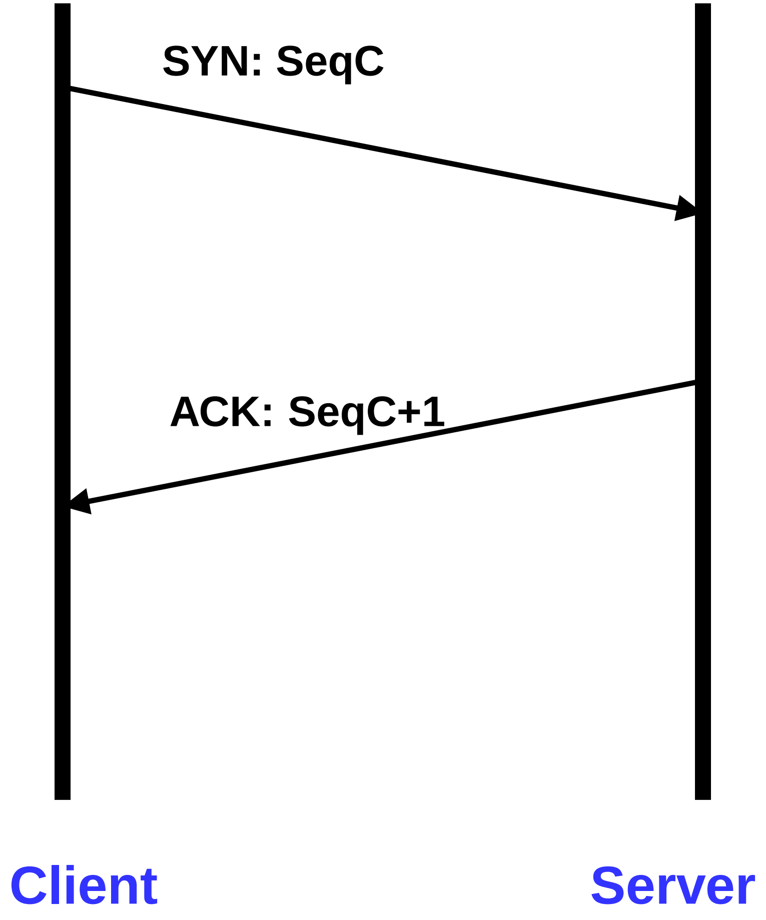
1. Create/Destroy connection
2. Flow-control (in-order, error recovery)
3. Congestion control

1. Create/Destroy Connection

Establishing 1-way Connection: Two-Way handshake

- Each side notifies other of starting sequence number it will use for sending
  - Why not simply chose 0?
    - Must avoid overlap with earlier incarnation
    - Security issues

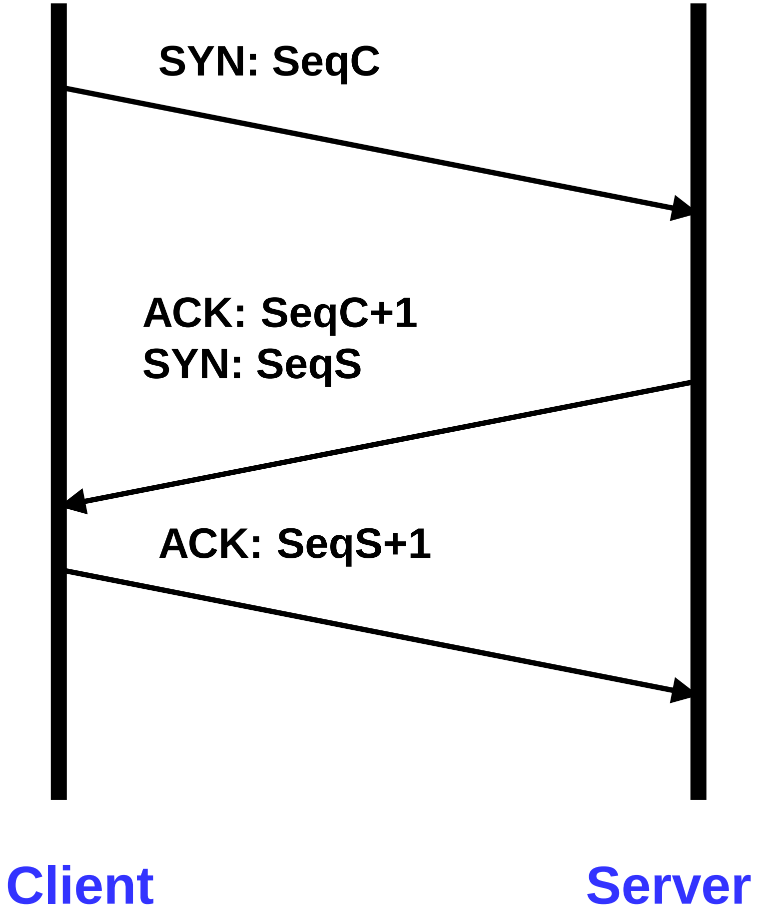
- Each side acknowledges other's sequence number
  - SYN-ACK: Acknowledge sequence number + 1



1. Create/Destroy Connection

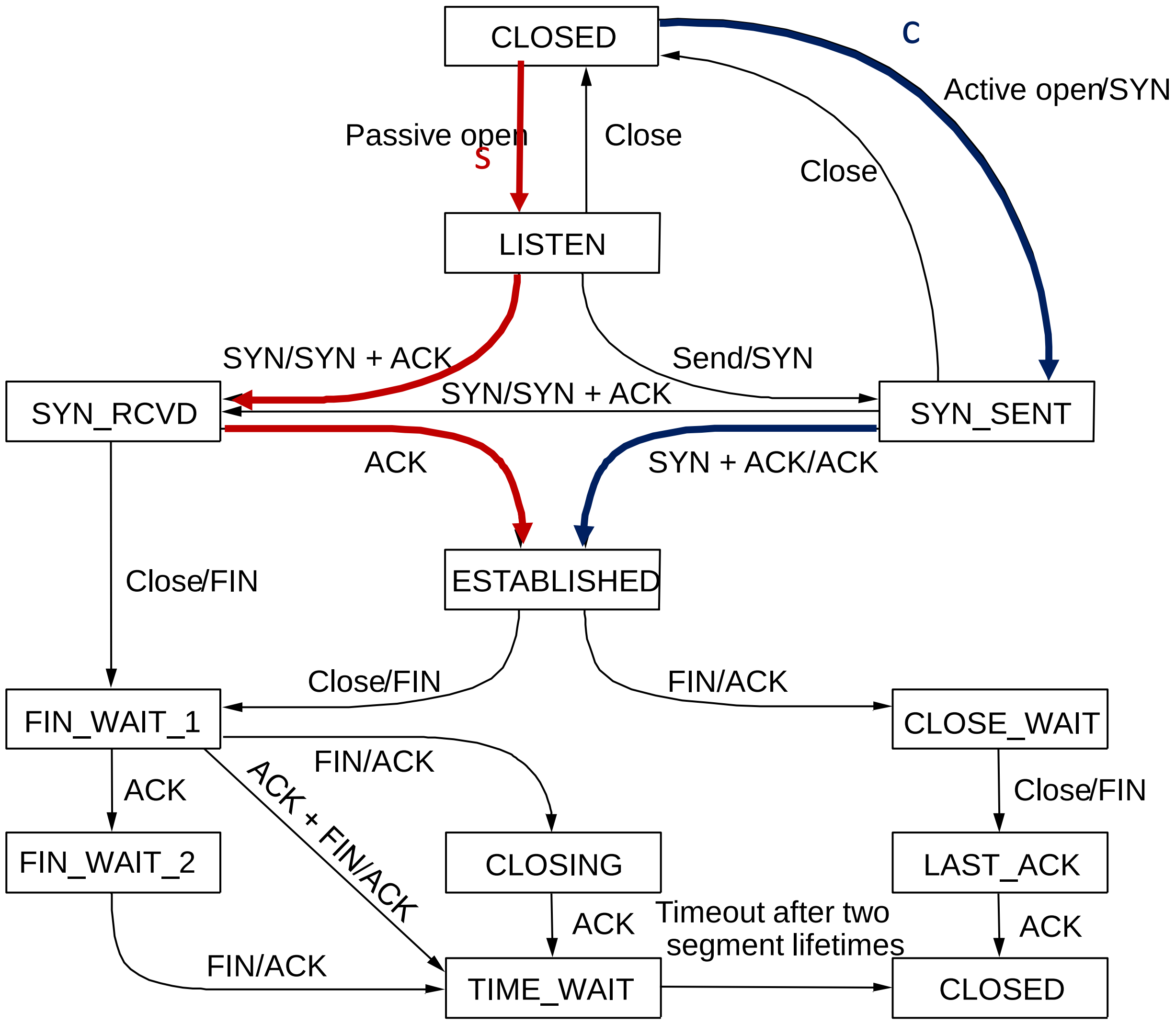
Establishing 2-way Connection: Three-Way handshake

- Optimize by combining second SYN with first ACK
- Lots of things can go wrong!
  - SYN lost
  - SYN-ACK lost
  - ACK lost
  - .. Time-out meaningfully



1. Create/Destroy Connection

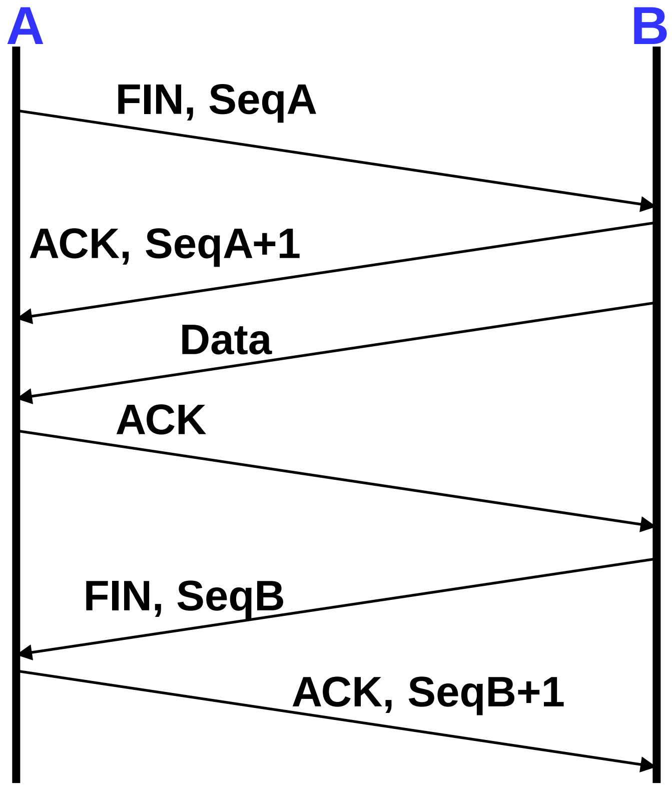
TCP State Diagram: Connection Setup



1. Create/Destroy Connection

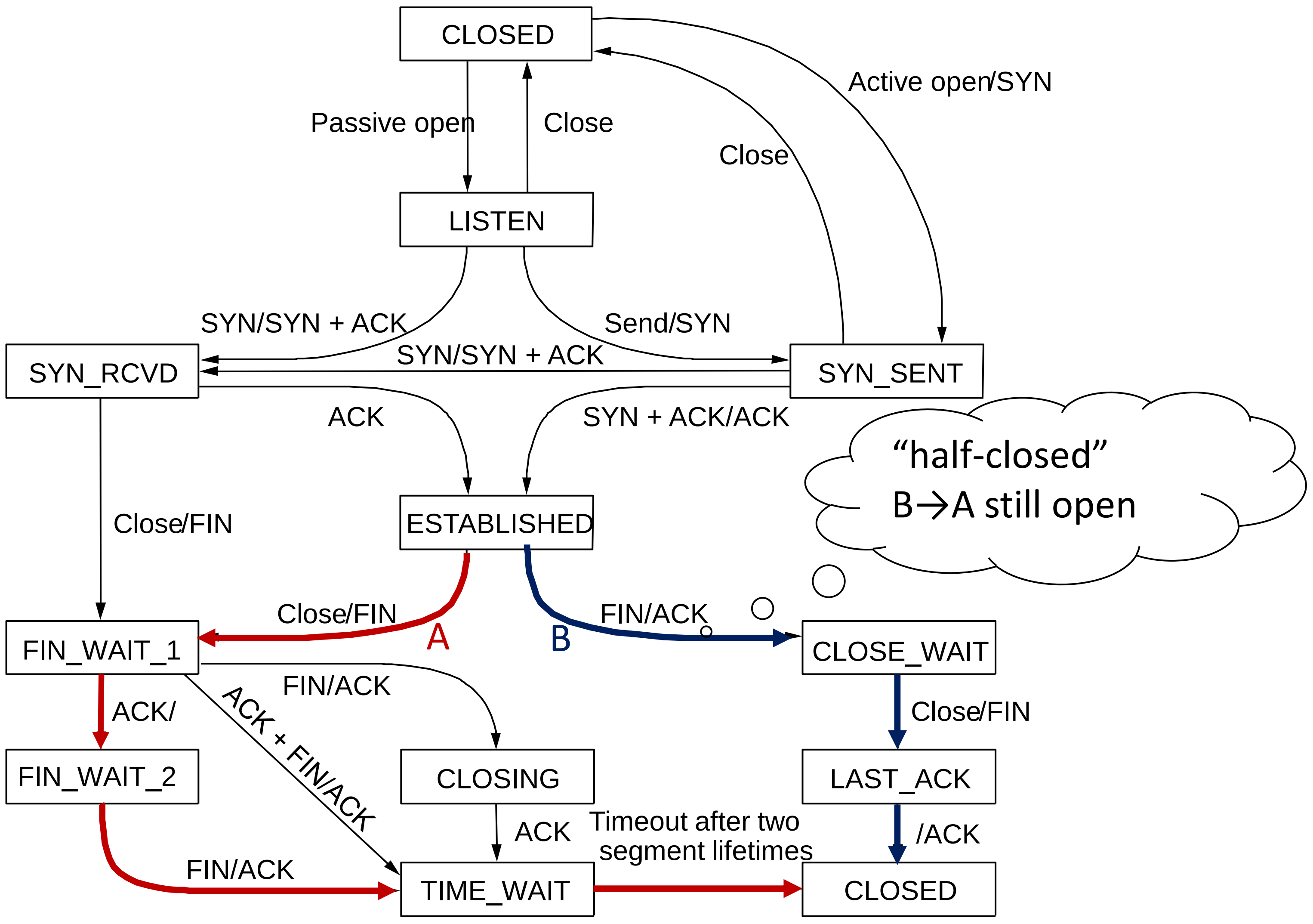
Tearing Down Connection

- Either side can initiate tear down
  - Send FIN signal
  - “I’m not going to send any more data”
- Other side can continue sending data
  - Half open connection
  - Must continue to acknowledge
- Acknowledging FIN
  - Acknowledge last sequence number + 1



1. Create/Destroy Connection

TCP State Diagram: Tearing Down Connection



Ok.. What do I need to build?

- “Something like TCP over UDP.. (you can get away with less.. TCP-lite)”

1. Create/Destroy connection
2. Flow-control (in-order, error recovery)
3. Congestion control

Ok.. What do I need to build?

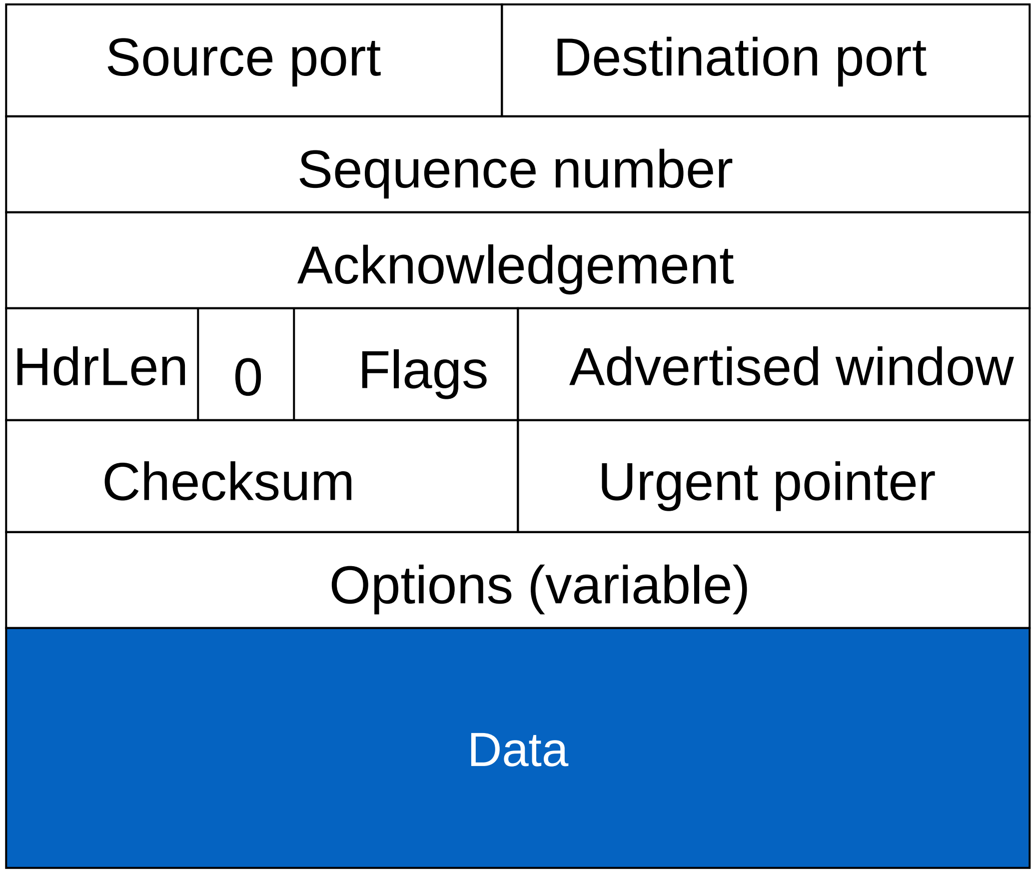
- “Something like TCP over UDP.. (you can get away with less.. TCP-lite)”

1. Create/Destroy connection
2. Flow-control (in-order, error recovery)
3. Congestion control

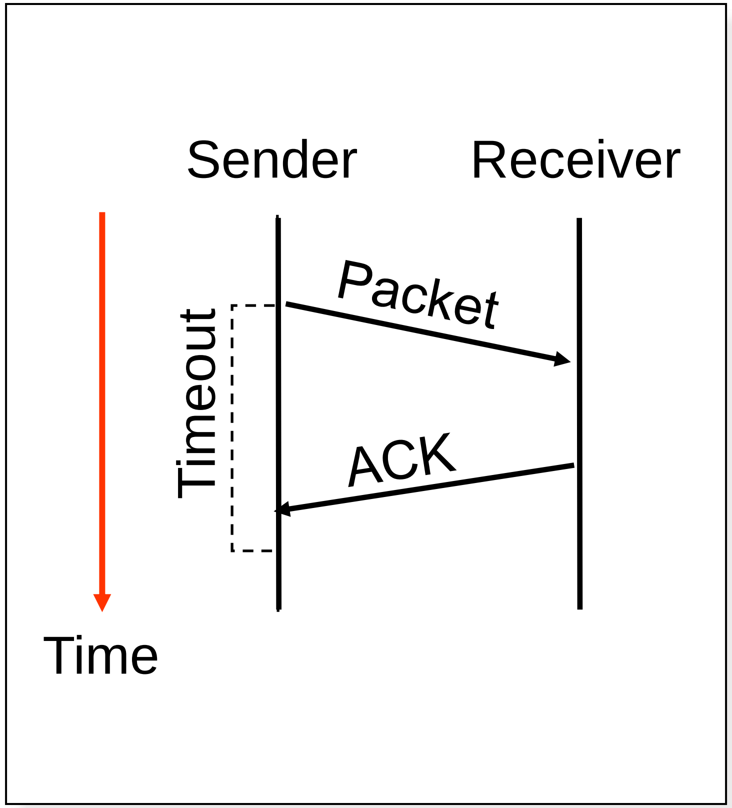
Goals here..

- Ensure in-order delivery…
  → Sequence numbers
- Ensure you know packets are lost
  → ACKs
- Recover from loss → Retransmit

Example: TCP Header

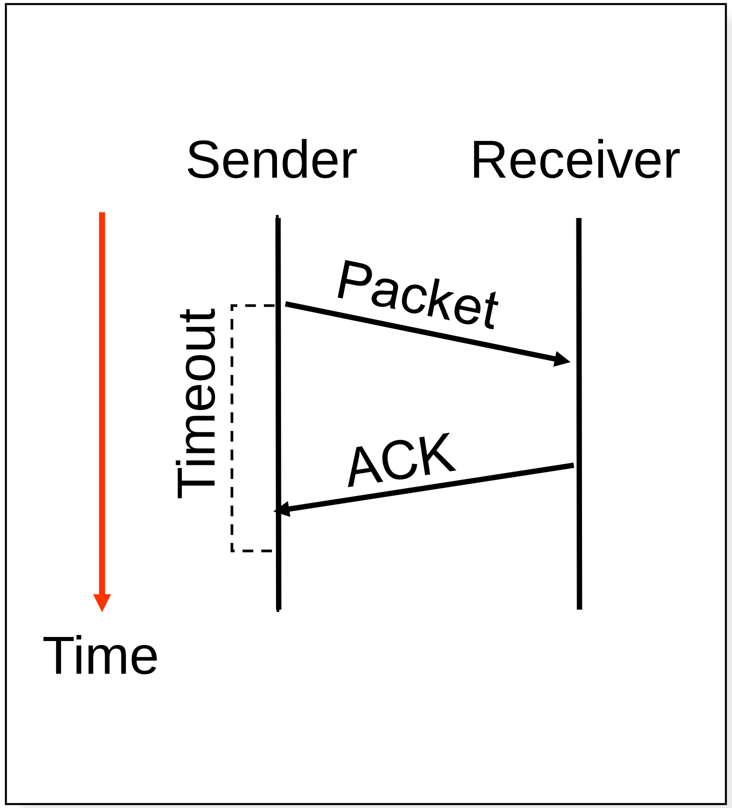


ACKs are important



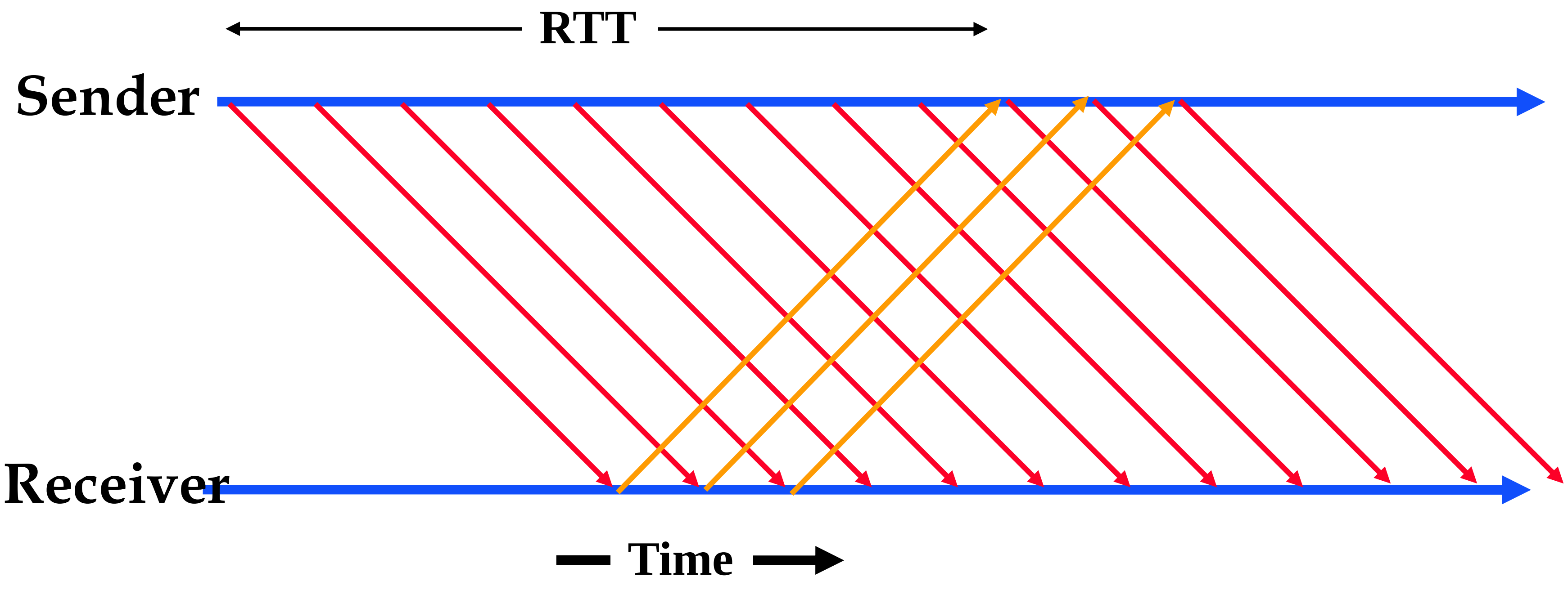
Review: Stop and Wait is BAD

- Simplest ARQ protocol
- Send a packet, stop and wait until ACK arrives
- Inefficient!



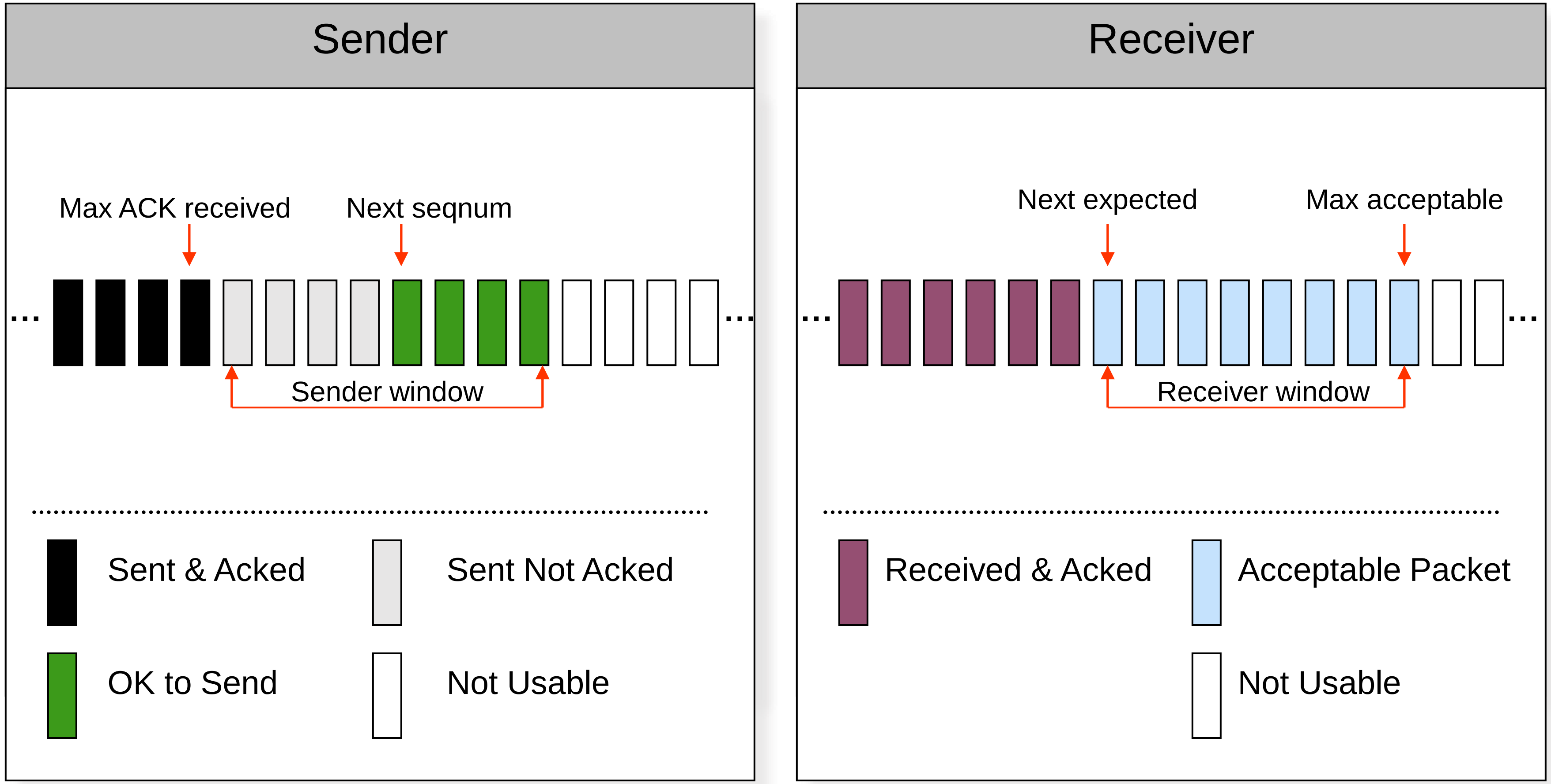
Correct approach:

Window = Bandwidth-Delay Product



$$\text{Max Throughput} = \frac{\text{Window Size}}{\text{Roundtrip Time}}$$

Sliding Window
Sender/Receiver State



Sliding Window Algorithm 101

- @TX: If new ACK is received (in order)
  - Increment MAX ack received
  - Send next packet
- @RX: If new pkt is received (in order)
  - Increment next expected
  - Increment MAX acceptable
- What about loss / out-of-order?
  - Many options! (up to you)
  - <u>Simplest</u>: Both TX & RX share their queue state

Important Detail: Timeout = ?

- Wait at least one RTT before retransmitting
- Importance of accurate RTT estimators:
  - Low RTT estimate
    - unneeded retransmissions
  - High RTT estimate
    - poor throughput
- RTT estimator must adapt to change in RTT
- How to pick timeout?
  - Up to you!
  - First cut: Estimate based on history.. Be conservative

Ok.. What do I need to build?

- “Something like TCP over UDP.. (you can get away with less.. TCP-lite)”

1. Create/Destroy connection
2. Flow-control (in-order, error recovery)
3. Congestion control

Congestion Control

How do I pick window?

- Too small → Underutilization
- Too big → Congestion!
- "Just right" … but how do we find it?

Idea: Additive Increase Multiplicative Decrease (AIMD)

- Initialize Window

- Every RTT…
  - If all packets received: W = W + 1
  - If any packet dropped: W = W/2

- Why not: MIAD or AIAD or MIMD?

Proof by Example

- Assume two users W_0=1, W_1=5… max 10 packets in n.w.
- AIAD:

  (1,5)->(2,6)->(3,7)->(4,8)->(3,7)->(4,8)->…(repeat)
- MIAD:

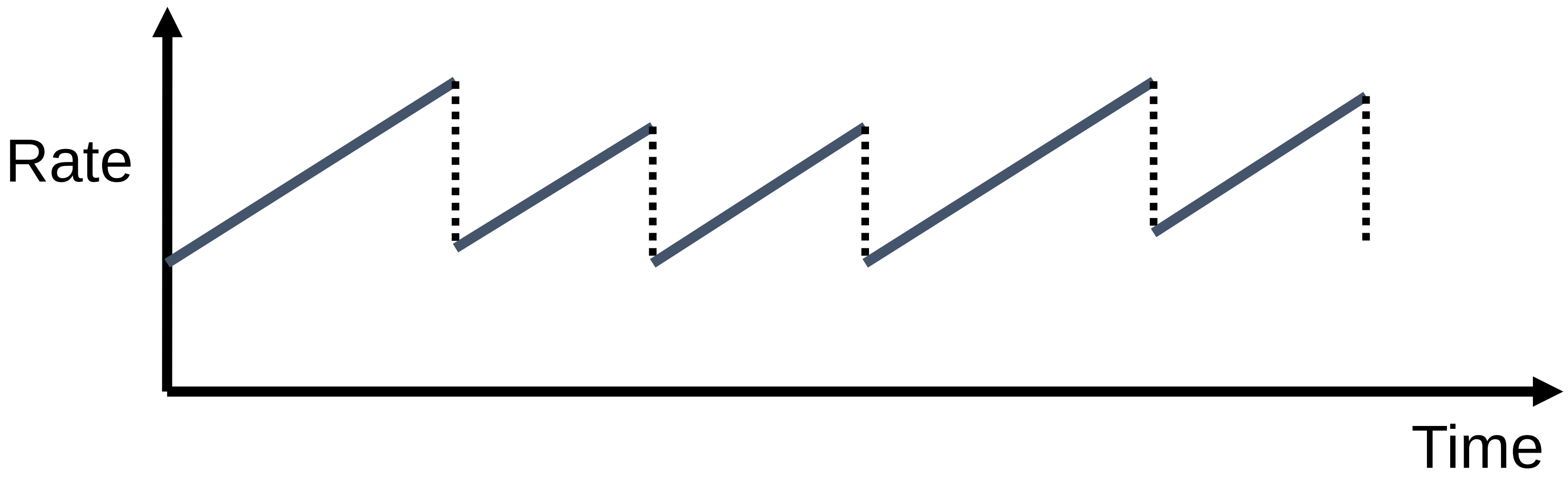
  (1,5)->(2,10)->(1,9)->(2,18)->(0,19)->….(congest!!)
- MIMD:

  (1,5)->(2,10)->(1,5)->(2,10)->….(repeat)
- AIMD:

  (1,5)->(2,6)->(3,7)->(4,8)->(2,4)->(3,5)->(4,6)->(5,7)->
  (3,4)->(4,5)->(5,6)->(3,3)->(4,4,)->**(5,5)**->(6,6)->repeat

TCP Congestion Control: Implicit Feedback and AIMD

- Distributed, fair and efficient
- Packet loss is seen as sign of congestion and results in a multiplicative rate decrease: factor of 2
- TCP periodically probes for available bandwidth by increasing its rate: by one packet per RTT



Implementing in practice

- Per packet..
  - W = W+1/W… (so W=W+1 in one RTT)
  - W = W/2 whenever a timeout occurs

- Many optimizations possible
  - Back-off if you detect loss from feedback
  - Increase W faster initially, e.g. W=2W in one RTT
  - Many more… (up to you!)

Ok.. What do I need to build?

- “Something like TCP over UDP.. (you can get away with less.. TCP-lite)”

1. Create/Destroy connection
2. Flow-control (in-order, error recovery)
3. Congestion control

Summary: How do I start?

- Start simple: UDP echoserver + client..
  - Transfer a file (you already know how to!)
- Create a header to send appropriate packets, multiplex, add sequence #, field for ACKs, etc.
- Implement flow control (window, ACKs)
- +Congestion control (updates to window)
- Note: Also do URI parsing, frontend-backend comm., etc..