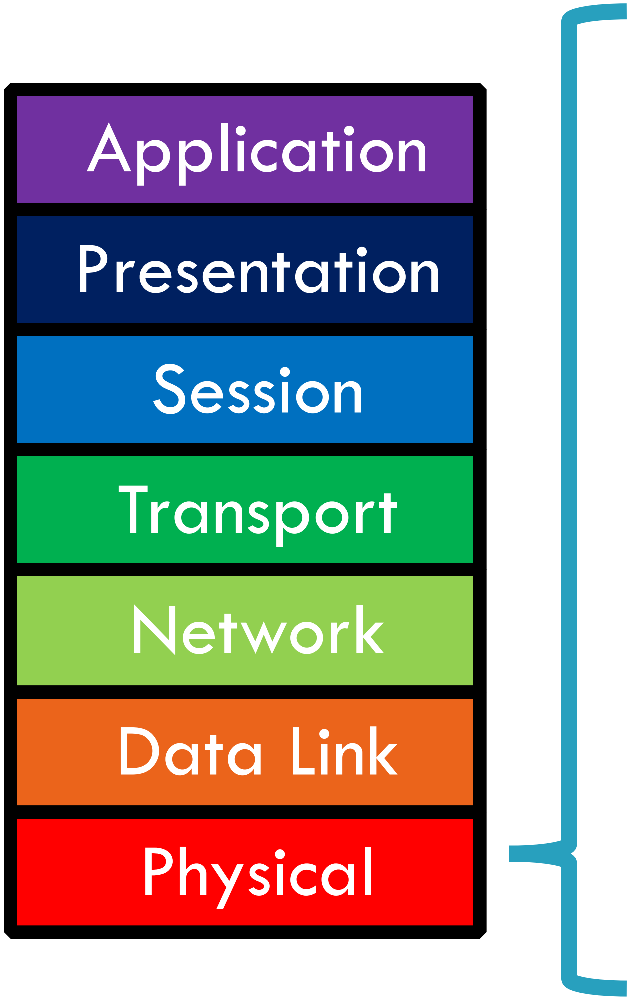
CS 3700

Networks and Distributed Systems

Lecture 3: Physical and Data Link

Revised 1/14/13

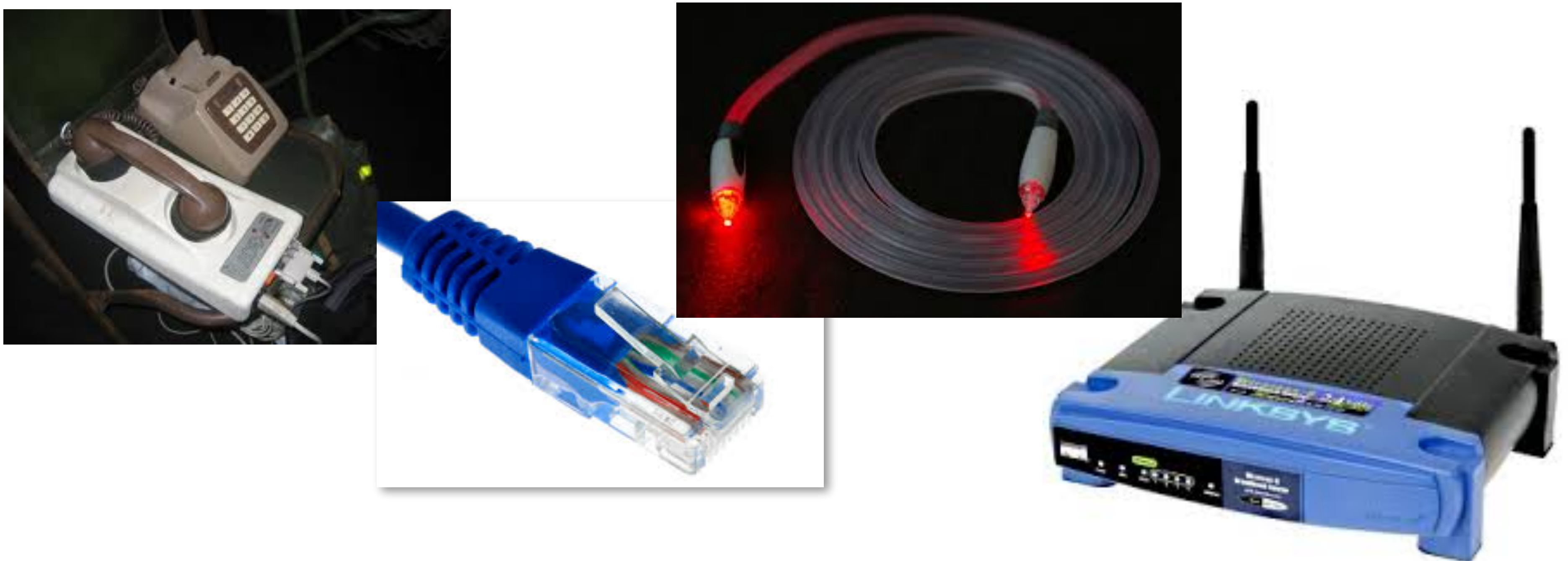
Physical Layer



- Function:
  - Get bits across a physical medium
- Key challenge:
  - How to represent bits in analog
  - Ideally, want high-bit rate
  - But, must avoid desynchronization

Key challenge

- Digital computers
  - 0s and 1s
- Analog world
  - Amplitudes and frequencies

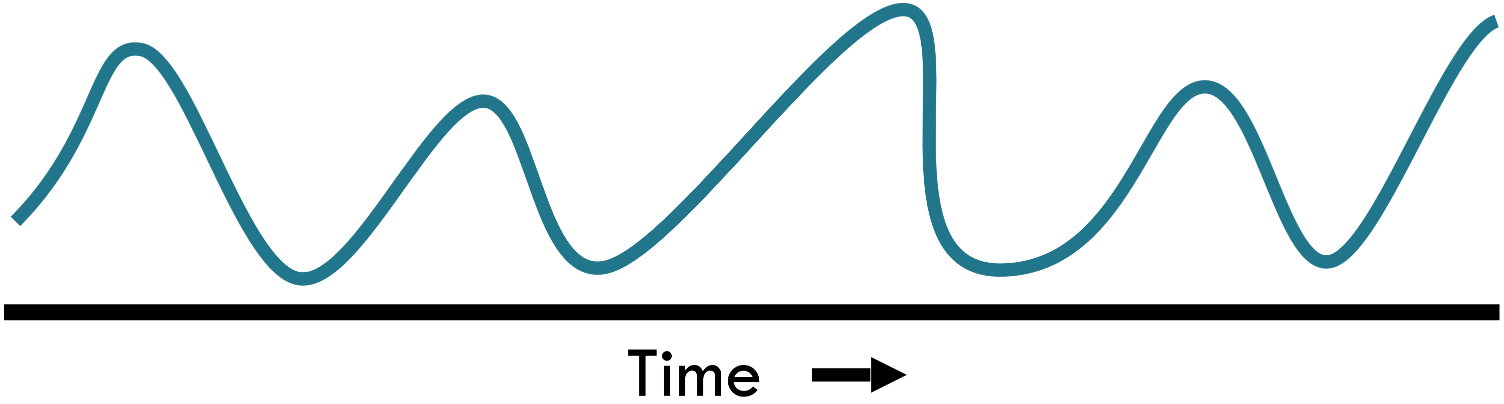


Assumptions

- We have two discrete signals, high and low, to encode 1 and 0

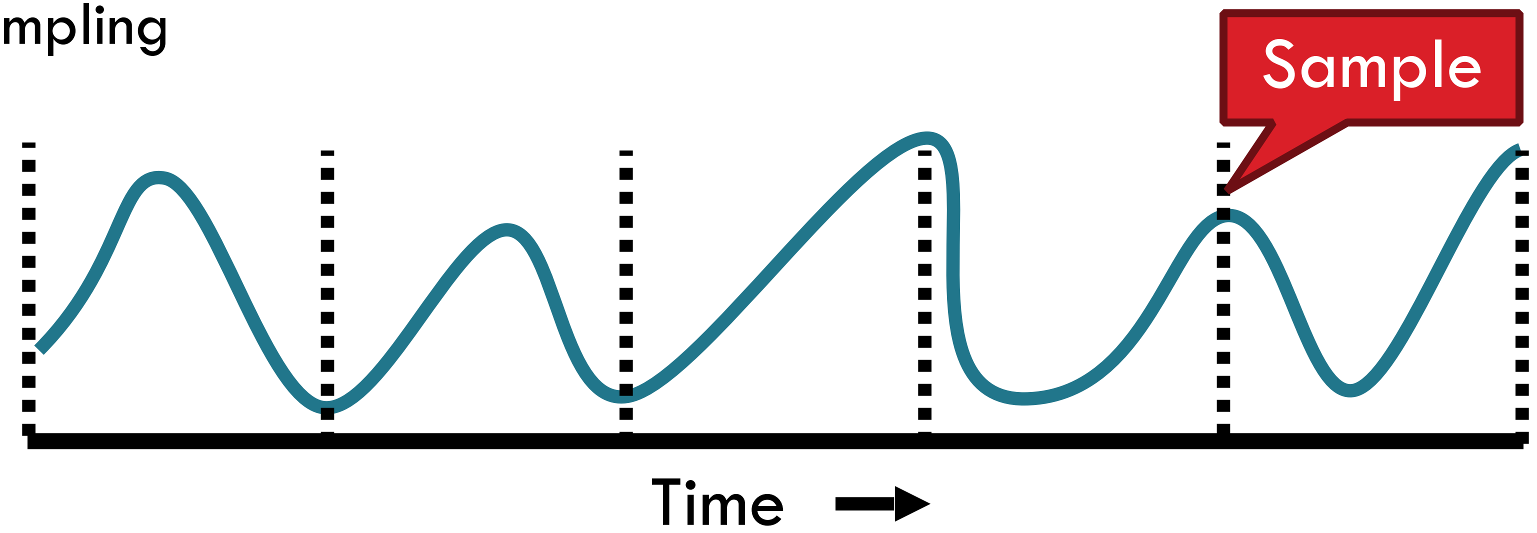
Assumptions

- We have two discrete signals, high and low, to encode 1 and 0
- Transmission is synchronous, i.e. there is a clock that controls signal sampling



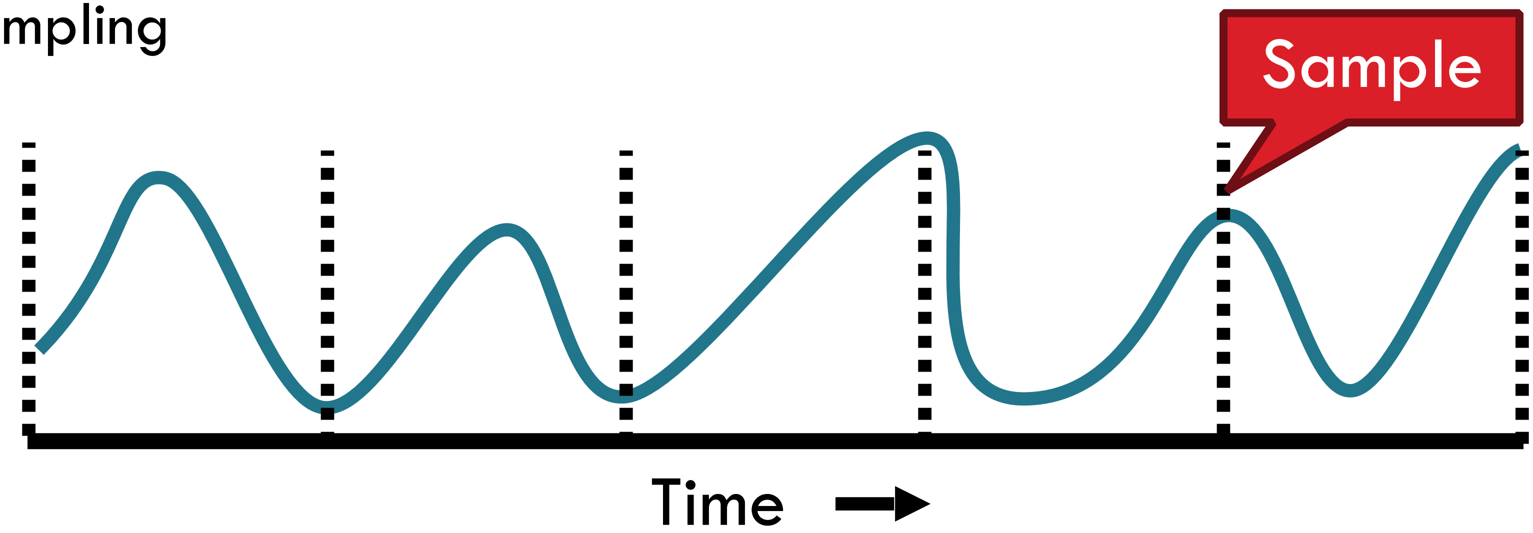
Assumptions

- We have two discrete signals, high and low, to encode 1 and 0
- Transmission is synchronous, i.e. there is a clock that controls signal sampling



Assumptions

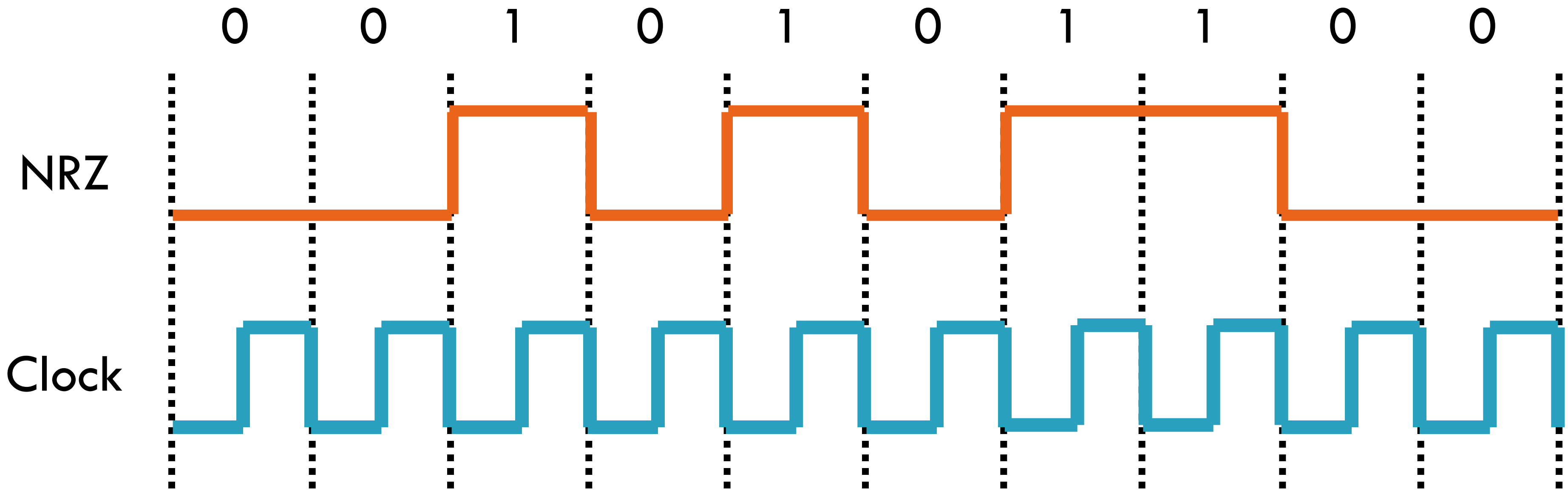
- We have two discrete signals, high and low, to encode 1 and 0
- Transmission is synchronous, i.e. there is a clock that controls signal sampling



- Amplitude and duration of signal must be significant

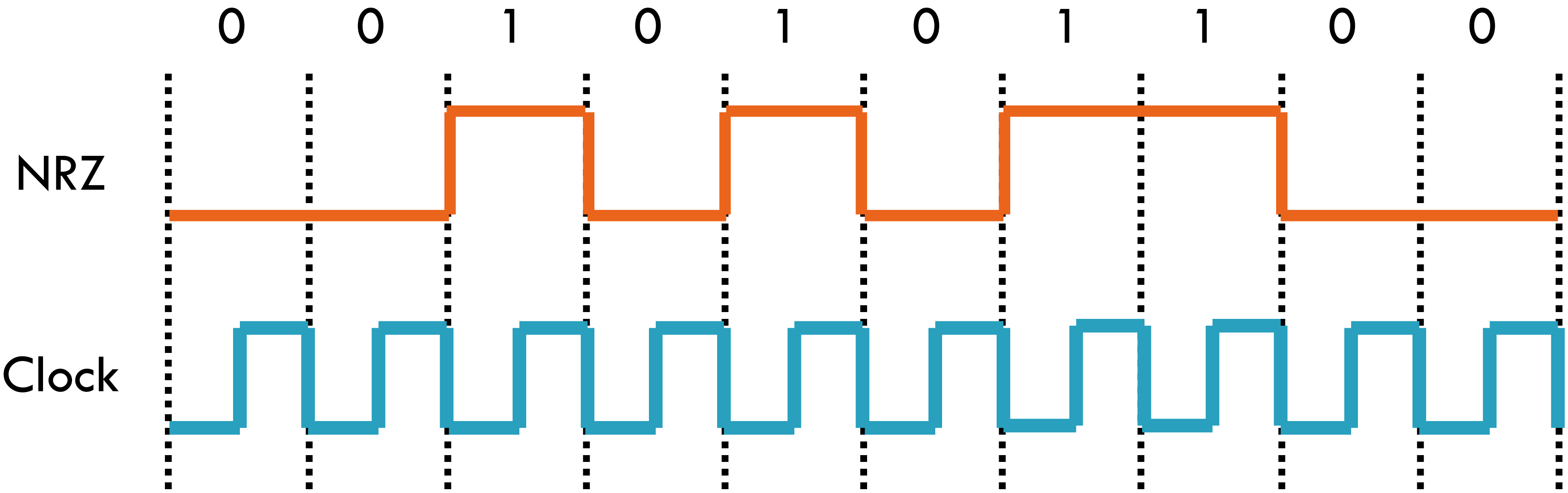
Non-Return to Zero (NRZ)

- 1 → high signal, 0 → low signal



Non-Return to Zero (NRZ)

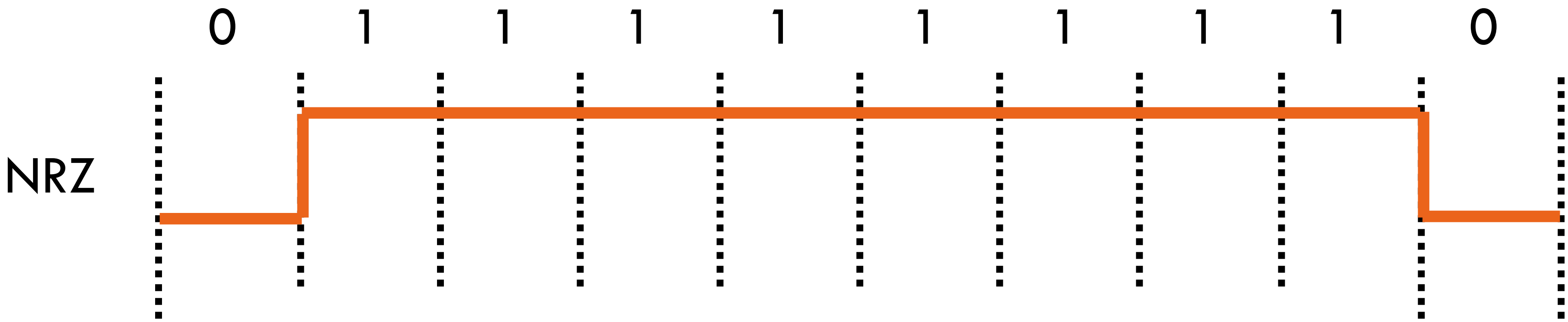
- 1 → high signal, 0 → low signal



- Problem: long strings of 0 or 1 cause desynchronization
  - How to distinguish lots of 0s from no signal?
  - How to recover the clock during lots of 1s?

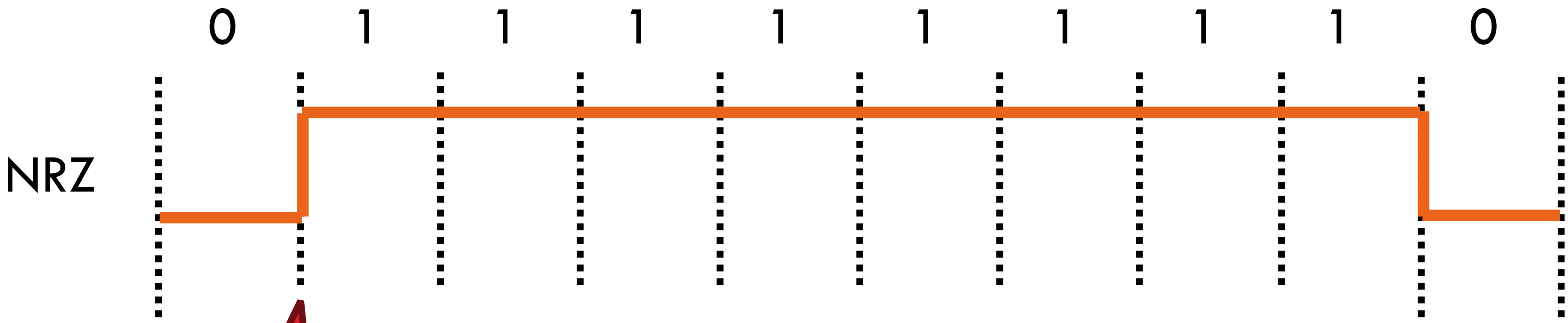
Desynchronization

- Problem: how to recover the clock during sequences of 0's or 1's?



Desynchronization

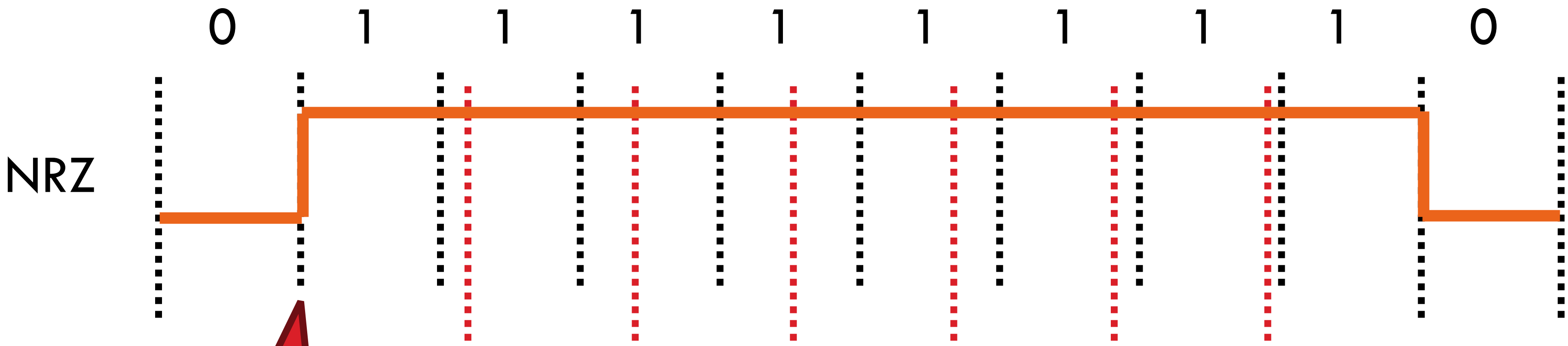
- Problem: how to recover the clock during sequences of 0's or 1's?



Transitions signify clock ticks

Desynchronization

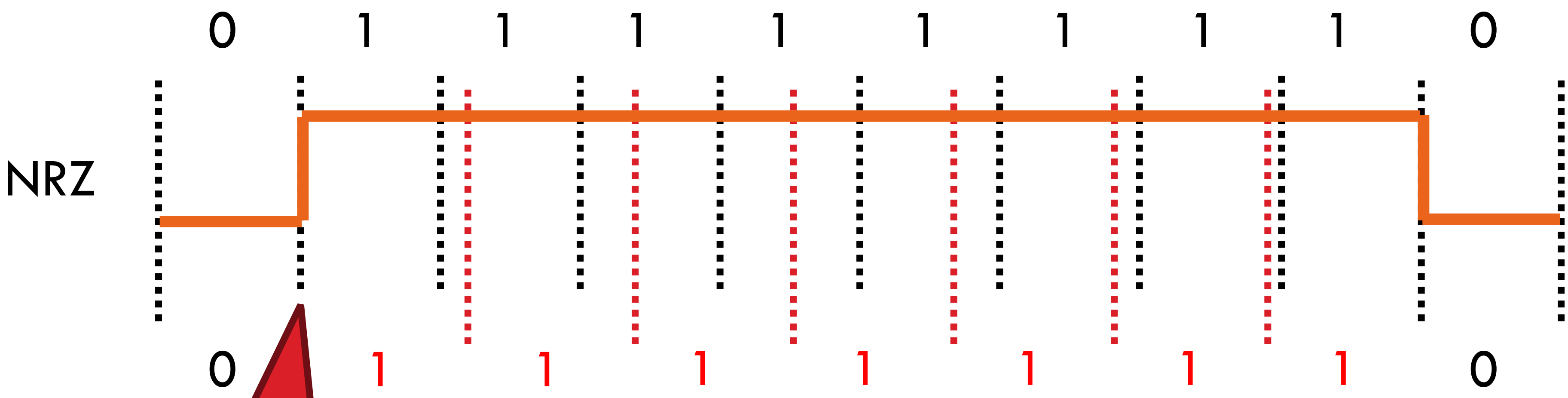
- Problem: how to recover the clock during sequences of 0's or 1's?



Transitions signify clock ticks

Desynchronization

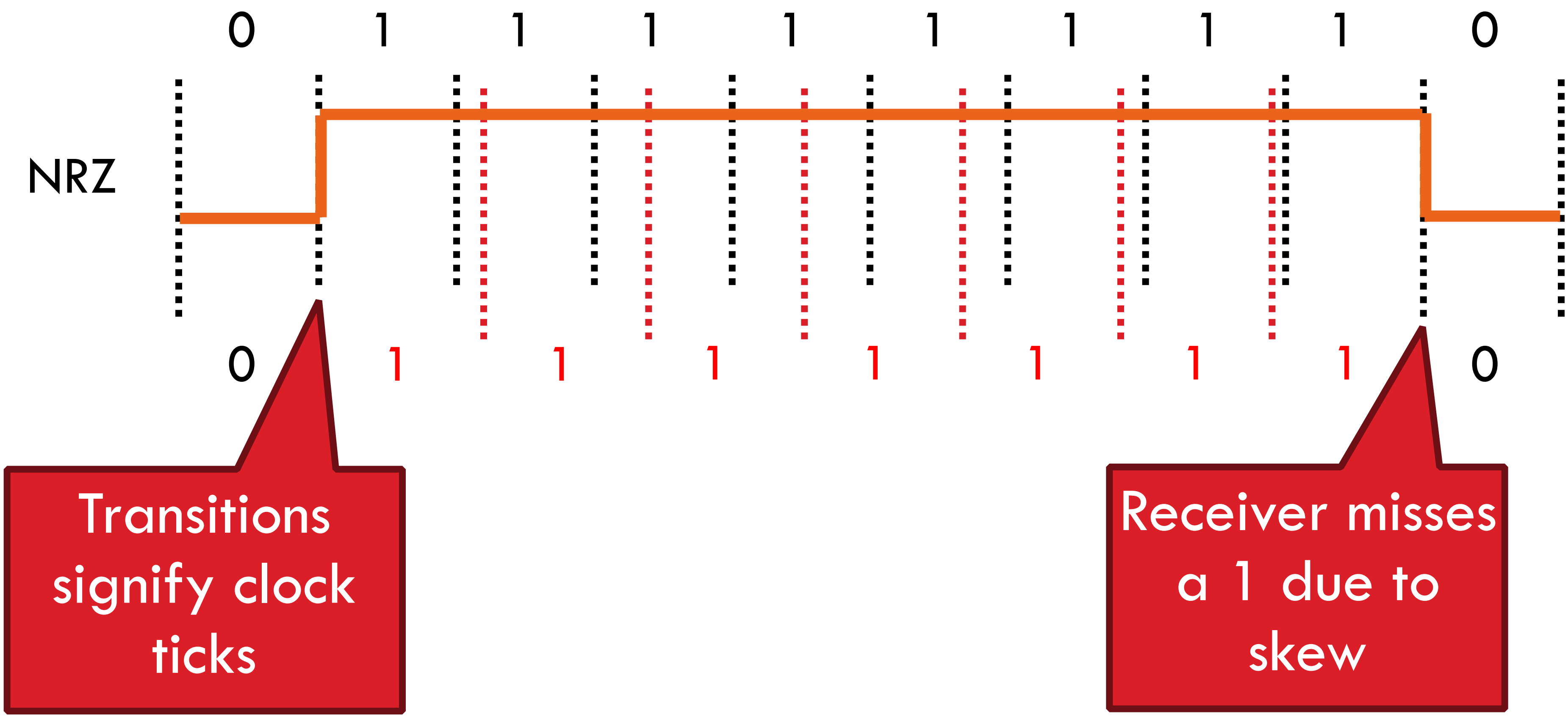
- Problem: how to recover the clock during sequences of 0's or 1's?



Transitions signify clock ticks

Desynchronization

- Problem: how to recover the clock during sequences of 0's or 1's?

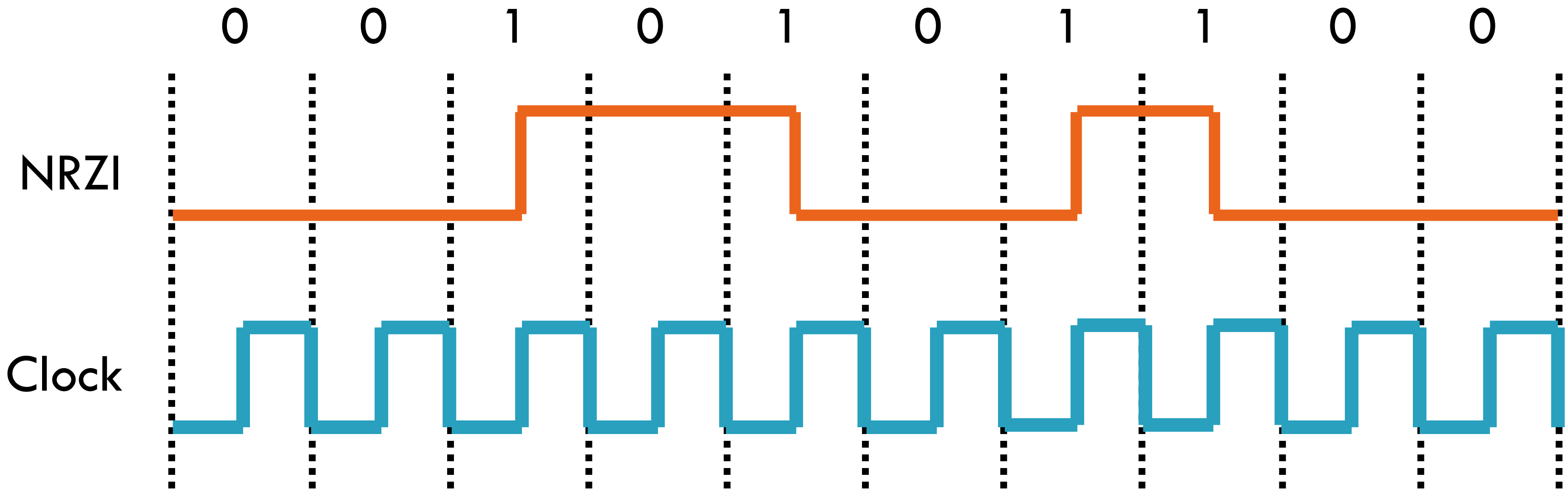


Transitions signify clock ticks

Receiver misses a 1 due to skew

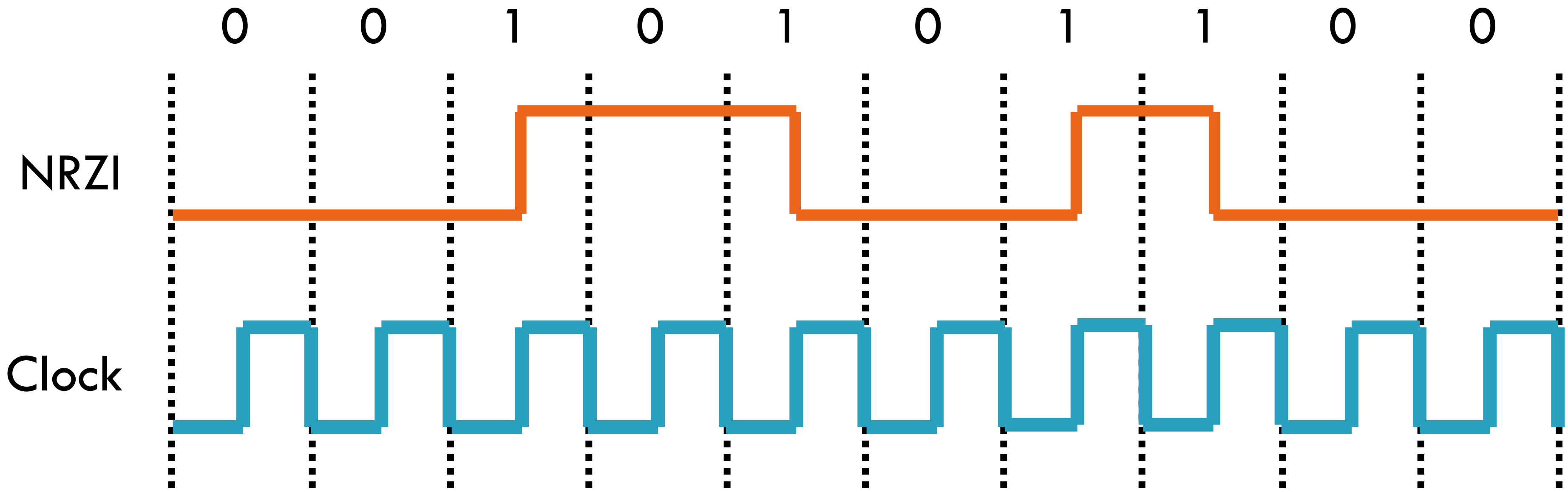
Non-Return to Zero Inverted (NRZI)

- 1 → make transition, 0 → remain the same



Non-Return to Zero Inverted (NRZI)

- 1 → make transition, 0 → remain the same



- Solves the problem for sequences of 1s, but not 0s

4-bit/5-bit (100 Mbps Ethernet)

4-bit	5-bit
0000	11110
0001	01001
0010	10100
0011	10101
0100	01010
0101	01011
0110	01110
0111	01111

4-bit	5-bit
1000	10010
1001	10011
1010	10110
1011	10111
1100	11010
1101	11011
1110	11100
1111	11101

4-bit/5-bit (100 Mbps Ethernet)

- Observation: NRZI works as long as no sequences of 0
- Idea: encode all 4-bit sequences as 5-bit sequences with no more than one leading 0 and two trailing 0

4-bit	5-bit
0000	11110
0001	01001
0010	10100
0011	10101
0100	01010
0101	01011
0110	01110
0111	01111

4-bit	5-bit
1000	10010
1001	10011
1010	10110
1011	10111
1100	11010
1101	11011
1110	11100
1111	11101

- Tradeoff: efficiency drops to 80%

4-bit/5-bit (100 Mbps Ethernet)

8-bit / 10-bit used in Gigabit Ethernet

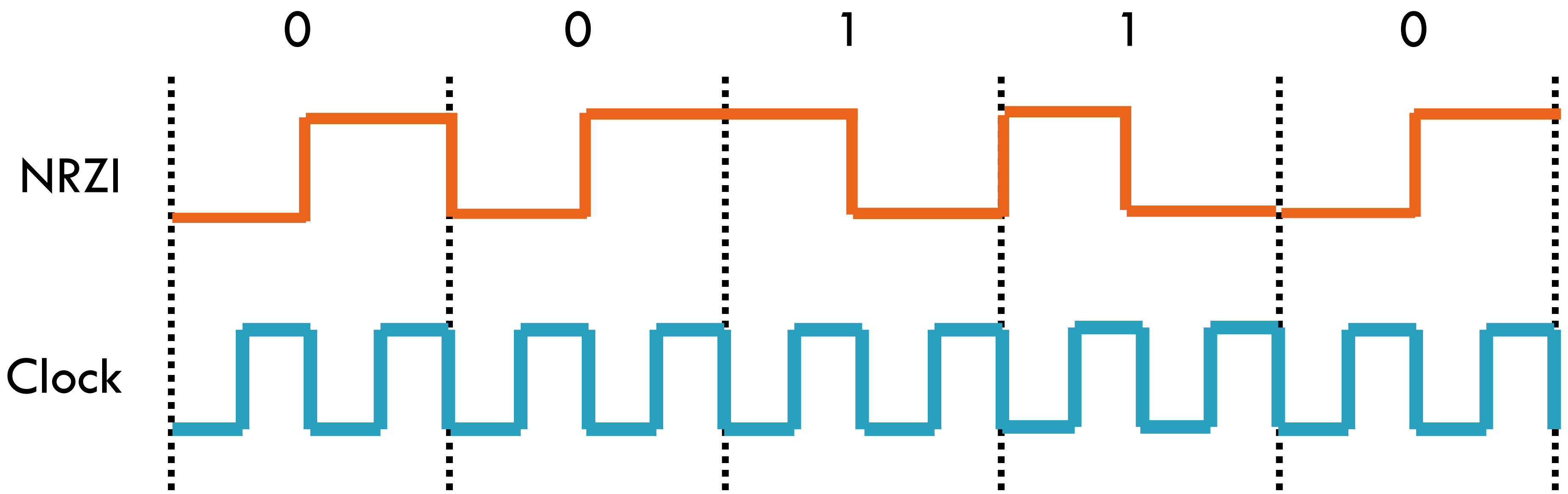
- Obser f 0
- Idea: encode all 4-bit sequences as 5-bit sequences with no more than one leading 0 and two trailing 0

4-bit	5-bit	4-bit	5-bit
0000	11110	1000	10010
0001	01001	1001	10011
0010	10100	1010	10110
0011	10101	1011	10111
0100	01010	1100	11010
0101	01011	1101	11011
0110	01110	1110	11100
0111	01111	1111	11101

- Tradeoff: efficiency drops to 80%

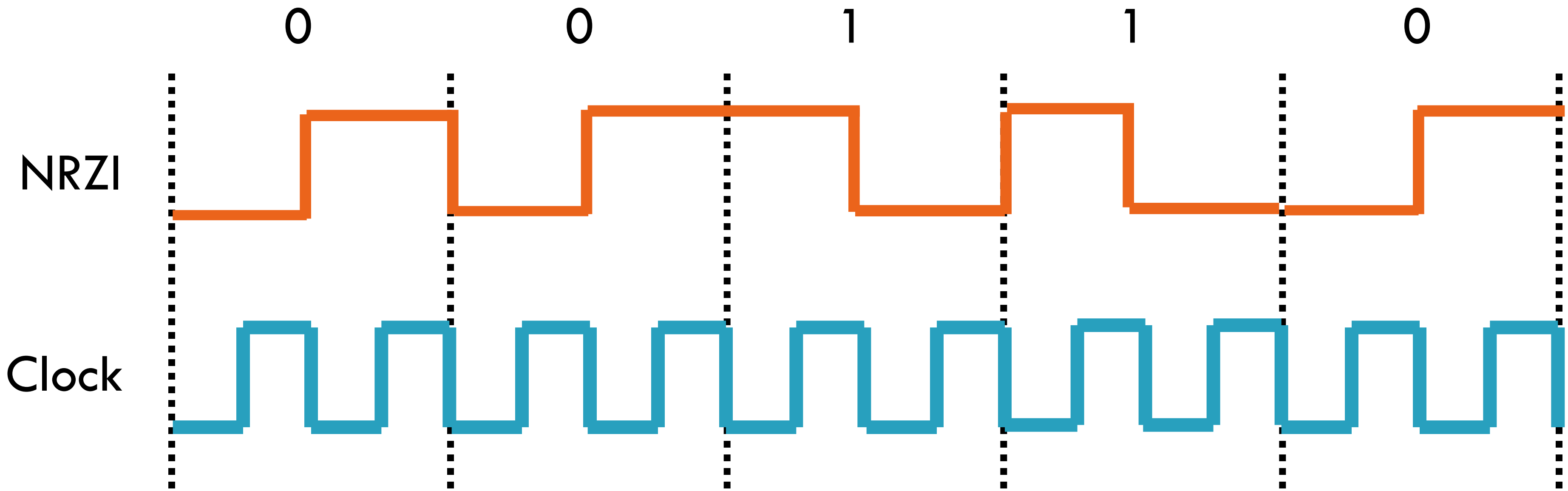
Manchester

- 1 → high-to-low, 0 → low-to-high



Manchester

- 1 → high-to-low, 0 → low-to-high



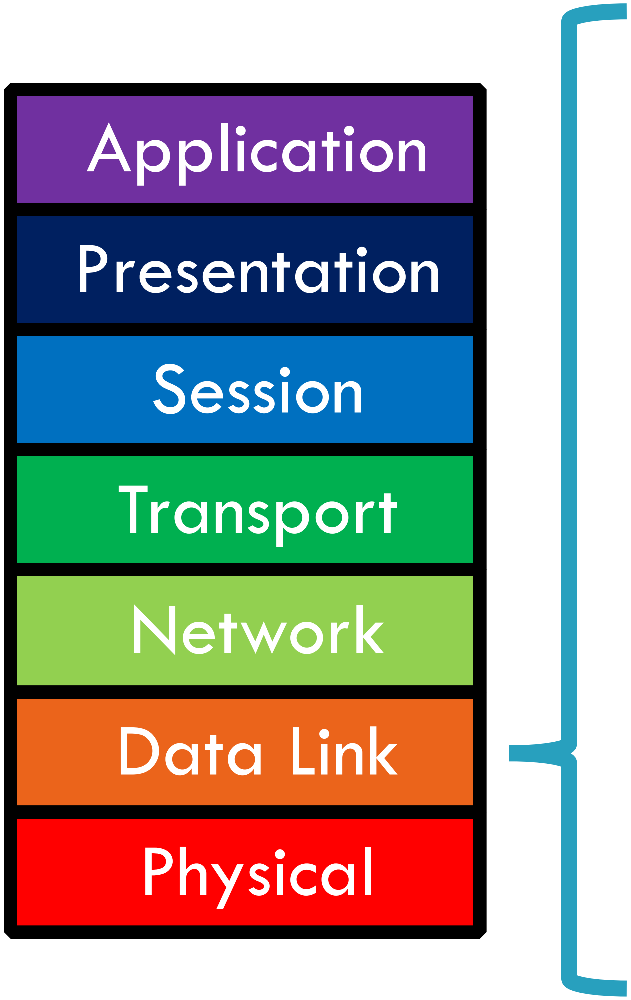
- Good: Solves clock skew (every bit is a transition)
- Bad: Halves throughput (two clock cycles per bit)

General comment

- Physical layer is the lowest, so…
  - We tend not to worry about where to place functionality
  - There aren't other layers that could interfere
  - We tend to care about it only when things go wrong

- Physical layer characteristics are still fundamentally important to building reliable Internet systems
  - Insulated media vs wireless
  - Packet vs. circuit switched media

Data Link Layer



- Function:
  - Send blocks of data (frames) between physical devices
  - Regulate access to the physical media
- Key challenge:
  - How to delineate frames?
  - How to detect errors?
  - How to perform media access control (MAC)?
  - How to recover from and avoid collisions?

Outline

- Framing
- Error Checking and Reliability

Framing

- Physical layer determines how bits are encoded
- Next step, how to encode blocks of data

Framing

- Physical layer determines how bits are encoded
- Next step, how to encode blocks of data
  - Packet switched networks
  - Each packet includes routing information
  - Data boundaries must be known so headers can be read

Framing

- Physical layer determines how bits are encoded
- Next step, how to encode blocks of data
  - Packet switched networks
  - Each packet includes routing information
  - Data boundaries must be known so headers can be read
- Types of framing
  - Byte oriented protocols
  - Bit oriented protocols
  - Clock based protocols

Byte Oriented: Sentinel Approach

Data

- Add **START** and **END** sentinels to the data

Byte Oriented: Sentinel Approach

- Add **START** and **END** sentinels to the data

Byte Oriented: Sentinel Approach

- Add **START** and **END** sentinels to the data
- Problem: what if **END** appears in the data?

Byte Oriented: Sentinel Approach

- Add **START** and **END** sentinels to the data
- Problem: what if **END** appears in the data?
  - Add a special **DLE** (Data Link Escape) character before **END**

Byte Oriented: Sentinel Approach

START | DLE | DLE | Data | DLE | END | END

- Add **START** and **END** sentinels to the data
- Problem: what if **END** appears in the data?
  - Add a special **DLE** (Data Link Escape) character before **END**
  - What if **DLE** appears in the data? Add **DLE** before it.

Byte Oriented: Sentinel Approach

START DLE DLE Data DLE END END

- Add **START** and **END** sentinels to the data
- Problem: what if **END** appears in the data?
  - Add a special **DLE** (Data Link Escape) character before **END**
  - What if **DLE** appears in the data? Add **DLE** before it.
  - Similar to escape sequences in C
    - printf(“You must \”escape\” quotes in strings”);
    - printf(“You must \\escape\\ forward slashes as well”);
- Used by Point-to-Point protocol, e.g. modem, DSL, cellular

Byte Oriented: Byte Counting

132

Data

Byte Oriented: Byte Counting

- Sender: insert length of the data in bytes at the beginning of each frame
- Receiver: extract the length and read that many bytes

Bit Oriented: Bit Stuffing

Data

Bit Oriented: Bit Stuffing

- Add sentinels to the start and end of data
  - Both sentinels are the same
  - Example: 01111110 in High-level Data Link Protocol (HDLC)

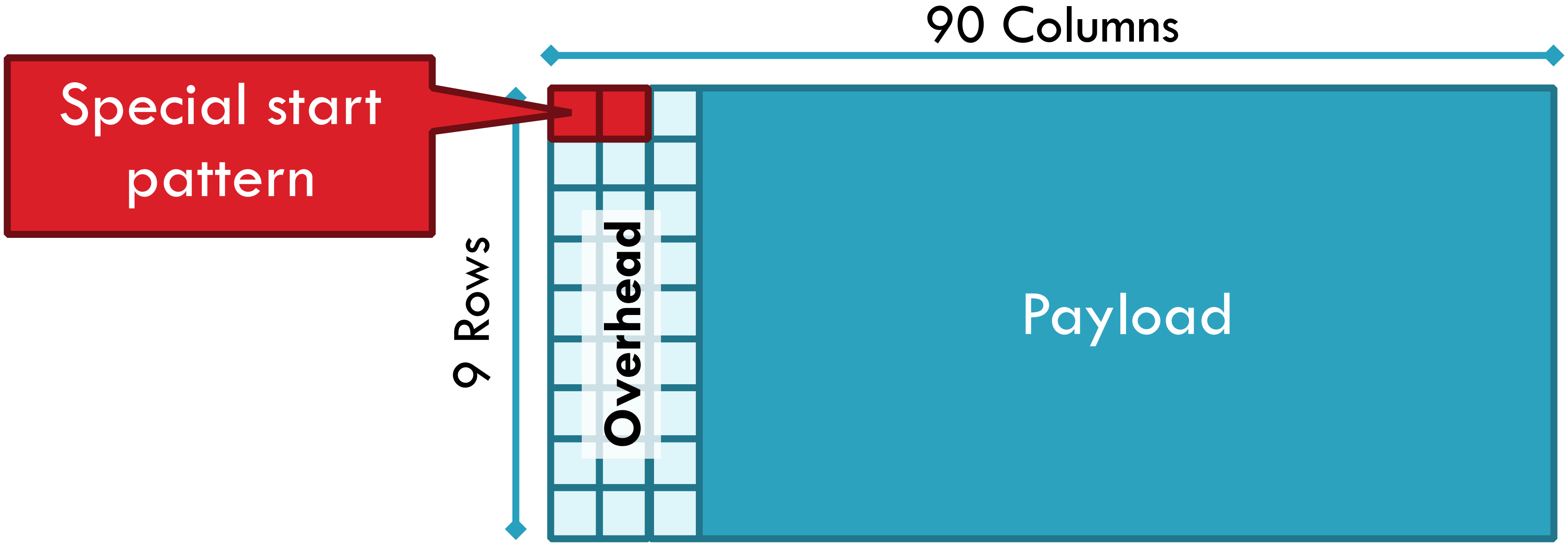
Bit Oriented: Bit Stuffing

01111110 Data 01111110

- Add sentinels to the start and end of data
  - Both sentinels are the same
  - Example: 01111110 in High-level Data Link Protocol (HDLC)
- Sender: insert a 0 after each 11111 in data
  - Known as “bit stuffing”
- Receiver: after seeing 11111 in the data…
  - 11111**0** → remove the 0 (it was stuffed)
  - 11111**1** → look at one more bit
    - 11111**10** → end of frame
    - 11111**11** → error! Discard the frame
- Disadvantage: 20% overhead at worst

Clock-based Framing: SONET

- **S**ynchronous **O**ptical **Net**work
  - Transmission over very fast optical links
  - STS-*n*, e.g. STS-1: 51.84 Mbps, STS-768: 36.7 Gbps
- STS-1 frames based on fixed sized frames
  - 9*90 = 810 bytes



Clock-based Framing: SONET

- **S**ynchronous **O**ptical **Net**work
  - Transmission over very fast optical links
  - STS-*n*, e.g. STS-1: 51.84 Mbps, STS-768: 36.7 Gbps
- STS-1 frames based on fixed sized frames
  - 9*90 = 810 bytes
- Physical layer details
  - Bits are encoded using NRZ
  - Payload is XORed with a special 127-bit pattern to avoid long sequences of 0 and 1

Outline

- Framing
- Error Checking and Reliability

Dealing with Noise

- The physical world is inherently noisy
  - Interference from electrical cables
  - Cross-talk from radio transmissions, microwave ovens
  - Solar storms
- How to detect bit-errors in transmissions?
- How to recover from errors?

Naïve Error Detection

- Idea: send two copies of each frame
  - if (memcmp(frame1, frame2) != 0) { OH NOES, AN ERROR! }
- Why is this a bad idea?

Naïve Error Detection

- Idea: send two copies of each frame
  - if (memcmp(frame1, frame2) != 0) { OH NOES, AN ERROR! }
- Why is this a bad idea?
  - Extremely high overhead
  - Poor protection against errors
    - Twice the data means twice the chance for bit errors

Parity Bits

- Idea: add extra bits to keep the number of 1s even
    - Example: 7-bit ASCII characters + 1 parity bit

0101001 1101001 1011110 0001110 0110100

Parity Bits

- Idea: add extra bits to keep the number of 1s even
  - Example: 7-bit ASCII characters + 1 parity bit

0101001 1 1101001 1011110 0001110 0110100

Parity Bits

- Idea: add extra bits to keep the number of 1s even
    - Example: 7-bit ASCII characters + 1 parity bit

0101001 1 1101001 0 1011110 1 0001110 1 0110100 1

Parity Bits

- Idea: add extra bits to keep the number of 1s even
    - Example: 7-bit ASCII characters + 1 parity bit

0101001 1 1101001 0 1011110 1 0001110 1 0110100 1
1

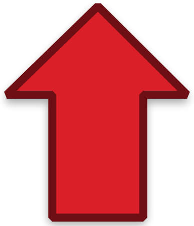
- Detects 1-bit errors and some 2-bit errors

Parity Bits

- Idea: add extra bits to keep the number of 1s even
  - Example: 7-bit ASCII characters + 1 parity bit

0101001 1 1101001 0 1011110 1 0001110 1 0110100 1

1



- Detects 1-bit errors and some 2-bit errors

Parity Bits

- Idea: add extra bits to keep the number of 1s even
  - Example: 7-bit ASCII characters + 1 parity bit

0101001 1 1101001 0 1011110 1 0001110 1 0110100 1

10

- Detects 1-bit errors and some 2-bit errors

Parity Bits

- Idea: add extra bits to keep the number of 1s even
  - Example: 7-bit ASCII characters + 1 parity bit

0101001 1 1101001 0 1011110 1 0001110 1 0110100 1
10

- Detects 1-bit errors and some 2-bit errors

Parity Bits

- Idea: add extra bits to keep the number of 1s even
  - Example: 7-bit ASCII characters + 1 parity bit

0101001 1 1101001 0 1011110 1 0001110 1 0110100 1 10

- Detects 1-bit errors and some 2-bit errors
- Not reliable against bursty errors

Two Dimensional Parity

0101001
1101001
1011110
0001110
0110100
1011111

Two Dimensional Parity

0101001	1
1101001	0
1011110	1
0001110	1
0110100	1
1011111	0

Parity bit for each row

Two Dimensional Parity

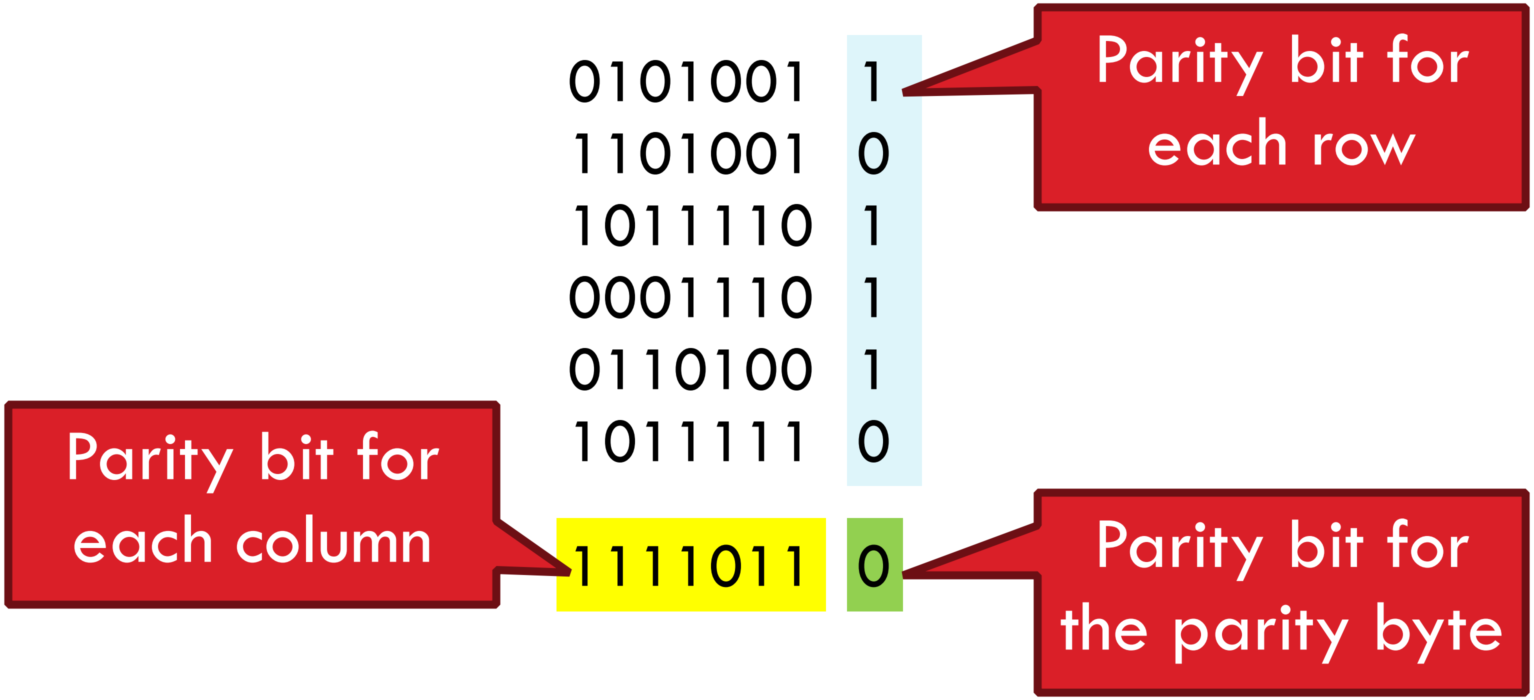
```
0101001  1
1101001  0
1011110  1
0001110  1
0110100  1
1011111  0

1111011
```

Parity bit for each row

Parity bit for each column

Two Dimensional Parity



0101001 1
1101001 0
1011110 1
0001110 1
0110100 1
1011111 0

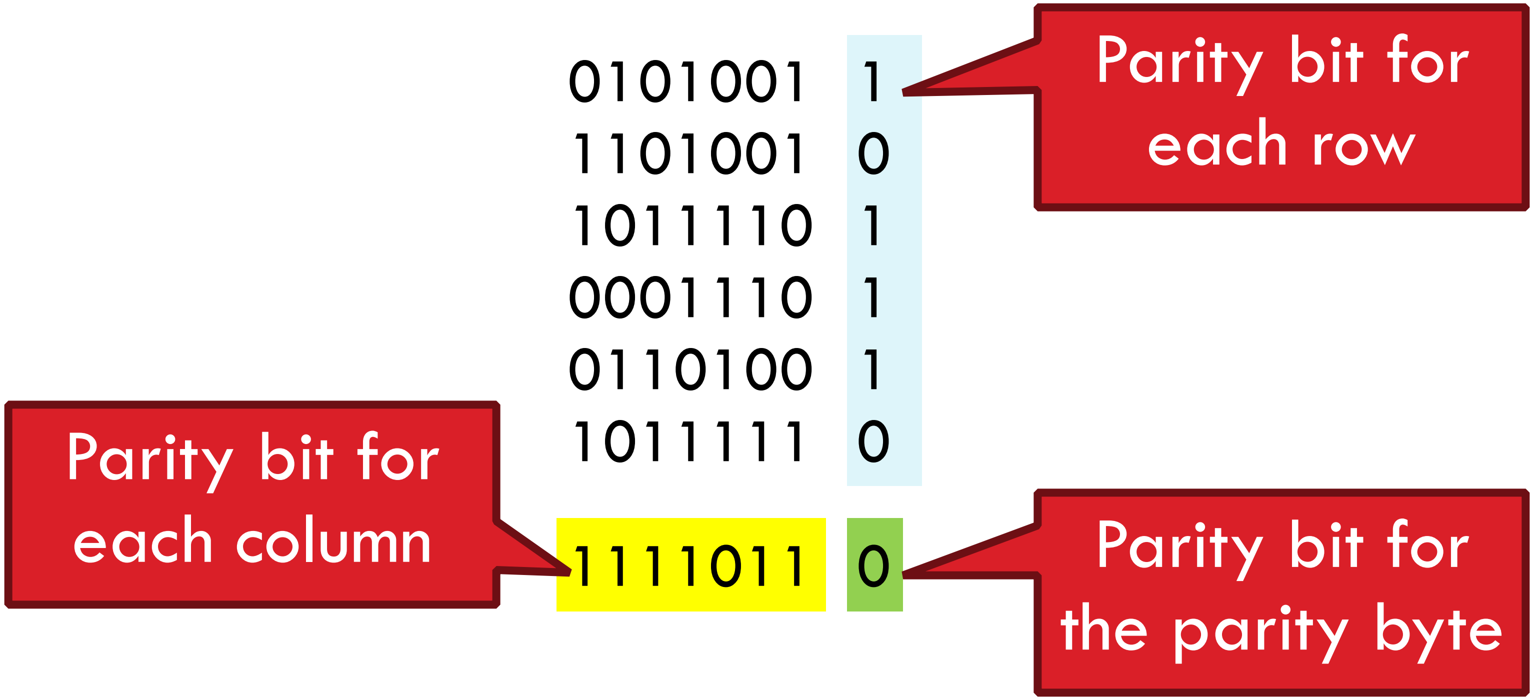
1111011 0

Parity bit for each row

Parity bit for each column

Parity bit for the parity byte

Two Dimensional Parity



0101001 1
1101001 0
1011110 1
0001110 1
0110100 1
1011111 0

1111011 0

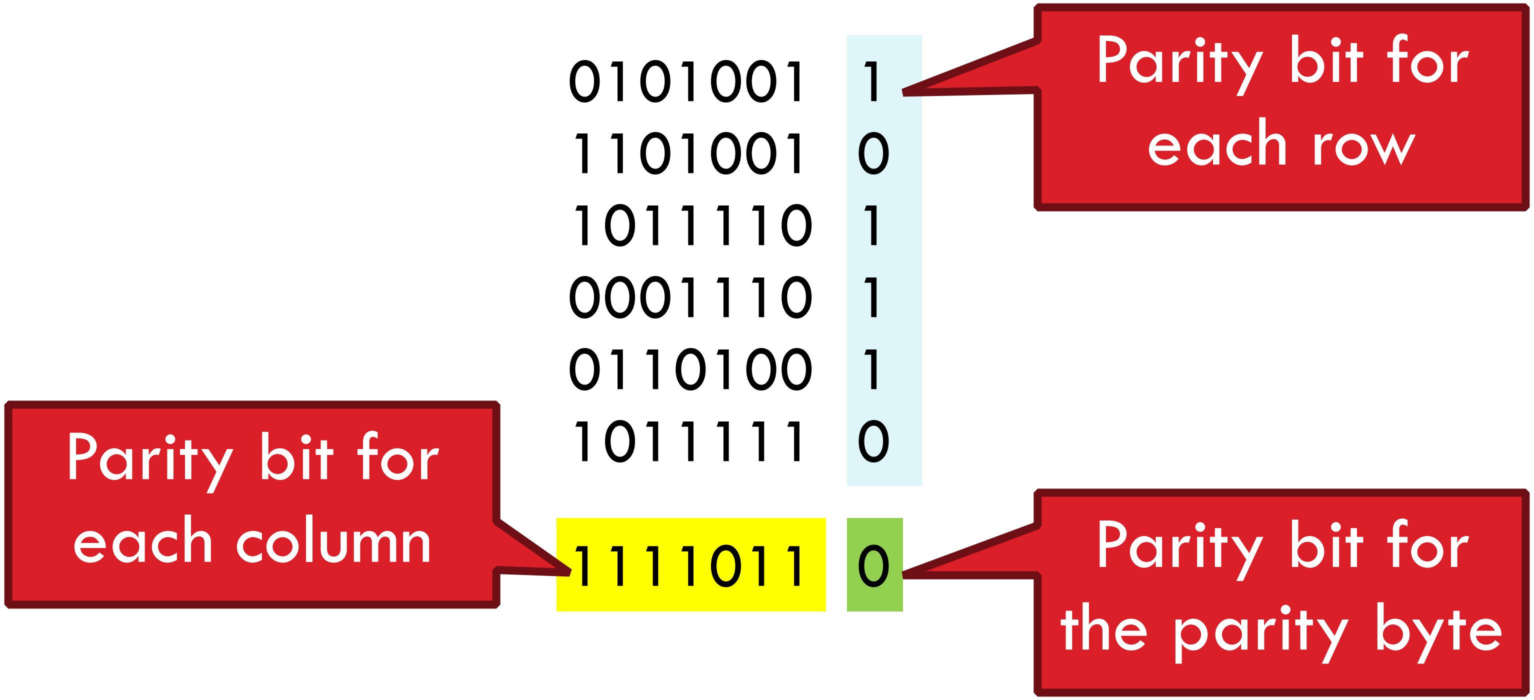
Parity bit for each row

Parity bit for each column

Parity bit for the parity byte

- Can detect all 1-, 2-, and 3-bit errors, some 4-bit errors

Two Dimensional Parity



0101001 1
1101001 0
1011110 1
0001110 1
0110100 1
1011111 0

1111011 0

Parity bit for each row

Parity bit for each column

Parity bit for the parity byte

- Can detect all 1-, 2-, and 3-bit errors, some 4-bit errors
- 14% overhead

Two Dimensional Parity Examples

0101001	1
1101001	0
1011110	1
0011110	1
0110100	1
1011111	0
1111011	0

Two Dimensional Parity Examples

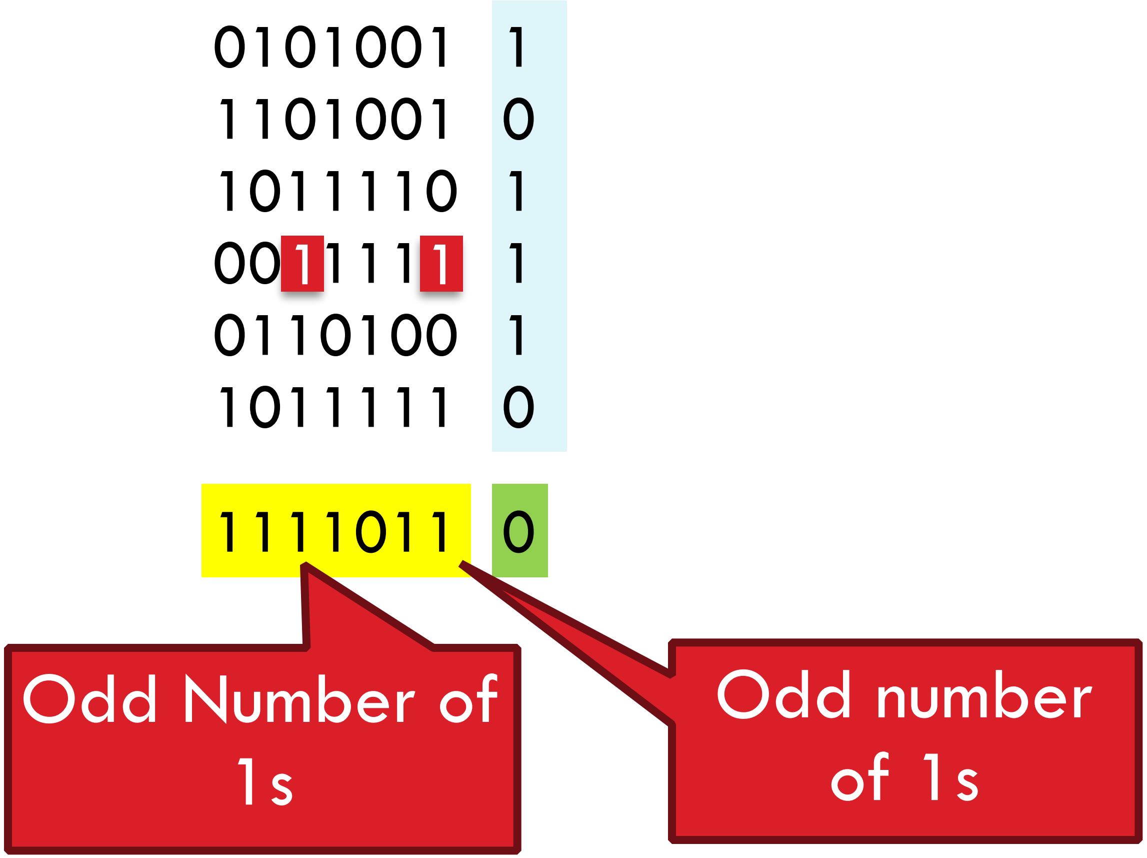
0101001 1
1101001 0
1011110 1
0011110 1
0110100 1
1011111 0

1111011 0

Odd number of 1s

Odd Number of 1s

Two Dimensional Parity Examples



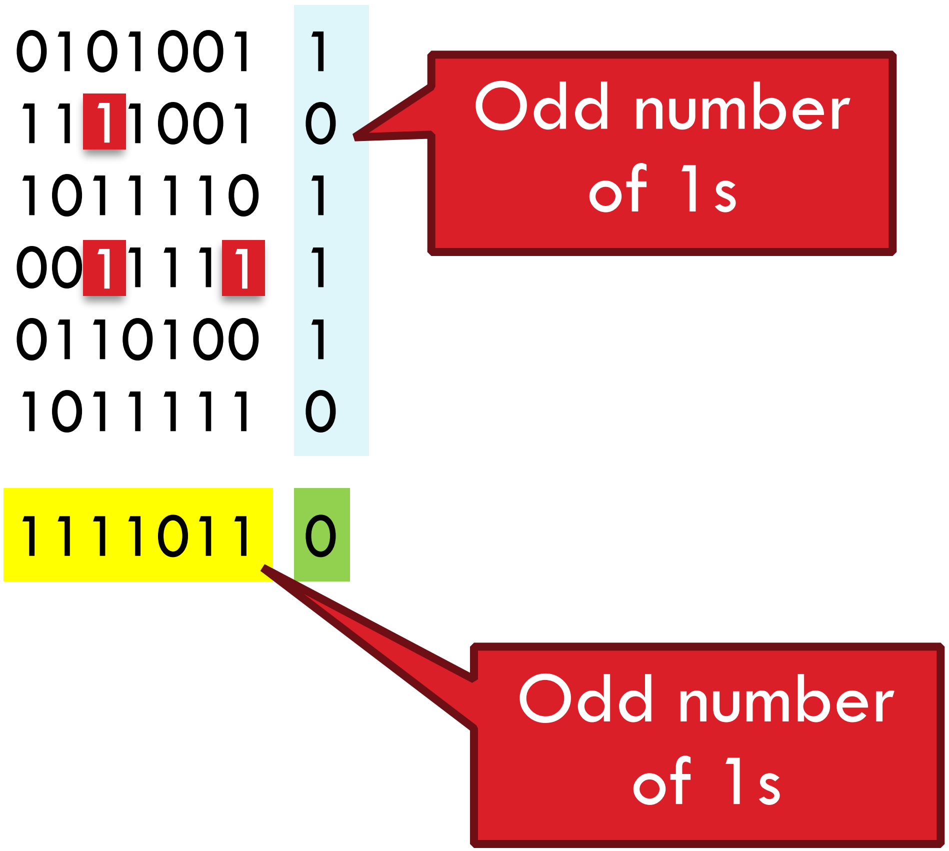
0101001 1
1101001 0
1011110 1
0011111 1
0110100 1
1011111 0

1111011 0

Odd Number of 1s

Odd number of 1s

Two Dimensional Parity Examples



0101001 1
1111001 0
1011110 1
0011111 1
0110100 1
1011111 0

1111011 0

Odd number of 1s

Odd number of 1s

Two Dimensional Parity Examples

0101001 1
1111000 0
1011110 1
0011111 1
0110100 1
1011111 0

1111011 0

Checksums

- Idea:
  - Add up the bytes in the data
  - Include the sum in the frame

- Use ones-complement arithmetic
- Lower overhead than parity: 16 bits per frame
- But, not resilient to errors
  - Why?
- Used in UDP, TCP, and IP

Checksums

- Idea:
  - Add up the bytes in the data
  - Include the sum in the frame

START | Data | Checksum | END

- Use ones-complement arithmetic
- Lower overhead than parity: 16 bits per frame
- But, not resilient to errors
  - Why? 0101001 + 1101001= 10010010
- Used in UDP, TCP, and IP

Checksums

- Idea:
  - Add up the bytes in the data
  - Include the sum in the frame

- Use ones-complement arithmetic
- Lower overhead than parity: 16 bits per frame
- But, not resilient to errors
  - Why? 1101001 + 0101001= 10010010
- Used in UDP, TCP, and IP

Cyclic Redundancy Check (CRC)

- Uses field theory to compute a semi-unique value for a given message

- Much better performance than previous approaches
  - Fixed size overhead per frame (usually 32-bits)
  - Quick to implement in hardware
  - Only 1 in 2^{32} chance of missing an error with 32-bit CRC

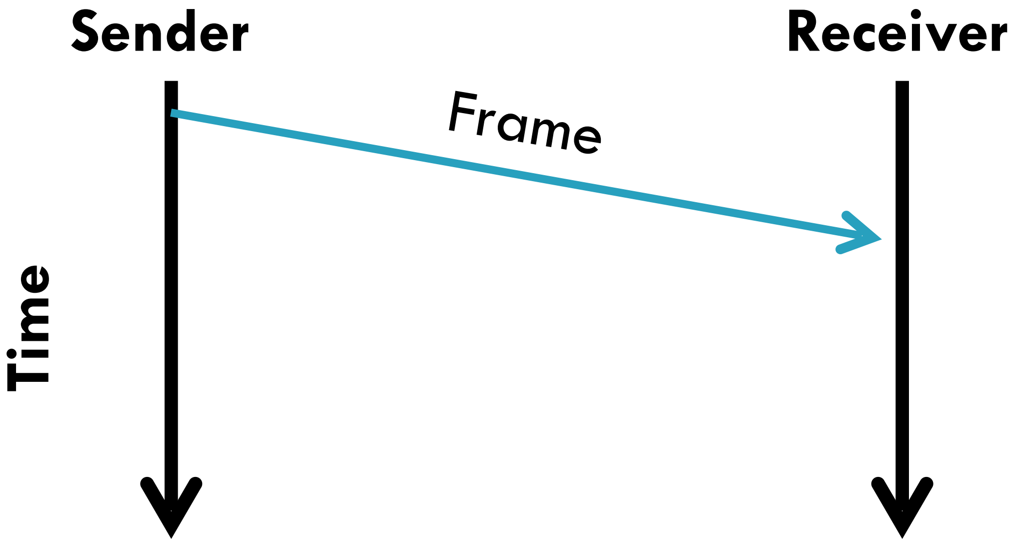
- Details are in the book/on Wikipedia

What About Reliability?

- How does a sender know that a frame was received?
  - What if it has errors?
  - What if it never arrives at all?

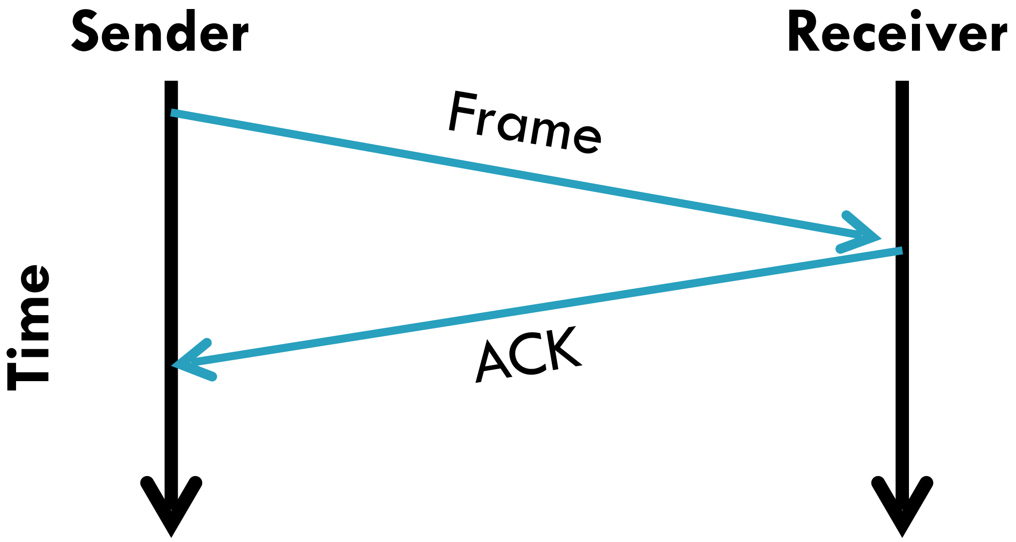
What About Reliability?

- How does a sender know that a frame was received?
  - What if it has errors?
  - What if it never arrives at all?



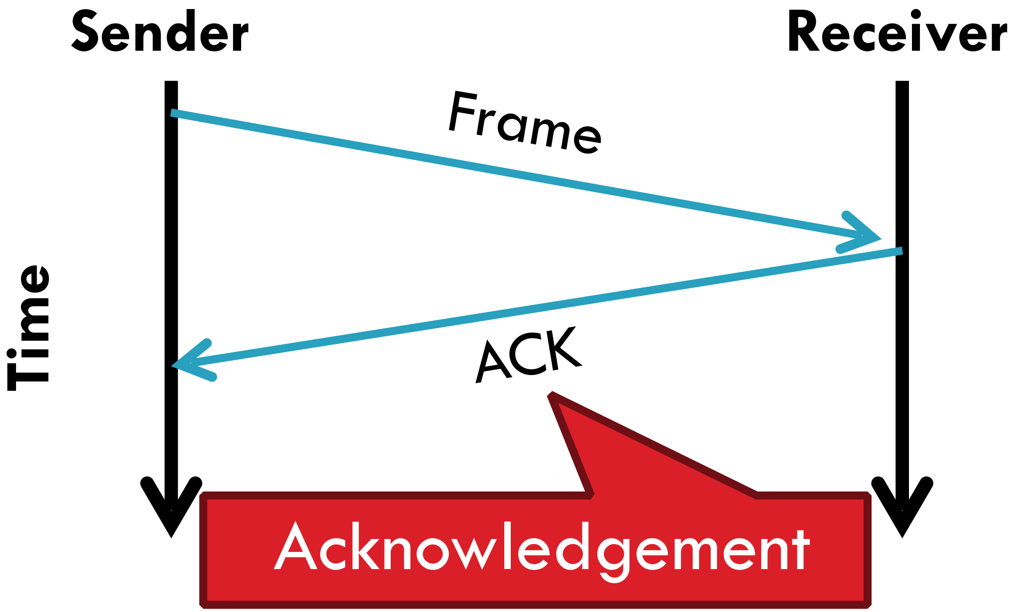
What About Reliability?

- How does a sender know that a frame was received?
  - What if it has errors?
  - What if it never arrives at all?



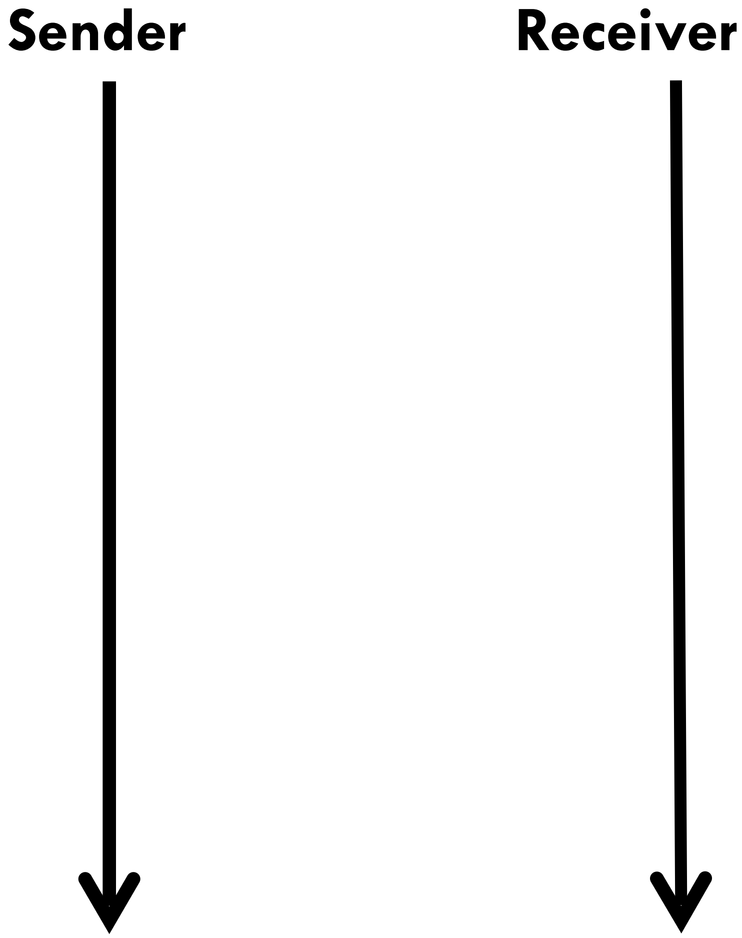
What About Reliability?

- How does a sender know that a frame was received?
  - What if it has errors?
  - What if it never arrives at all?



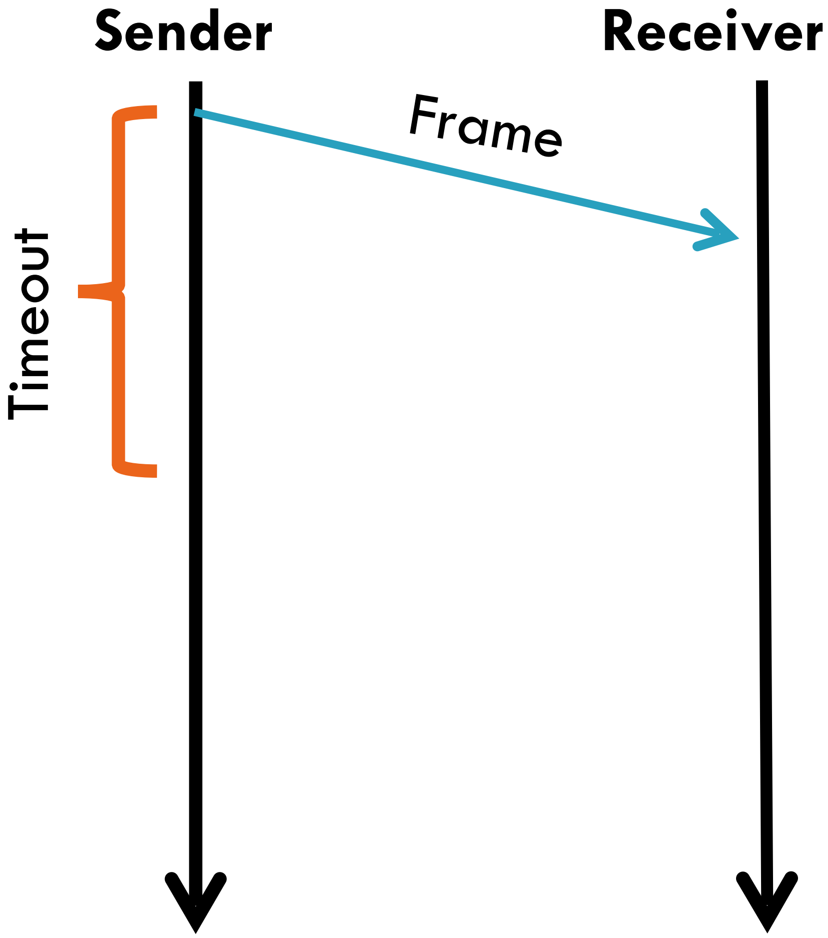
Stop and Wait

- Simplest form of reliability
- Example: Bluetooth



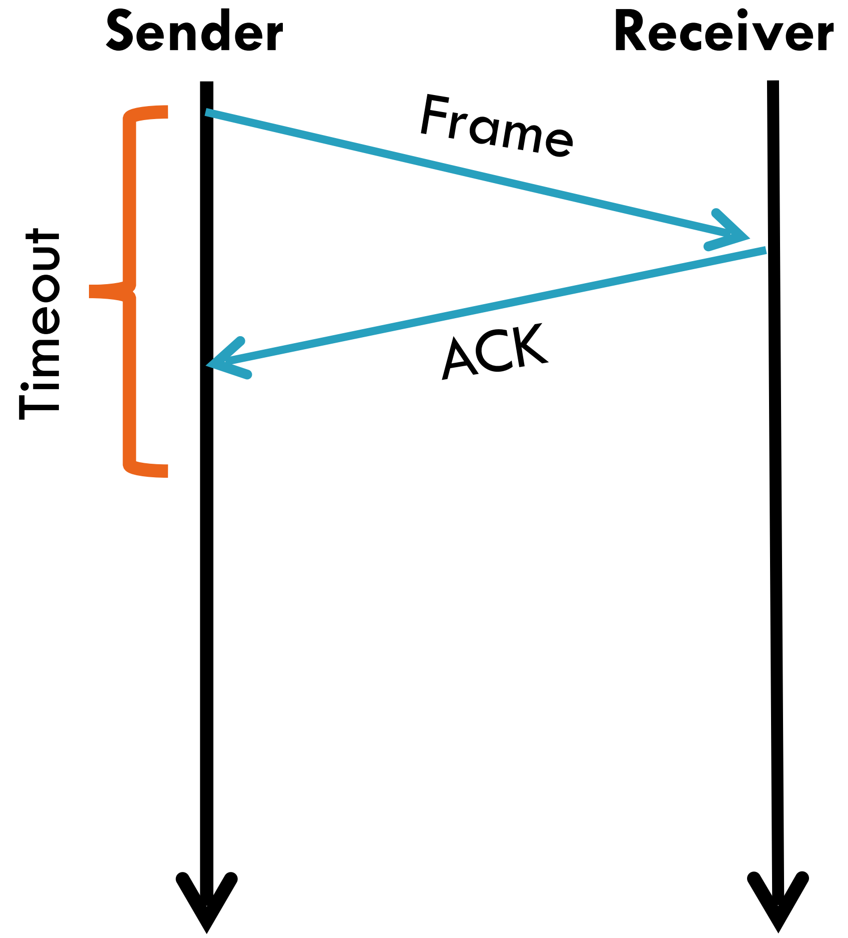
Stop and Wait

- Simplest form of reliability
- Example: Bluetooth



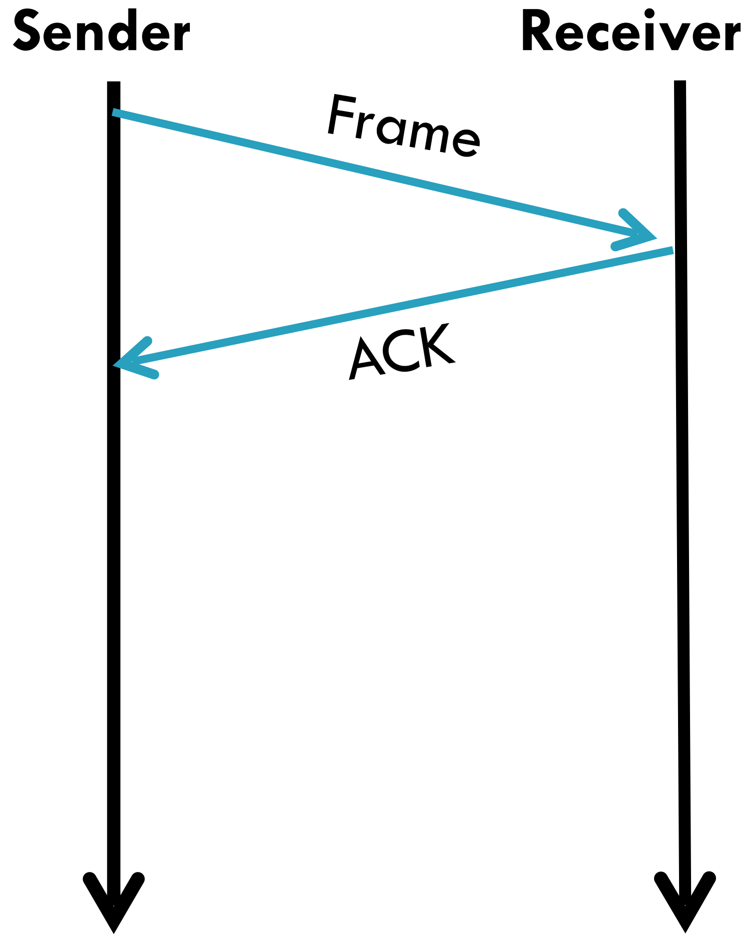
Stop and Wait

- Simplest form of reliability
- Example: Bluetooth



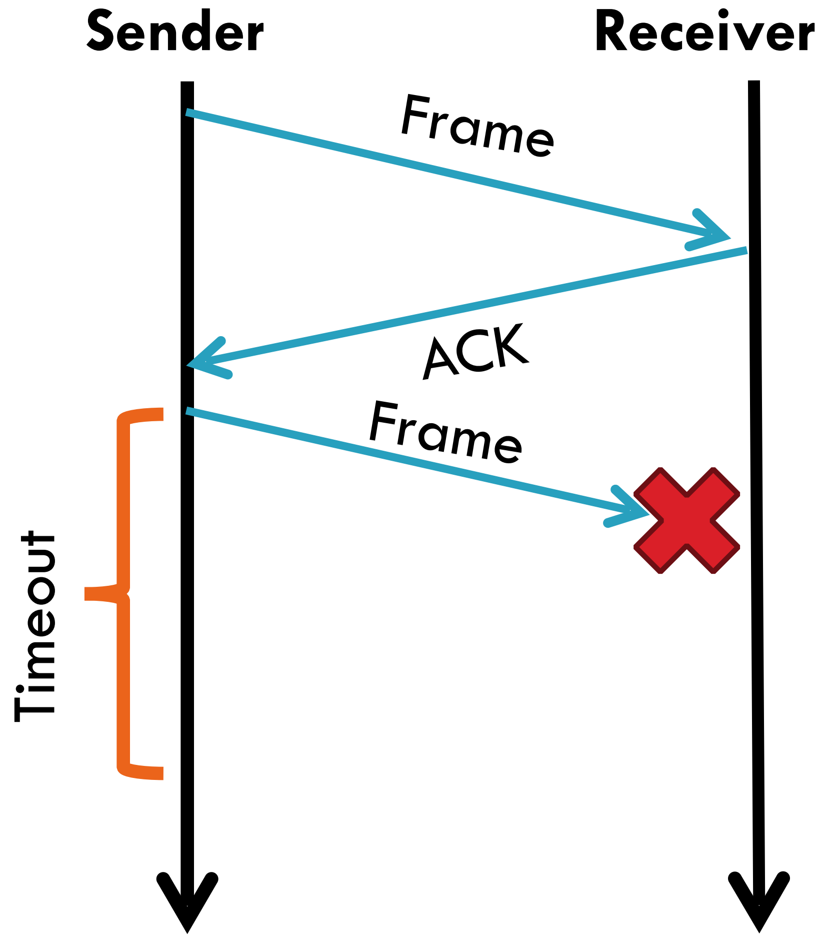
Stop and Wait

- Simplest form of reliability
- Example: Bluetooth



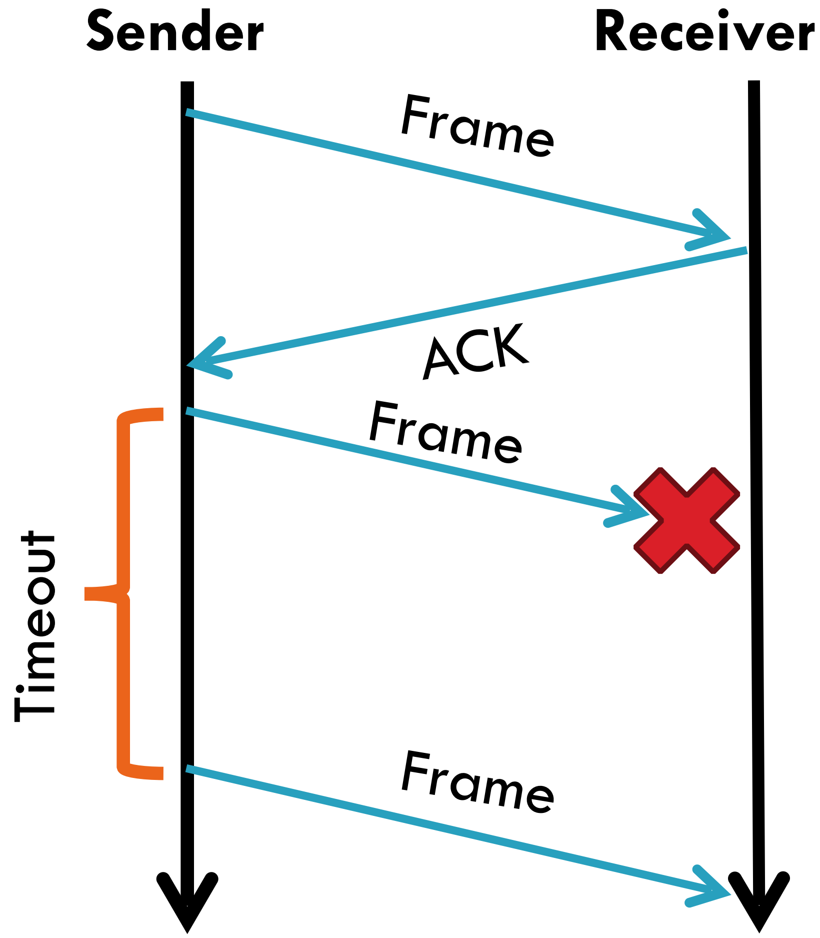
Stop and Wait

- Simplest form of reliability
- Example: Bluetooth



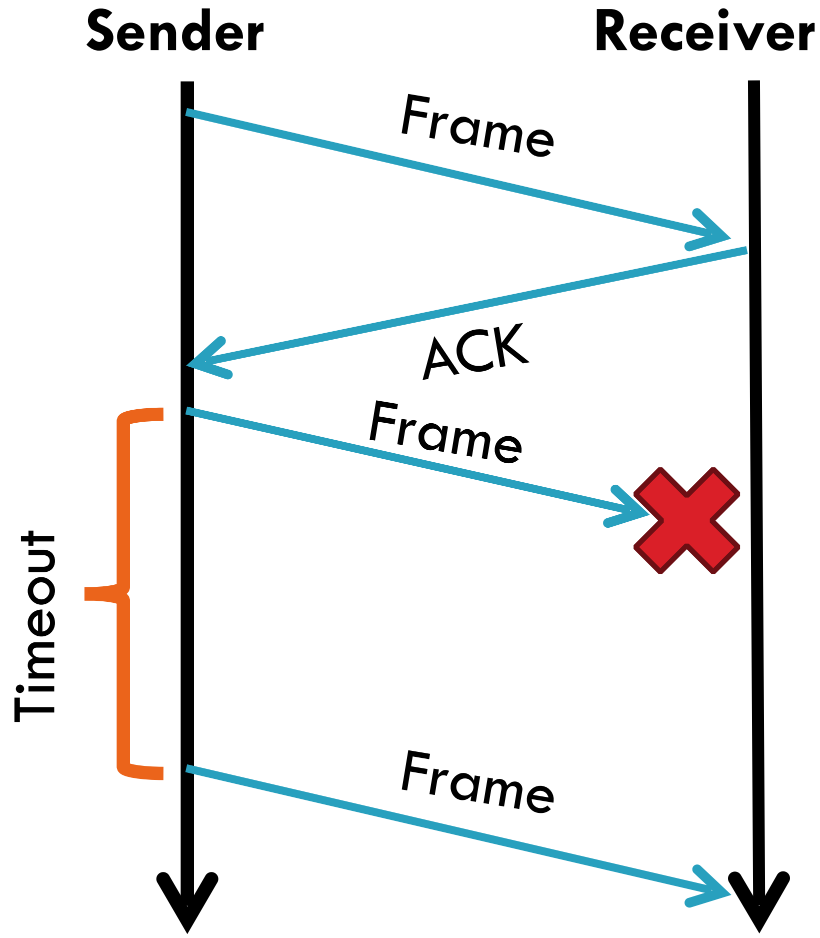
Stop and Wait

- Simplest form of reliability
- Example: Bluetooth



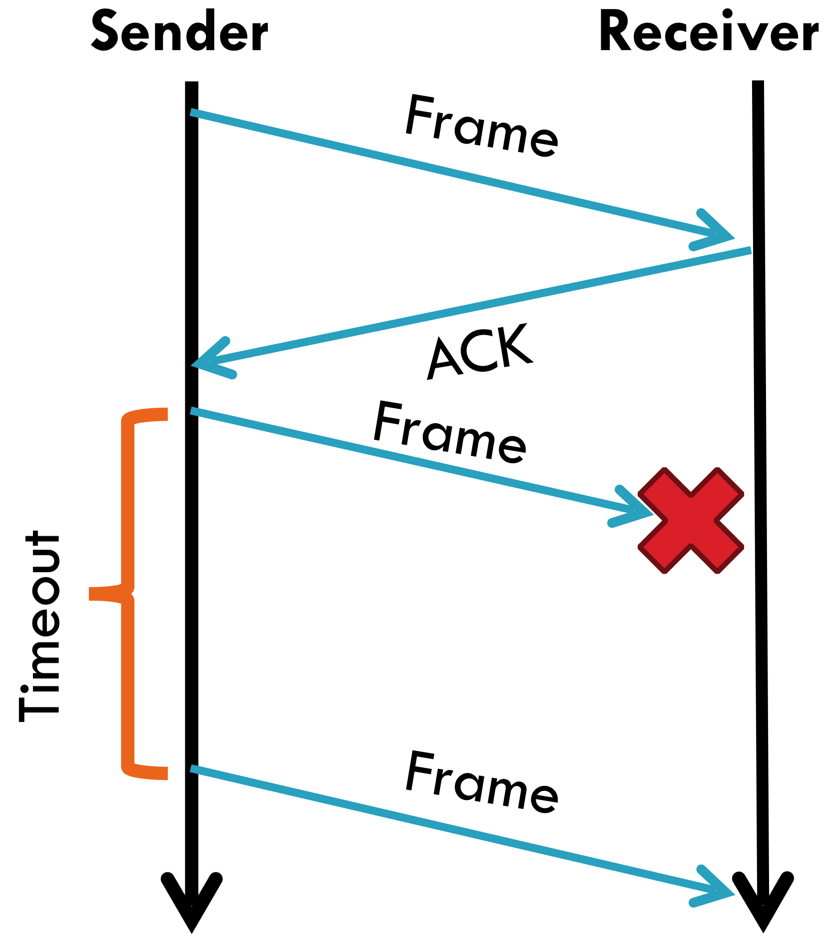
Stop and Wait

- Simplest form of reliability
- Example: Bluetooth
- Problems?



Stop and Wait

- Simplest form of reliability
- Example: Bluetooth
- Problems?
  - Utilization
  - Can only have one frame in flight at any time

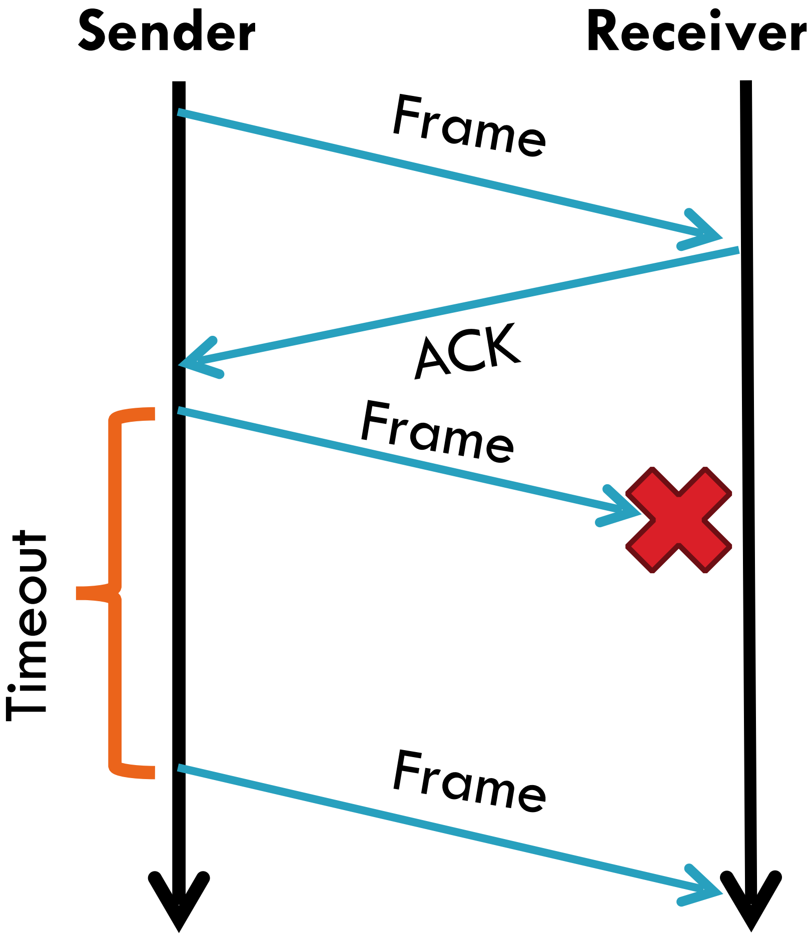


Stop and Wait

- Simplest form of reliability
- Example: Bluetooth
- Problems?
  - Utilization
  - Can only have one frame in flight at any time
- 10Gbps link and 10ms delay
  - Need 100 Mbit to fill the pipe
  - Assume packets are 1500B

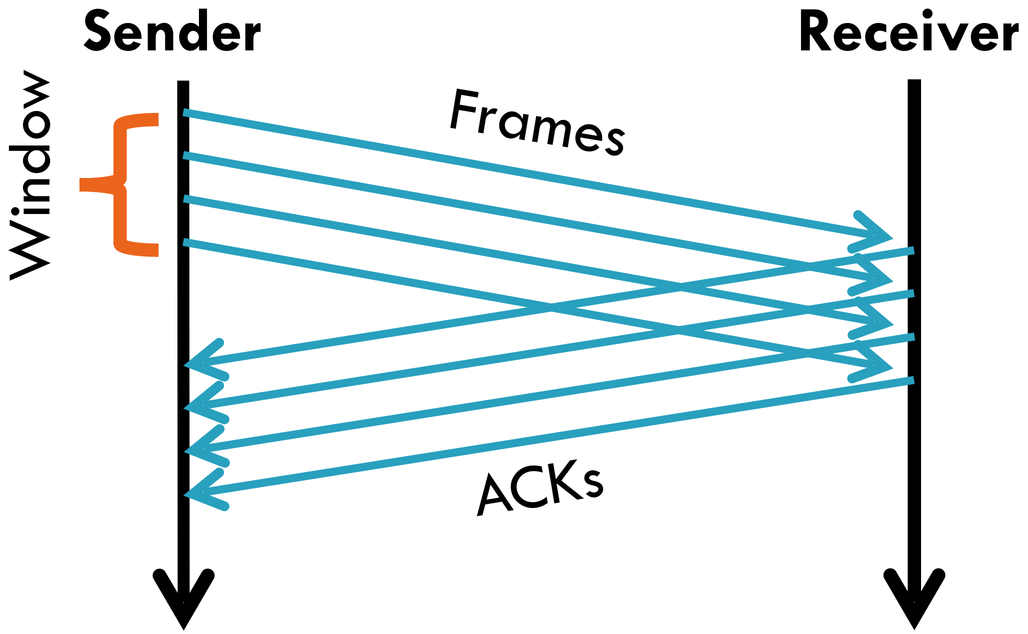
1500B*8bit/(2*10ms) = 600Kbps

Utilization is 0.006%



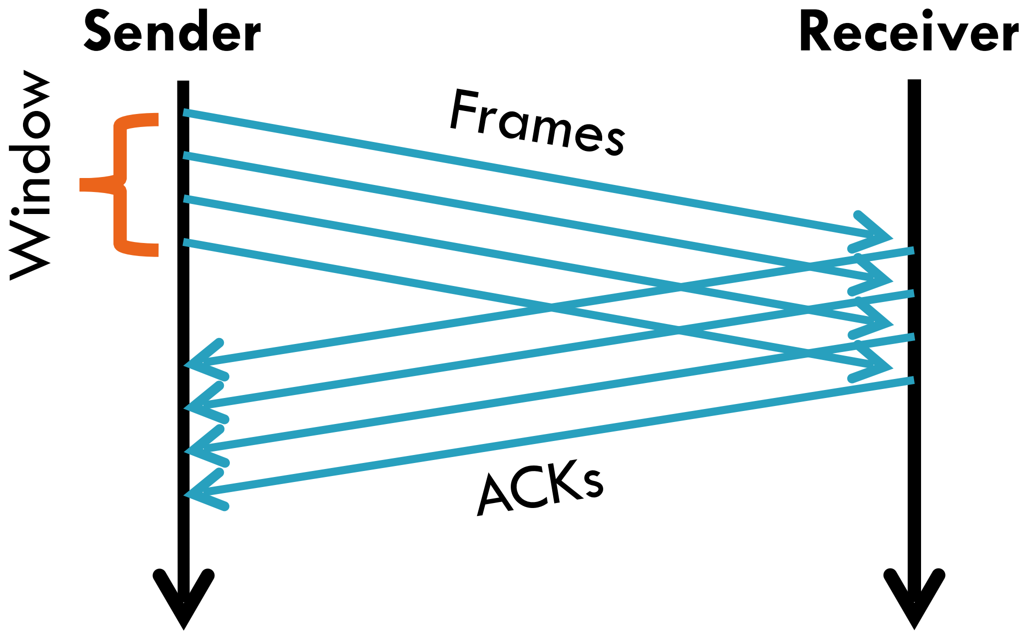
Sliding Window

- Allow multiple outstanding, un-ACKed frames
- Number of un-ACKed frames is called the window



Sliding Window

- Allow multiple outstanding, un-ACKed frames
- Number of un-ACKed frames is called the window



- Made famous by TCP
  - We'll look at this in more detail later

Should We Error Check in the Data Link?

- Recall the End-to-End Argument
- Cons:
  - Error free transmission cannot be guaranteed
  - Not all applications want this functionality
  - Error checking adds CPU and packet size overhead
  - Error recovery requires buffering

Should We Error Check in the Data Link?

- Recall the End-to-End Argument
- Cons:
  - Error free transmission cannot be guaranteed
  - Not all applications want this functionality
  - Error checking adds CPU and packet size overhead
  - Error recovery requires buffering
- Pros:
  - Potentially better performance than app-level error checking

Should We Error Check in the Data Link?

- Recall the End-to-End Argument
- Cons:
  - Error free transmission cannot be guaranteed
  - Not all applications want this functionality
  - Error checking adds CPU and packet size overhead
  - Error recovery requires buffering
- Pros:
  - Potentially better performance than app-level error checking
- Data link error checking in practice
  - Most useful over lossy links
  - Wifi, cellular, satellite