Computer Networks X_400487

Lecture 3

Chapter 3: The Data Link Layer—Part 1

Lecturer: Jesse Donkervliet

VU VRIJE UNIVERSITEIT AMSTERDAM

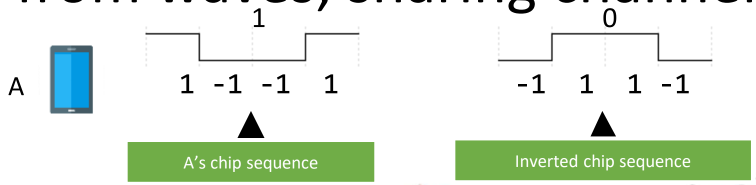
Recap of the Physical Layer

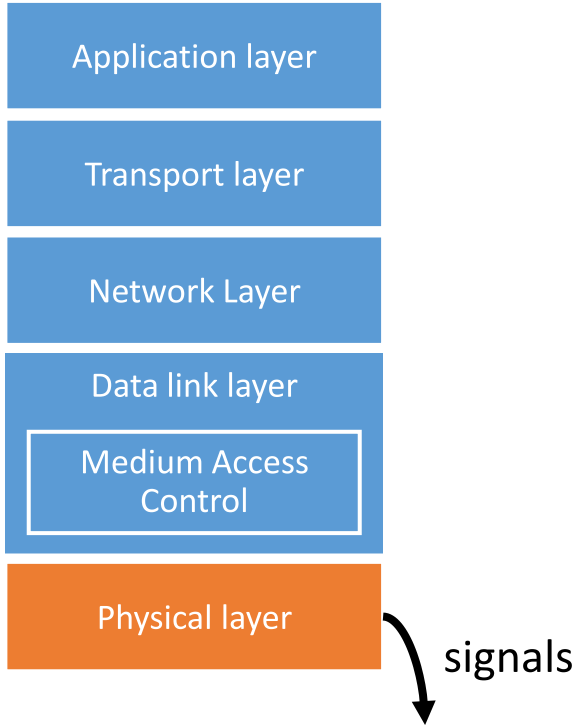
Responsible for transferring *bits* over a *wire-like* medium.

Maximum data rate determined by *bandwidth* and *signal-to-noise ratio.*

$$R = B \times \log_2\left(1 + \frac{S}{N}\right)$$

Physical layer responsible for translating bits to and from waves; sharing channel with multiple users



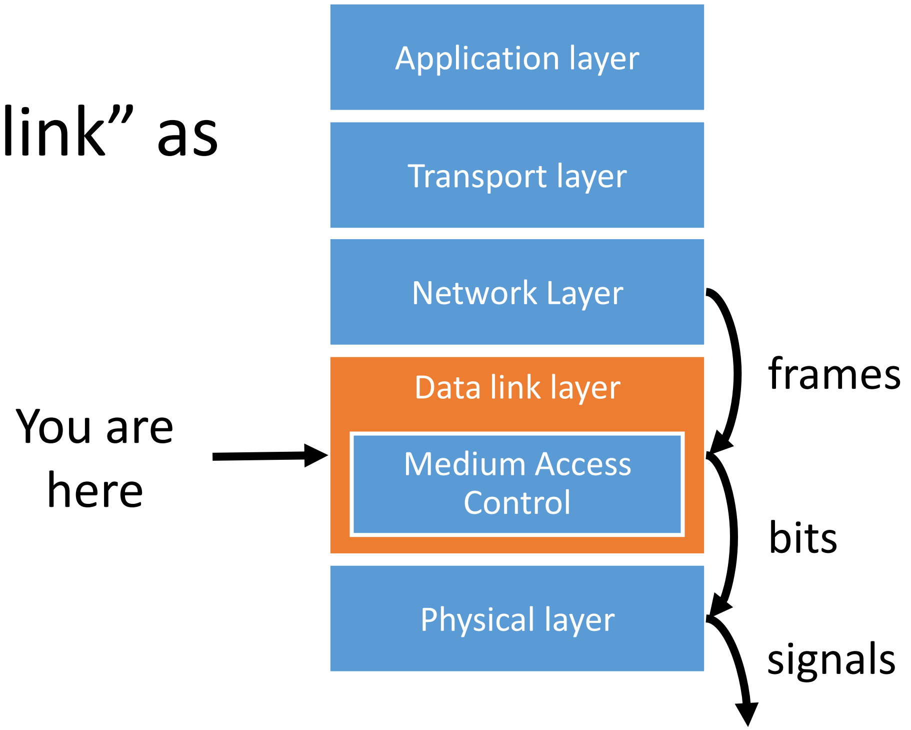


Where are the frames?

The Data Link layer

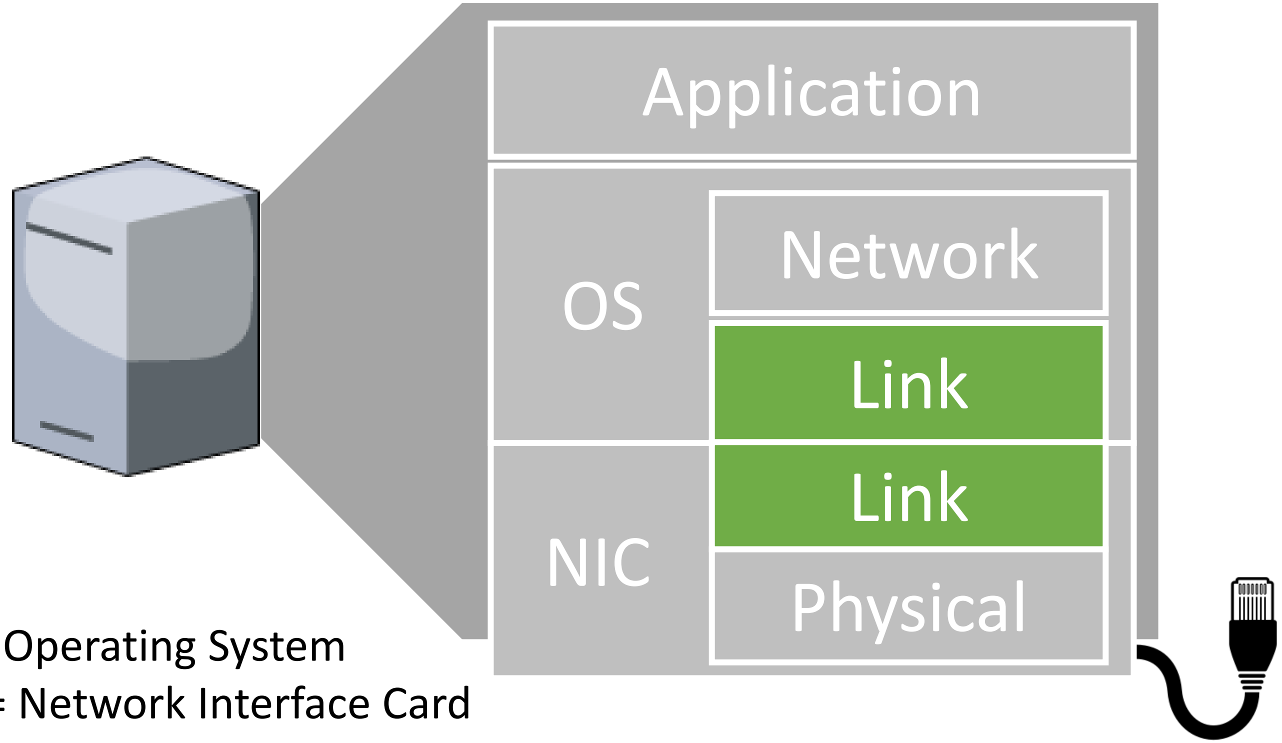
Responsible for transferring *frames* over a single link

1. Groups bits into frames
2. Offers "sending frames over a link" as a *service* to the network layer
3. Handles Q: Why needed? transmission errors
4. Regulates data flow



Link layer environment

Commonly implemented as NICs and OS drivers; network layer (IP) is often OS software.



OS = Operating System
NIC = Network Interface Card

Data Link Layer — Roadmap

Part 1

- **Framing**
- **Flow Control**
- **Guaranteed Delivery**
- **Sliding Window Protocols**

Part 2

- Error detection
- Error correction

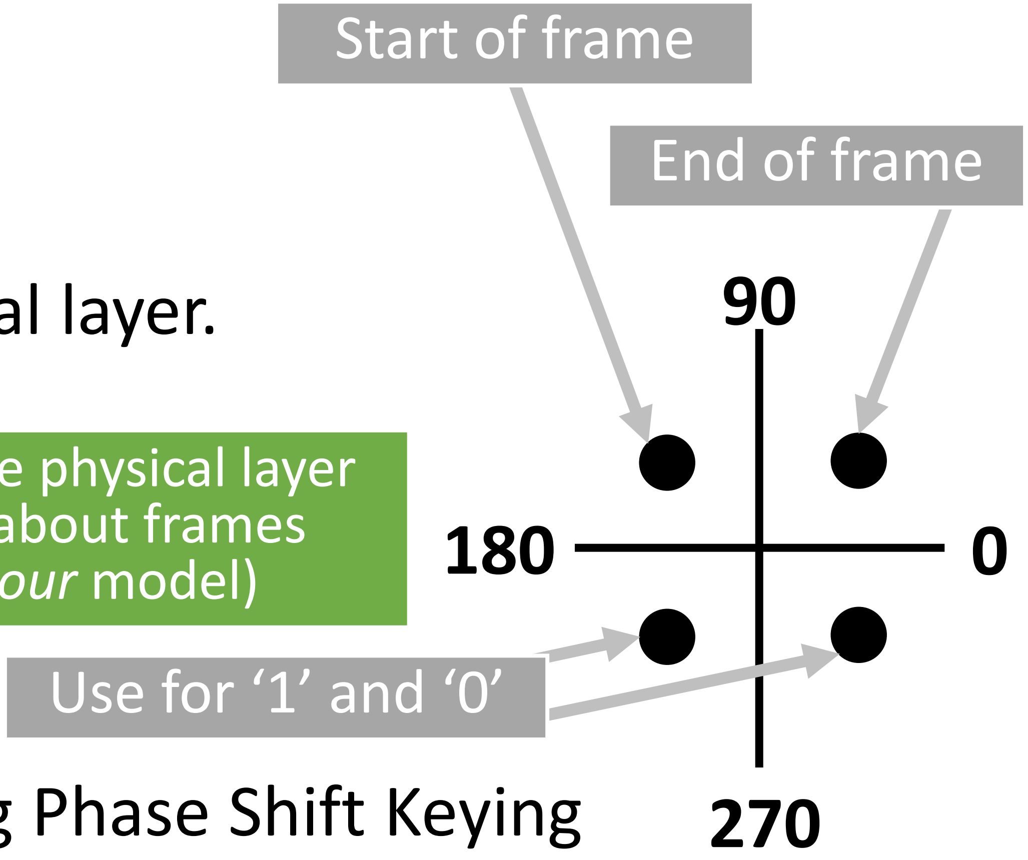
Framing

From Bit Stream to Discrete Units of Information

Framing Methods

1. Byte count.
2. Flag bytes with byte stuffing.
3. Flag bits with bit stuffing.
4. Use special symbols in physical layer.

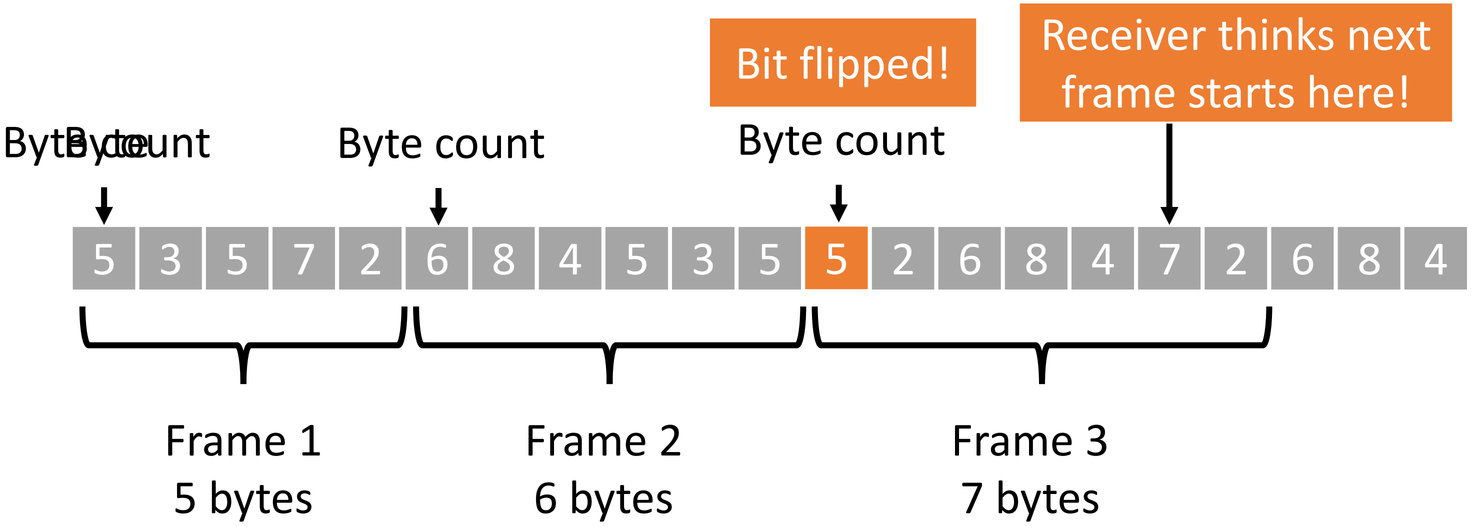
‘Cheating’ because physical layer does not know about frames (according to *our* model)



Example of method 4 using Phase Shift Keying

Framing
Byte count

Q: Advantage?
Disadvantage?

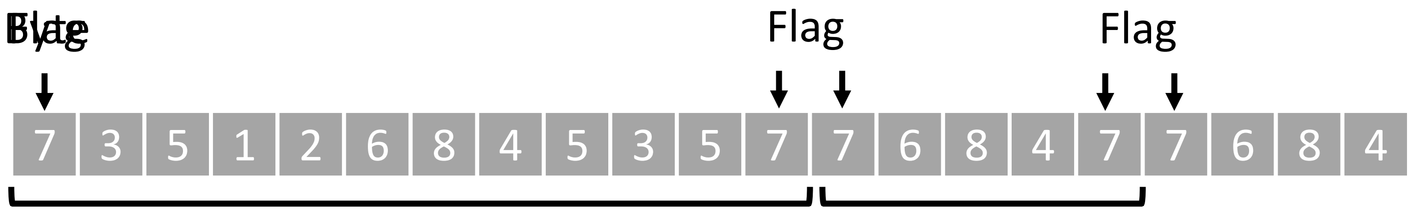


Framing
Byte stuffing

Q: Disadvantage?

Use a 'flag' byte to indicate start and end of frame.

Let's say our flag byte is 00000111_2 (7_{10}).



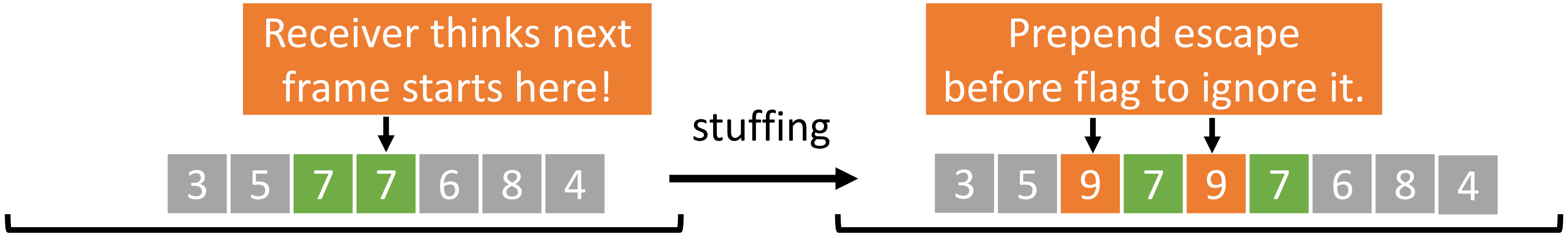
Framing
Byte stuffing

Character	Escape Character
%	%25

What if the data contains a flag byte?

Use an ‘escape’ byte to ignore certain flag bytes.

Let’s say our escape byte is 00001001_2 (9_{10}).



Q: Algorithm on the receiving side?

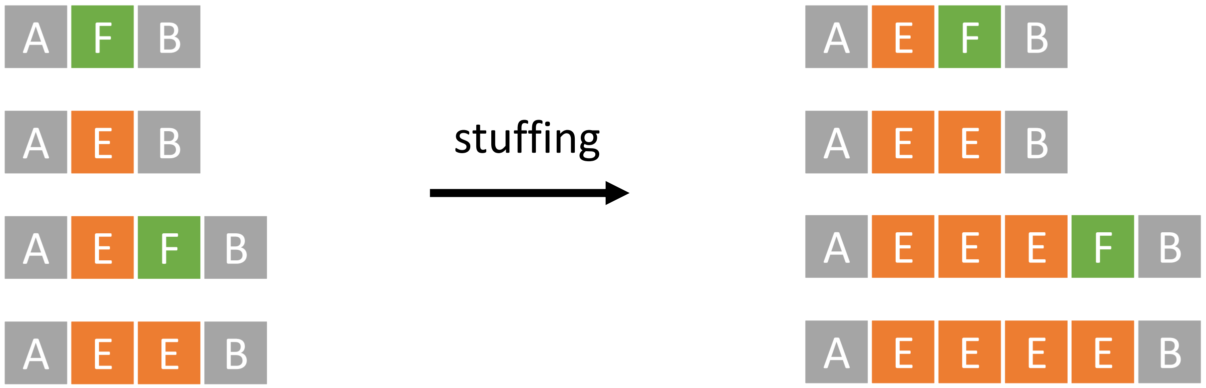
Q: Are we done?

Framing
Byte stuffing

Q: What is the overhead of this approach?

Escape bytes can also occur in data!

Let's use letters for generality.
Flag byte = F, Escape byte = E.



Escape both 'escape' and 'flag' bytes.

Framing
Bit stuffing

Byte stuffing can be space inefficient.

Byte stuffing		**Bit stuffing**
Flag byte	→	Bit pattern
Escape byte	→	Insert single bit

Example:

Bit pattern: `01111110`

Insert bit if pattern in data: `011111010`

Add one extra bit

Bit stuffing example Receiver

Bit pattern: 01111110

1100010100011110111100011100110110000001101110001111101000101101111
0100011010100010010100111110100001110010000001100100101000000101111
0101110000111110100110001110011111001100110000011101111001111001010
0001110111100001110111000110010110011001100011111001001110011010111
0101001110000111110110101000000010001111100110101110011001100001010 0
0101010000001101001011101101010010100100111110001001010010000000100 10
1101111101001100100111000011101111101111000010100110100011111100111
1110000000100100001001011001110101101100000011000110101100000000011
0101101001100001100000111101100100101001100110110110011001101101100
0101000111101011101000111110110010101000001010011011100110101101000
0110101111100011100010000010110101010100101111100101100010010010000
0110101110111011100000010001100100000000111110111001001011101101001 0
0010111110010101001011100011001010011001000100011010100100101010011
0111001111000111101000101001010011111000111110001001111101100010000
1010001101010111001011100010010011110101101001111100001111101101001
101100101111101000101011000000111111000111

Bit stuffing example
Receiver

Q: Are we done?

Bit pattern: 01111110

F1

```
11000101000111101111000111001101100000011011100011111010001011011111
01000110101000100101001111101000011100100000110010010100000001011111
01011100001111101001100011100111110011001100000111011110011110010100
00011101111000011101110001100101100110011000111110010011100110101111
01010011100001111101101010000000100011111001101011100110011000010100
01010100000011010010111011010100101001001111100010010100100000001001 0
1101111101001100100111000011101111101111000010100110100
    00000010010000100101100111010110110000000110001101011000000000011
01011010011000011000001111011001001010011001101101100110011011011100
01010001111010111010001111101100101010000001010011011100110101101000
01101011111000111000100000101101010101001011111001011000100100100000
01101011101110111000000100011001000000001111101110010010111011010010
00101111100101010010111000110010100110010001000110101001001010100 11
01110011110001111010001010010100111110001111100010011111011000100000
10100011010101110010111000100100111101011010011111000011111101101001
10110010111110100010101100000011111000111
```

F2

Bit stuffing example
Receiver

Bit pattern: 01111110

F1

```
110001010001111011110001110011011000000110111000 11111 1000101101111
0100011010100010010100 11111 1000011100100000110010010100000010 11111
 101110000 11111 100110001110011111 0110011000001110111100111100 1010
000111011110000111011100011001011001100110001 1111 01001110011010111
0101001110000 11111 11010100000010001 1111 0110101110011001100001 0100
0101010000001101001011101101010010100100 11111 0010010100100000010010
110 11111 1001100100111000011101 1111 111100001010011 0100
     00000010010000100101100111010110110000000110001101011000000000011
010110100110000110000011110110010010100110011011011001100110 1101100
0101000111101011101000 11111 11001010100000101001101110011010110 1000
011010 11111 001110001000001011010101010010 11111 010110001001001 0000
0110101110111011100000010001100100000000 11111 111001001011101101 0010
0010 11111 0101010010111000110010100110010001000110101001001 01010011
011100111100011110100010100101 00 11111 00 11111 00100 11111 1100010000
10100011010101110010111000100100111101011010011111 00011111 1101001
10110010 11111 1000101011000 00 11111 00111
```

F2

Bit pattern in data

Bit stuffing example
Receiver

Q: What is the overhead of this approach?

Bit pattern: 01111110

F1:

```
11000101000111101111000111001101100000011 … 101111
0100011010100010010100111111 10000111001000 … 011111
 1011100001111 10011000111001111 011001 … 001010
00011101111100001110111000110010110011001 … 010111
0101001110000011111 110101000000100011111 … 010100
01010100000110100101110110101001010010011111 001001010010000010010
11011111 100110010011100001110111111 11110 …
     000000100100001001011001110101101100000 … 000011
```

F2:

```
01011010011000011000001111011001001010011 … 101100
010100011110101110100011111 110010101000000 … 101000
0110101111 00111000100000101101010101001 … 010000
011010111011101110000001000110010000000111 … 010010
00101111 0101010010111000110010100110010 … 010011
0111001111000111101000101001010011111 001 … 010000
1010001101010111001011100010010011110101 1 … 101001
101100101111 100010101100000011111 00111
```

Q: What is the algorithm at the sender?

Sender:

1. Change every '11111' into '111110' (stuffing).
2. Add '01111110' to start and end of each frame.

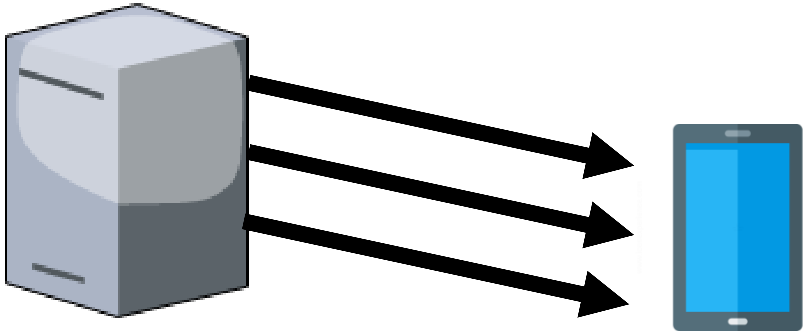
Flow Control

Could you speak more slowly, please?

A *Resource Management* problem

Utopian simplex protocol
The ideal case

```
…
while True:
        packet = from_network_layer()
        frame.payload = packet
        to_physical_layer(frame)
```

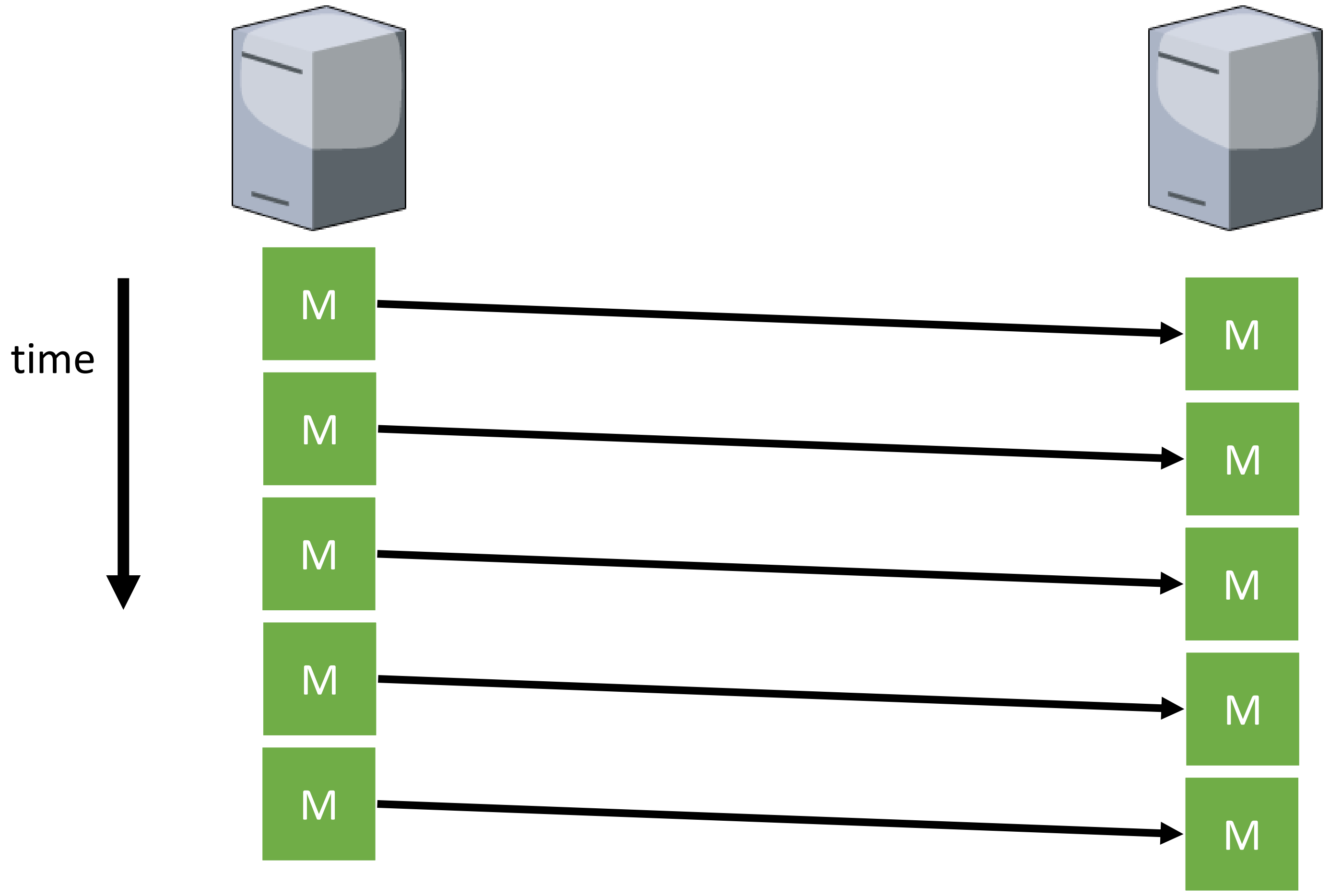


Stop-and-wait
for error-free channel

```
…
while True:
    packet = from_network_layer()
    frame.payload = packet
    to_physical_layer(frame)
    event = wait_for_event()
```

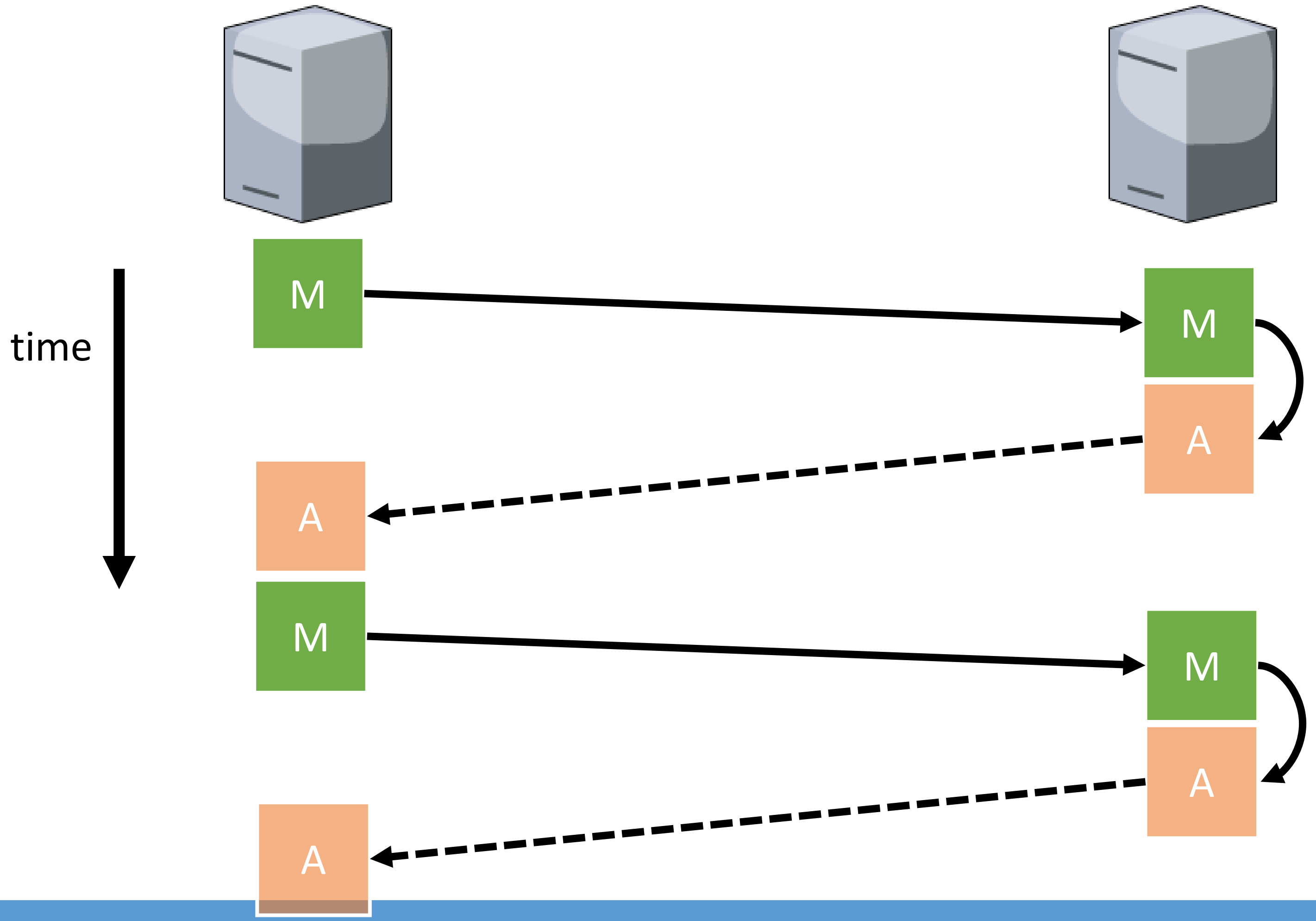
← Wait for signal from receiver.

Example of Utopian Simplex Protocol



Example of Stop-and-Wait Protocol

Q: What if a frame gets lost?



Guaranteed Delivery

Acknowledgments, Sequence Numbers, and Retransmissions

How Can We Know If a Frame Gets Lost?

Ask a different question

Q: How can we know if a frame *arrives*?

Send a message back:
“I got your message!”

Q: When do we want to retransmit data?

Q: What if the acknowledgment gets lost?

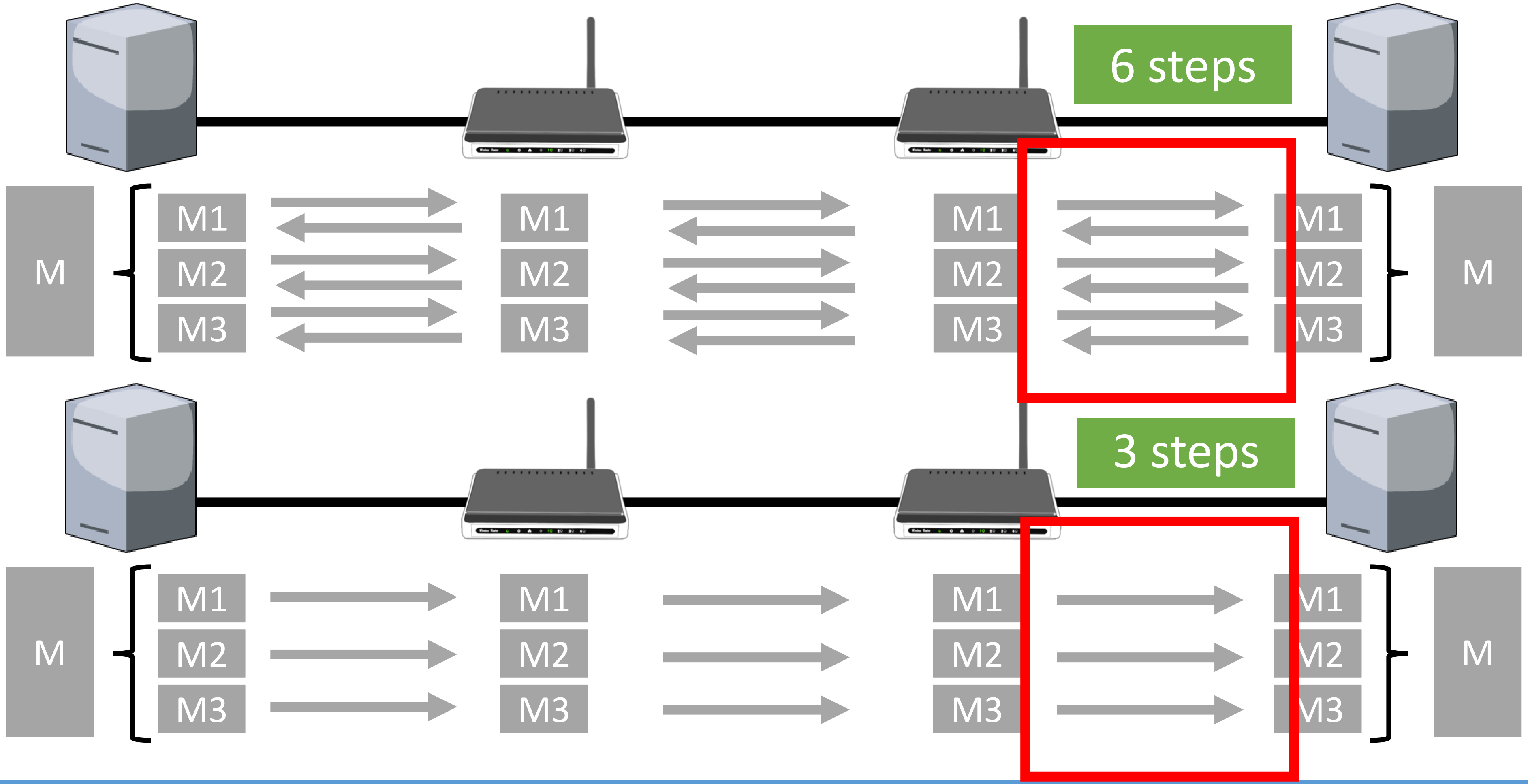
We assume our original message did not arrive

It depends on … the application!

Acknowledgments let the sender know it does ***not*** need to retransmit data.

Many protocols either use or don’t use acknowledgments. Different approach: Support acknowledgments, but let the application decide if it needs to use acknowledgments or not.

To acknowledge,
or not to acknowledge

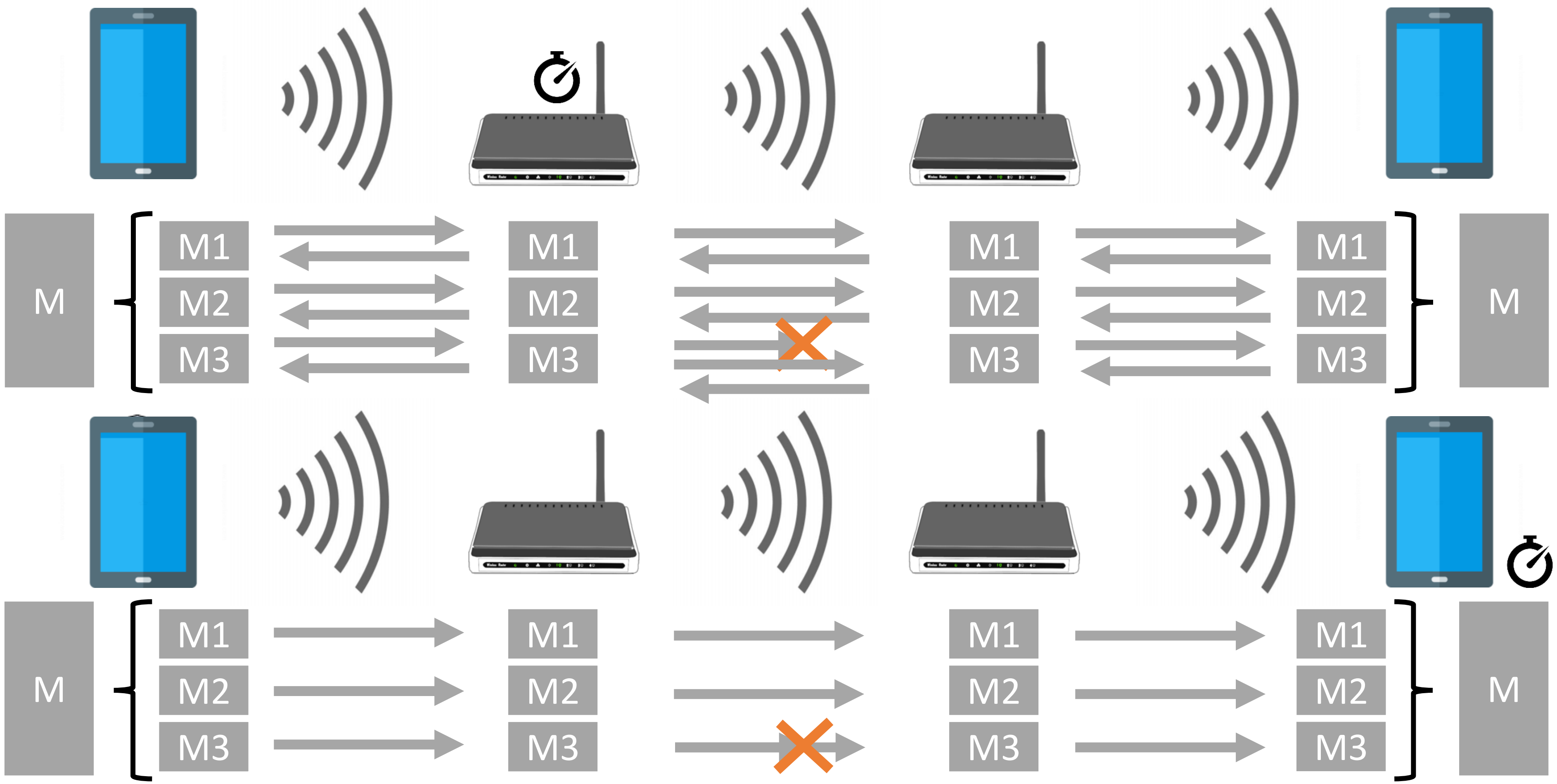


To acknowledge, or not to acknowledge

It depends on …
the physical medium!

It depends on …
the application!

This problem will return later
(in the transport layer)

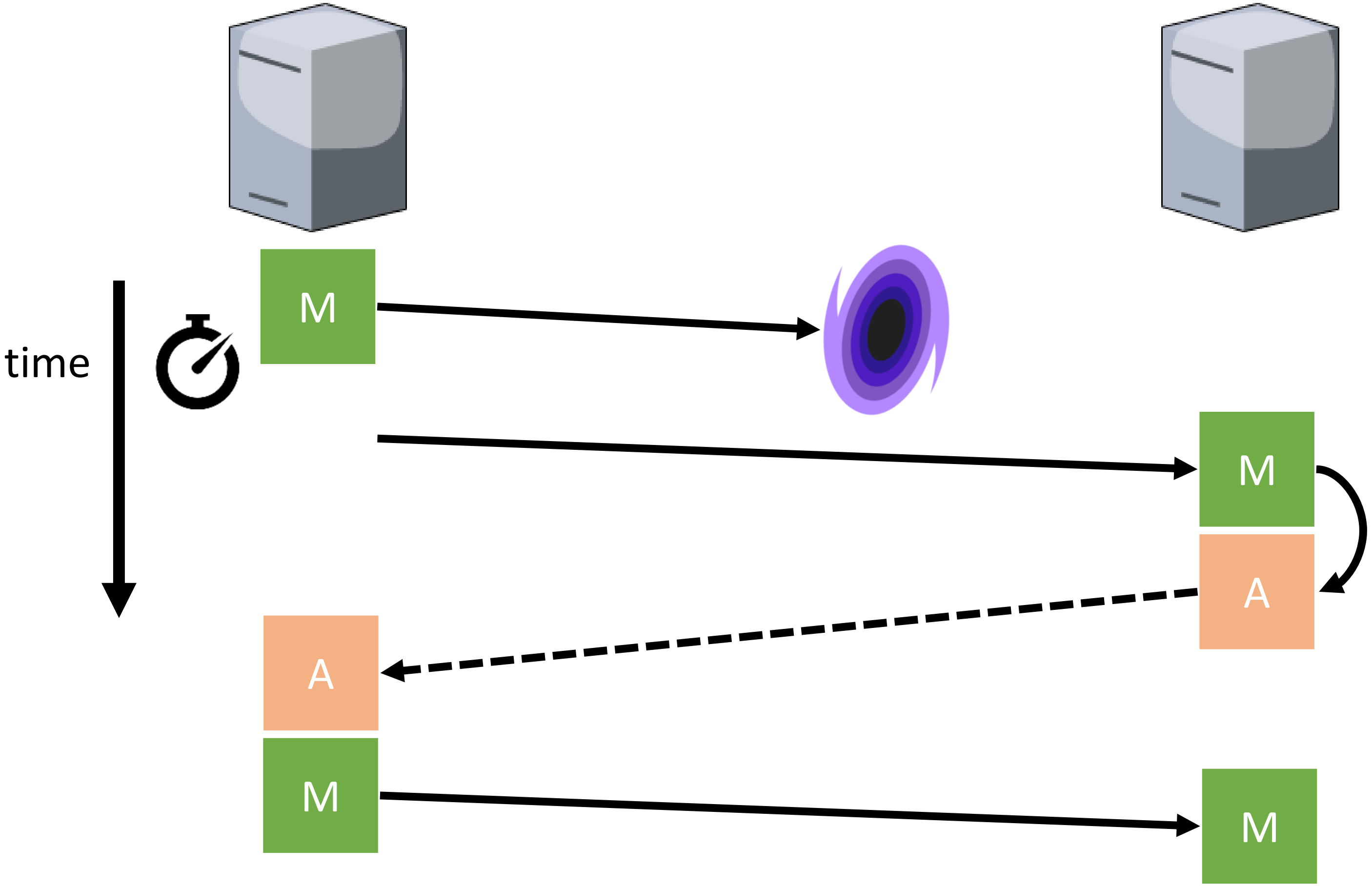


Automatic Repeat ReQuest (ARQ) Guaranteed Delivery over Unreliable Channel

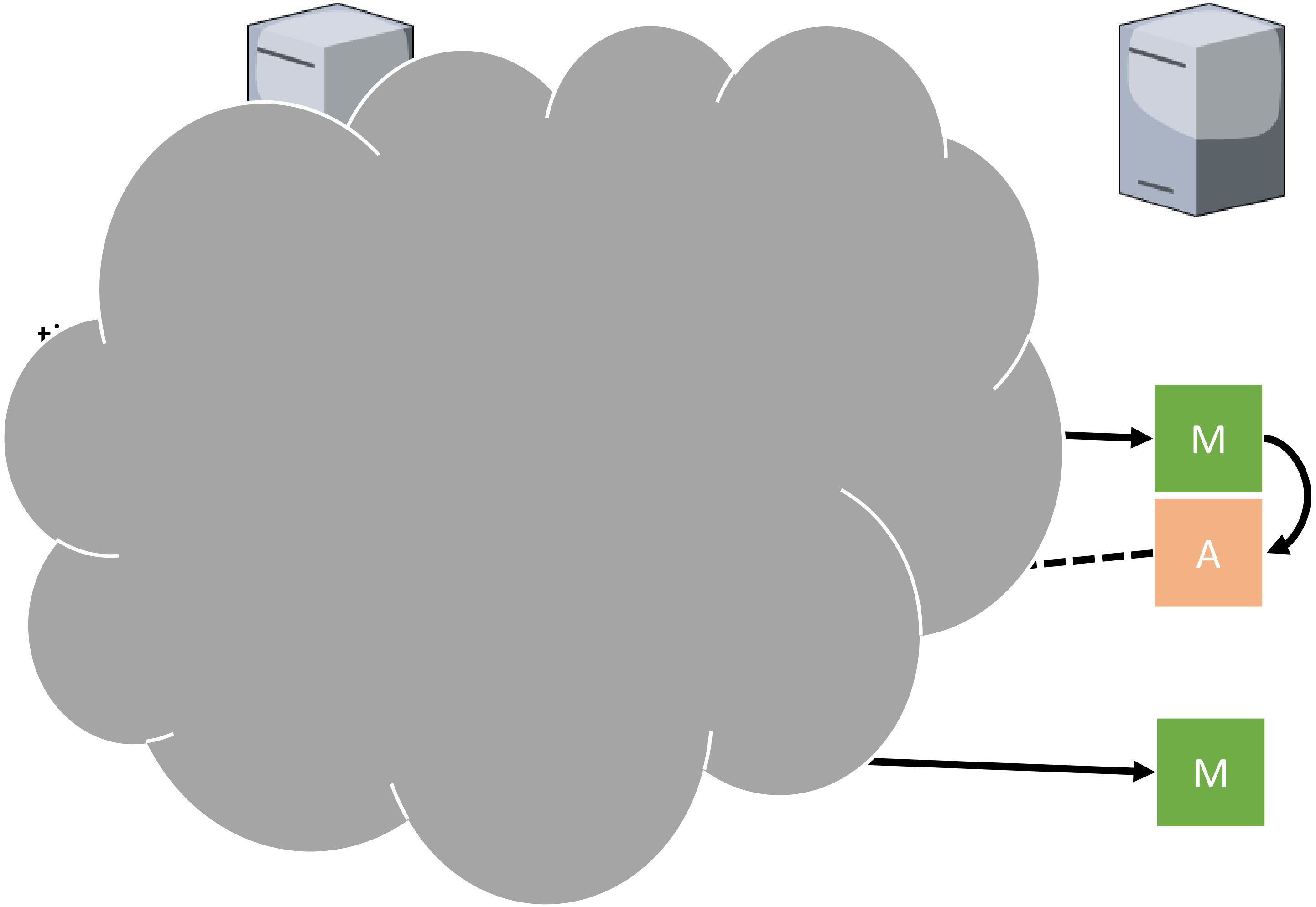
Same as stop-and-wait, except:

1. Keep track of frames using sequence numbers.
2. Wait until previous frame has been accepted.

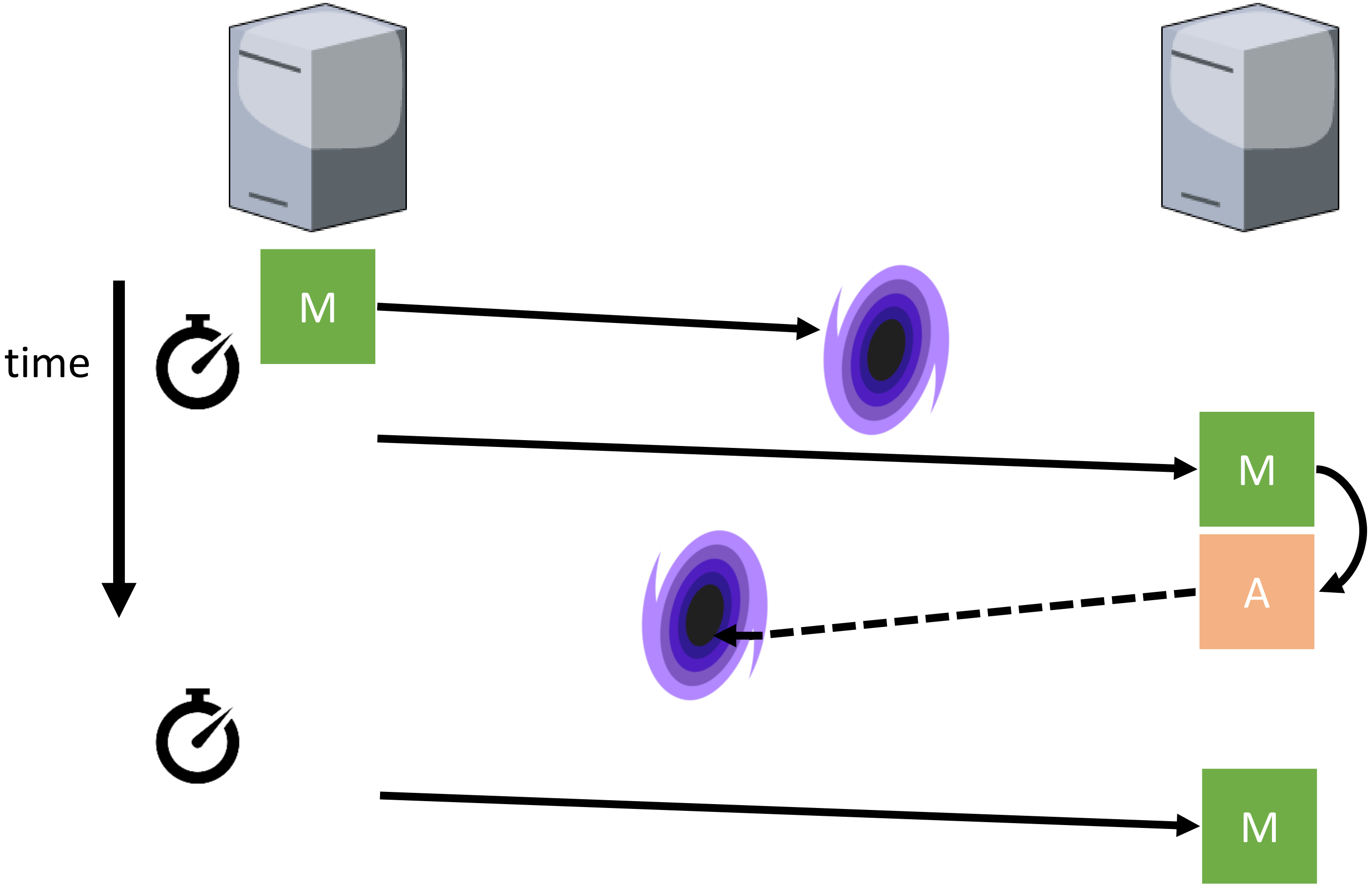
ARQ Example



ARQ Example

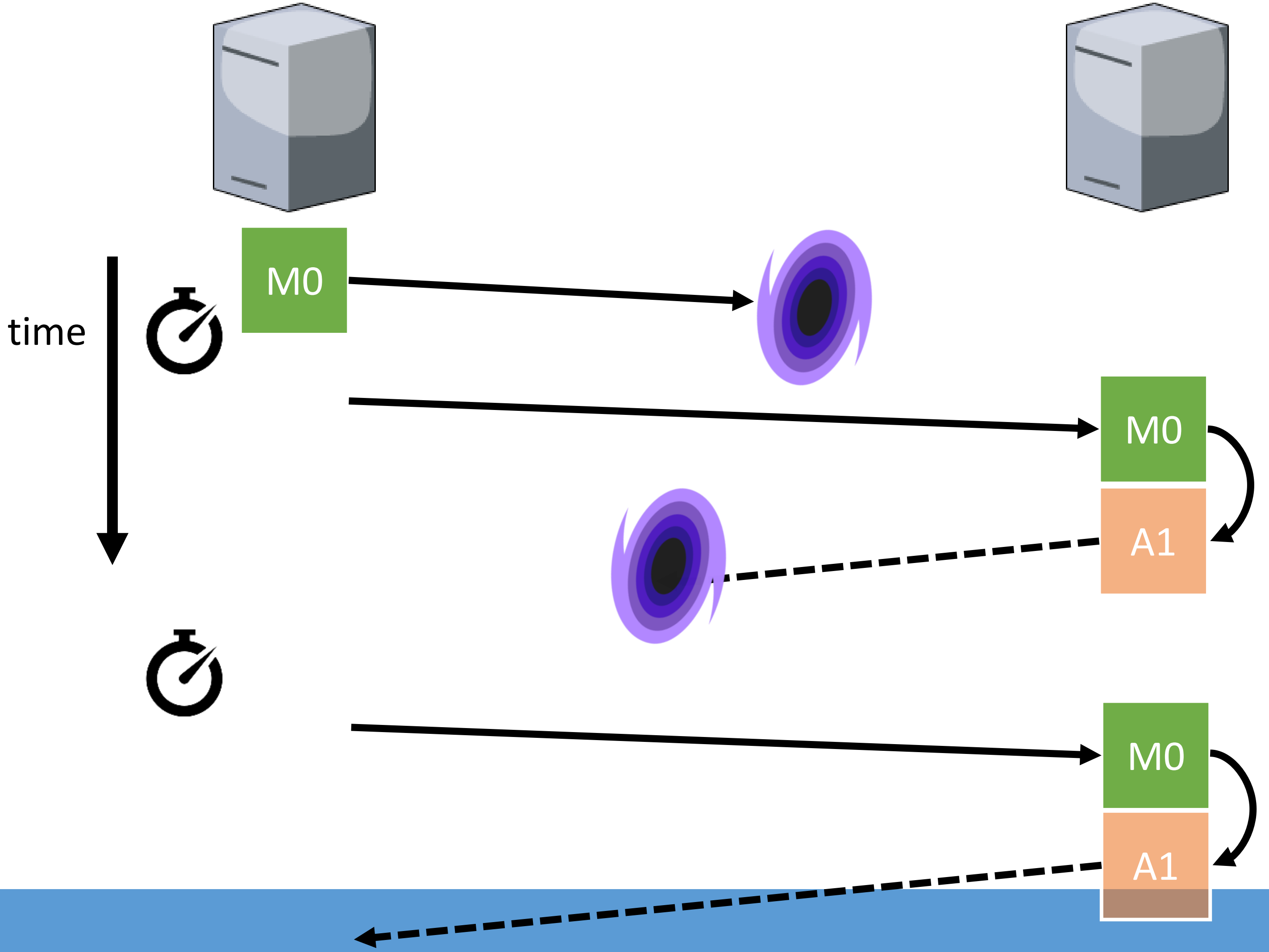


ARQ Example



Sequence numbers needed to differentiate between retransmission and next frame

ARQ Example



Sequence numbers needed to differentiate between retransmission and next frame

Automatic Repeat ReQuest (ARQ) Guaranteed Delivery over Unreliable Channel

ARQ adds error control

Receiver acks frames that are correctly delivered.

Sender sets timer and resends frame if no ack.

Q: How long should we wait?

Q: What can go wrong?

Frames and acks must be identifiable (e.g., with sequence number)

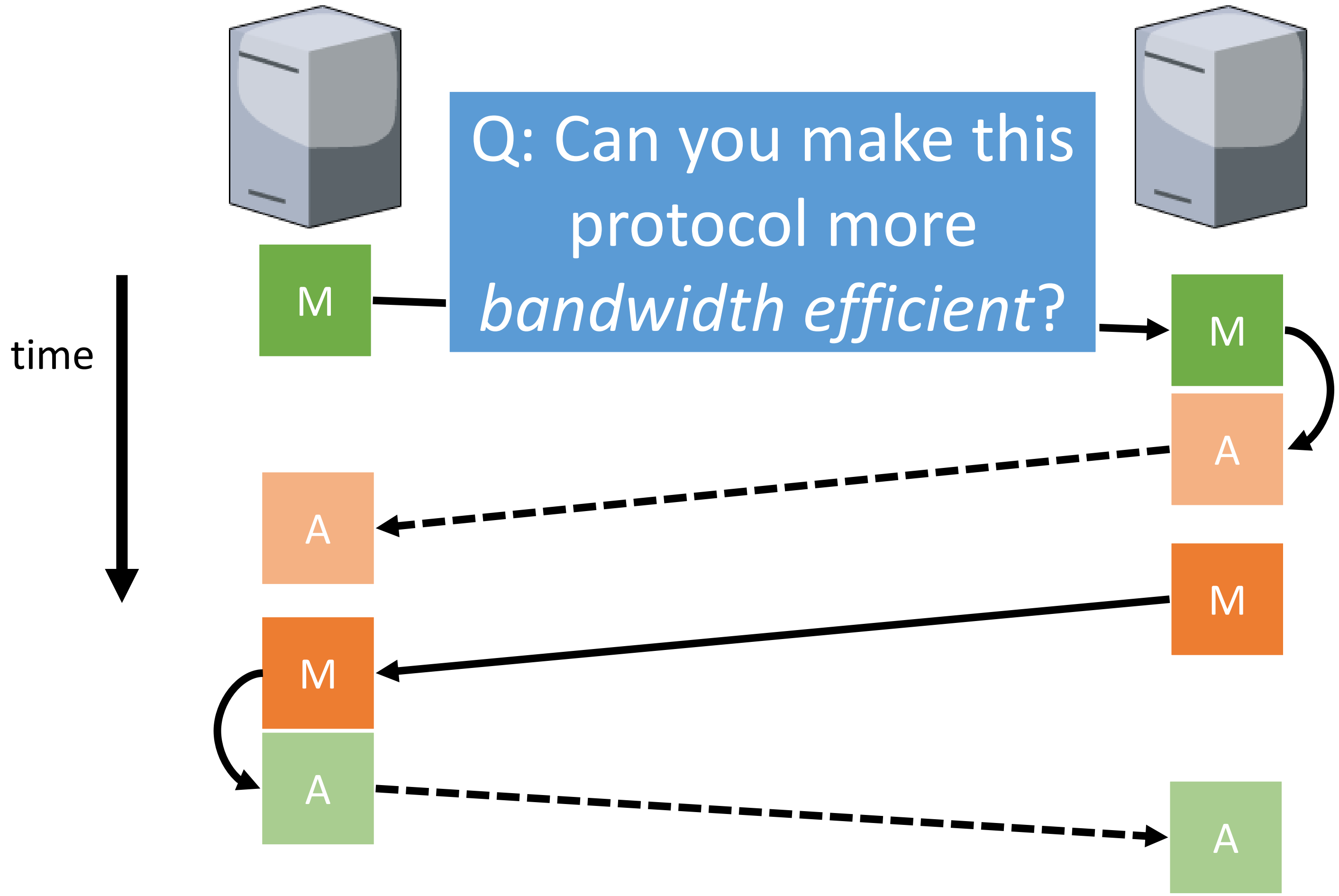
Else receiver cannot tell retransmission (due to lost ack or early timer) from new frame.

For stop-and-wait, 2 numbers (1 bit) are sufficient.

Q: Why sufficient?

Acknowledgements

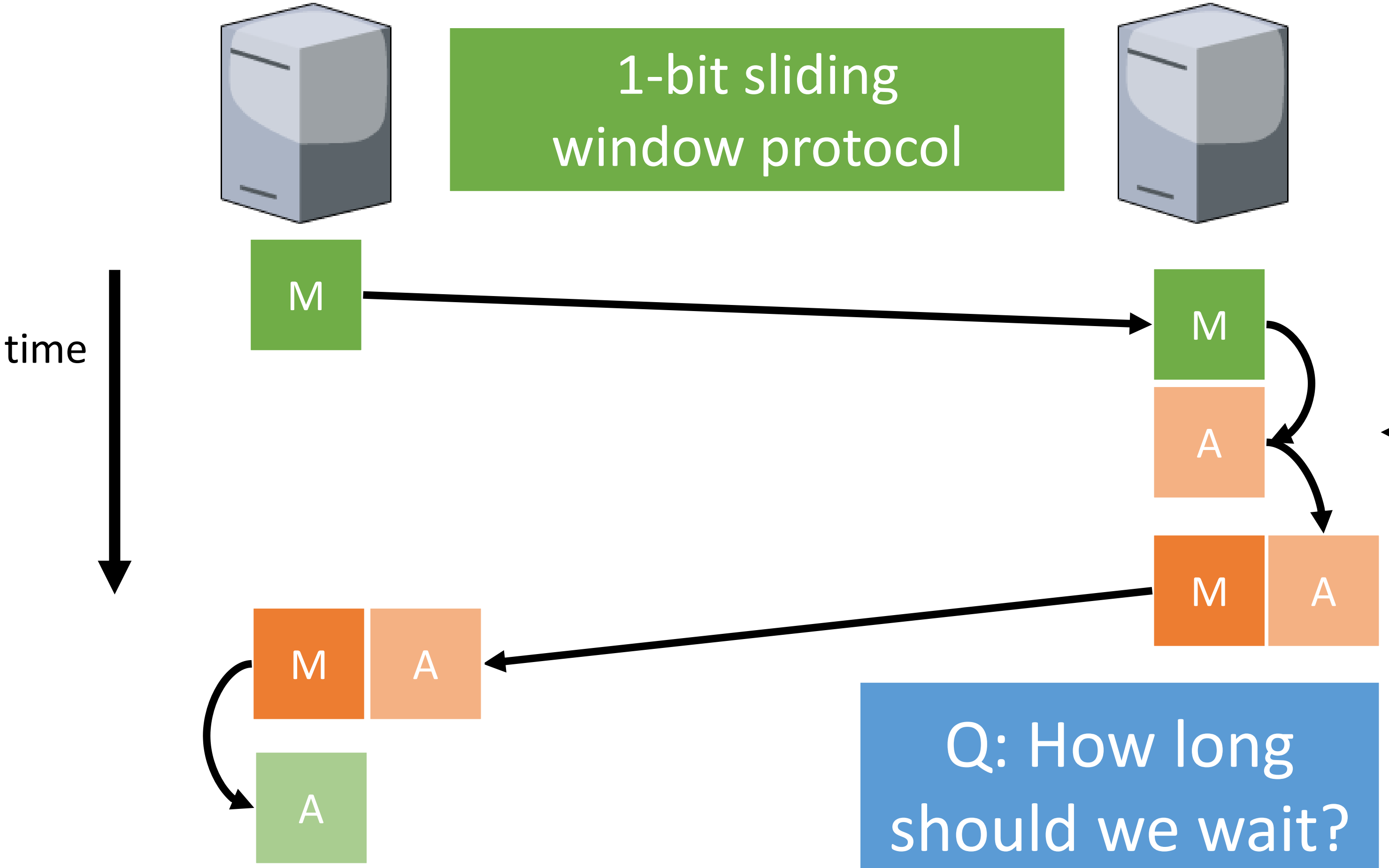
Bidirectional communication



Q: Can you make this protocol more *bandwidth efficient*?

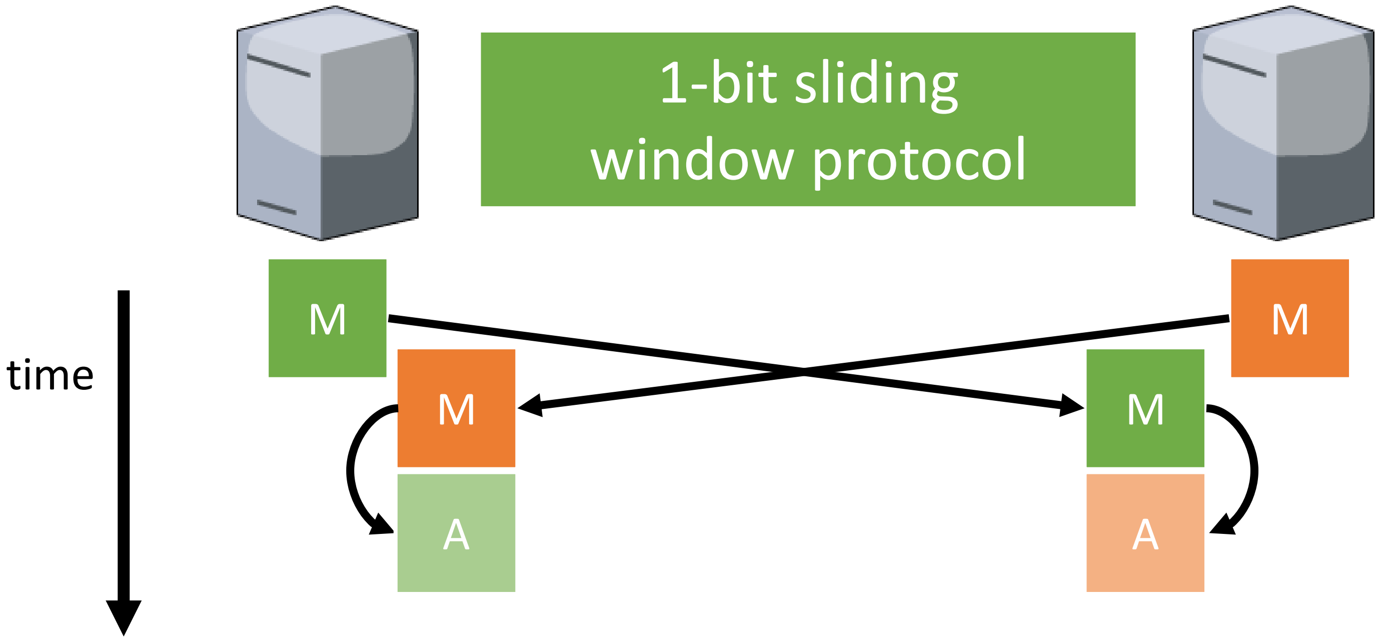
Acknowledgements With *piggybacking*

Wait before sending



Q: How long should we wait?

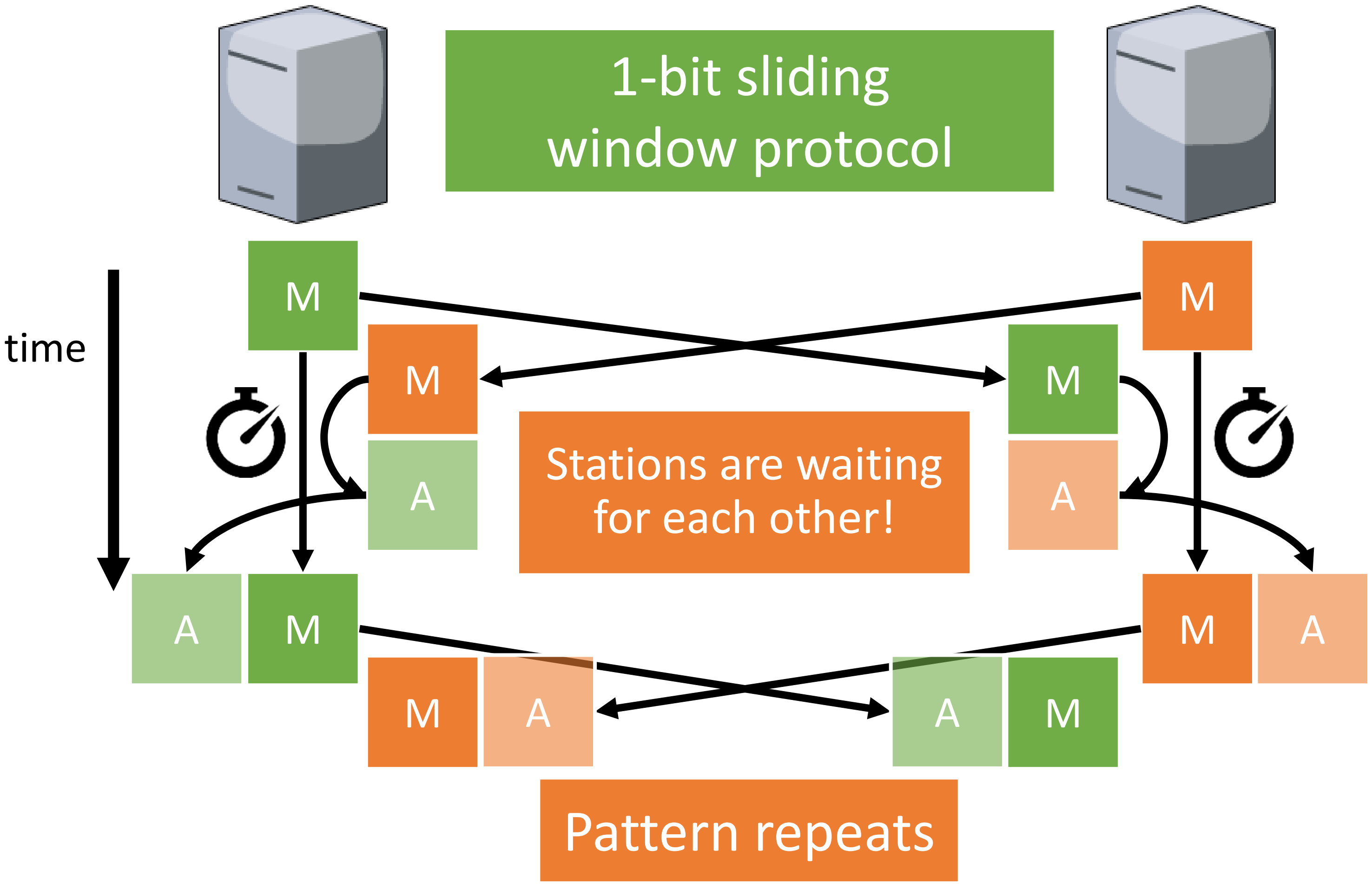
Stop-and-Wait



Q: What happens now?

Stop-and-Wait Special Case

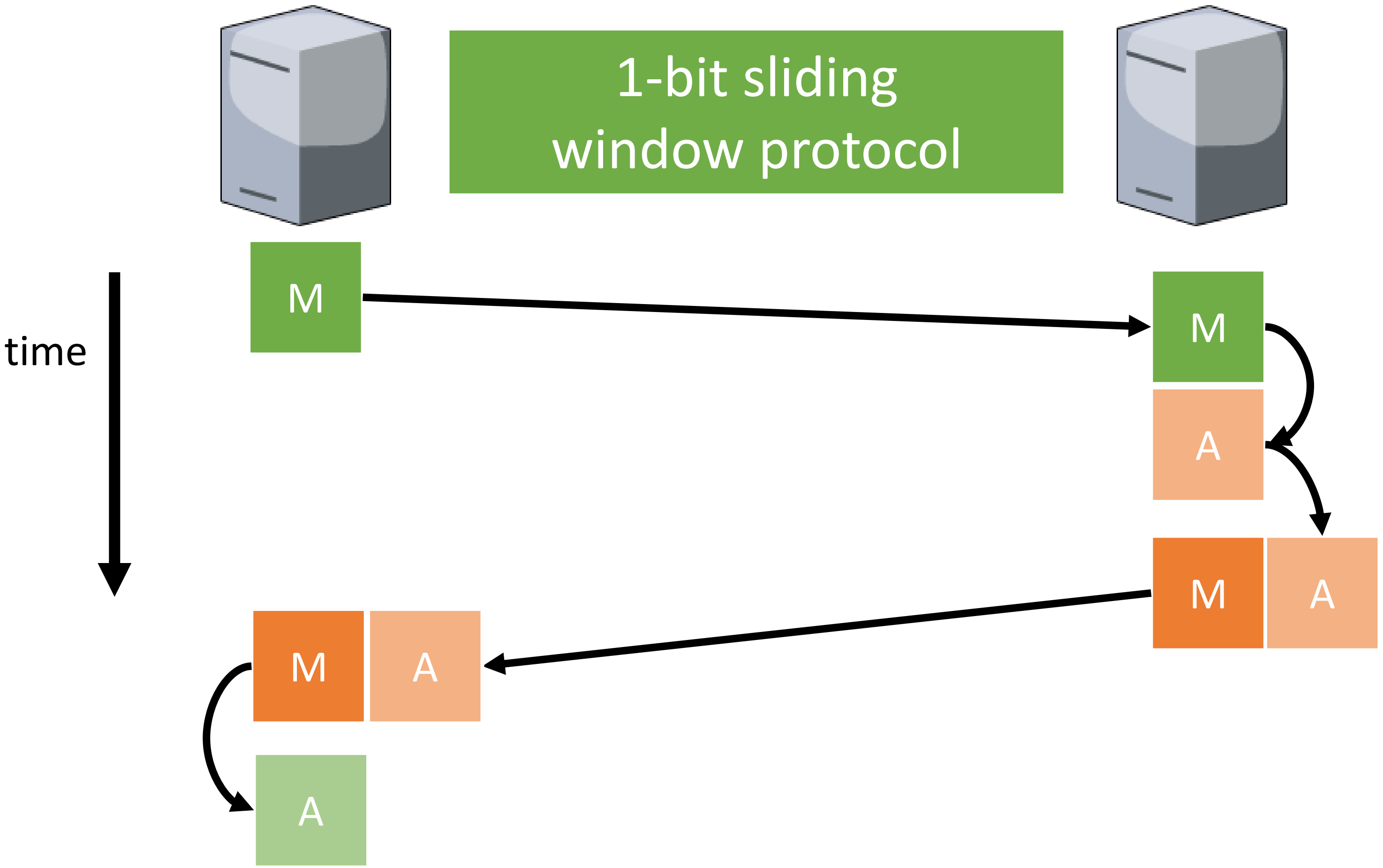
Even without errors, half the bandwidth is wasted on retransmissions



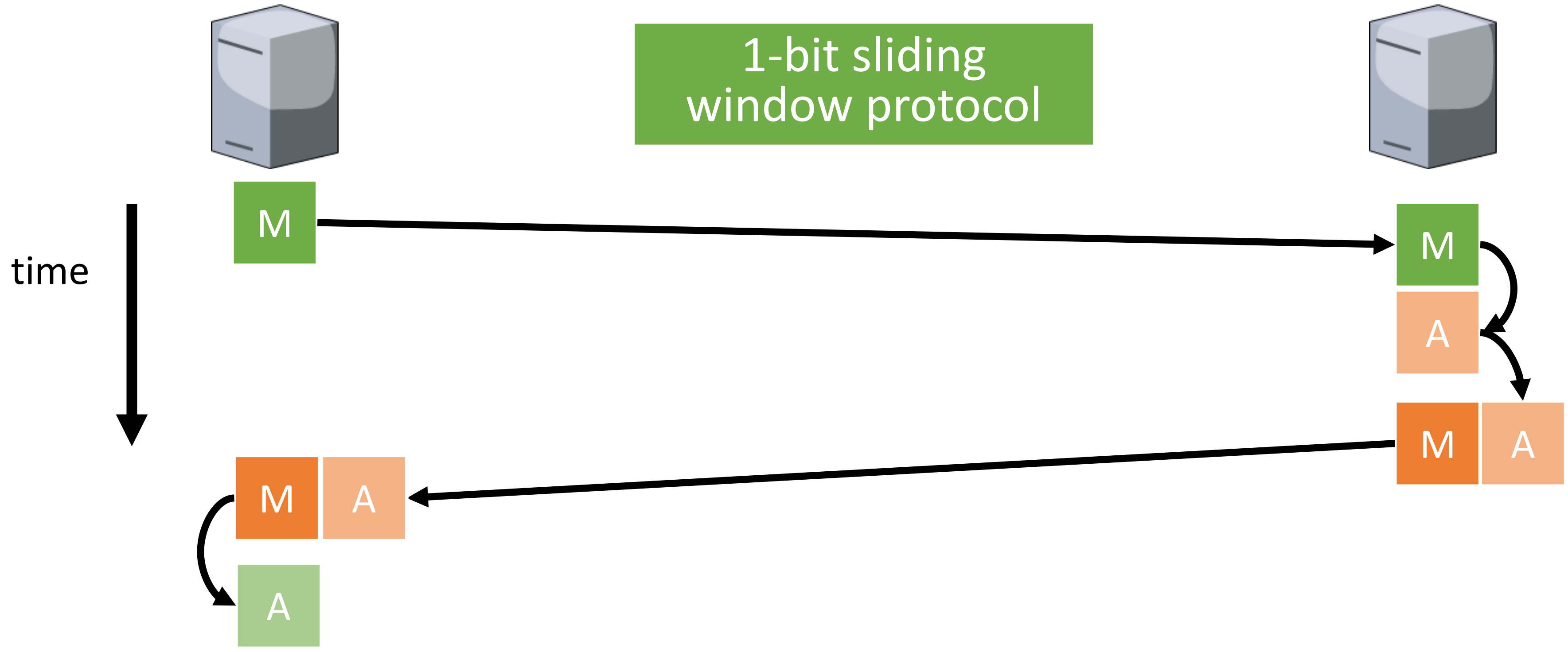
Sliding Window Protocols

Improving Performance using *Pipelining*

Stop-and-Wait:
A 1-Bit Sliding Window Protocol

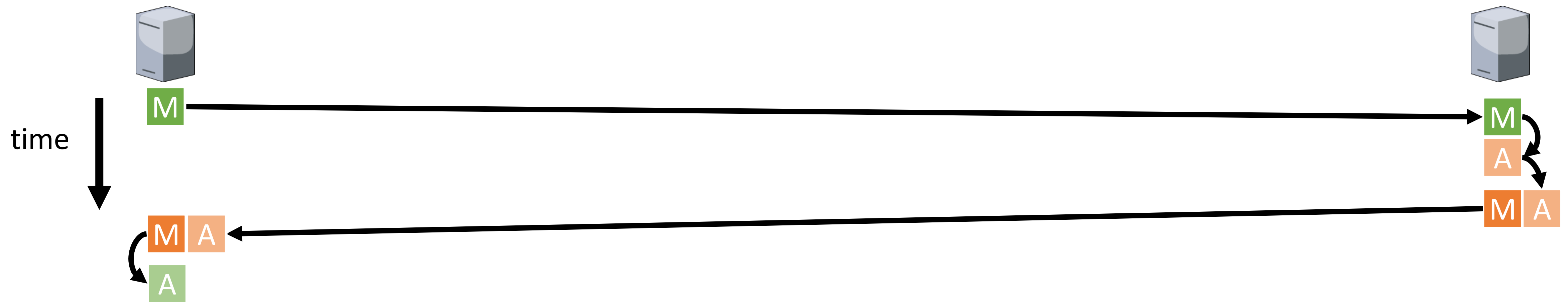


Stop-and-Wait:
A 1-Bit Sliding Window Protocol



Stop-and-Wait:
A 1-Bit Sliding Window Protocol

1-bit sliding window protocol



Bandwidth inefficient for high-latency channels

Sliding window protocols

Q: Which properties cause performance to decrease?

When using stop-and-wait, *data rate decreases* when:

- Latency increases
- Frame size decreases

Solution

Send next frame while waiting for acknowledgment of current frame

Sender window specifies how many frames a sender is allowed to send before waiting for an acknowledgement.

Receiver window specifies the range of frames that the receiver is allowed to accept.

Stop-and-Wait / ARQ:
A 1-Bit Sliding Window Protocol

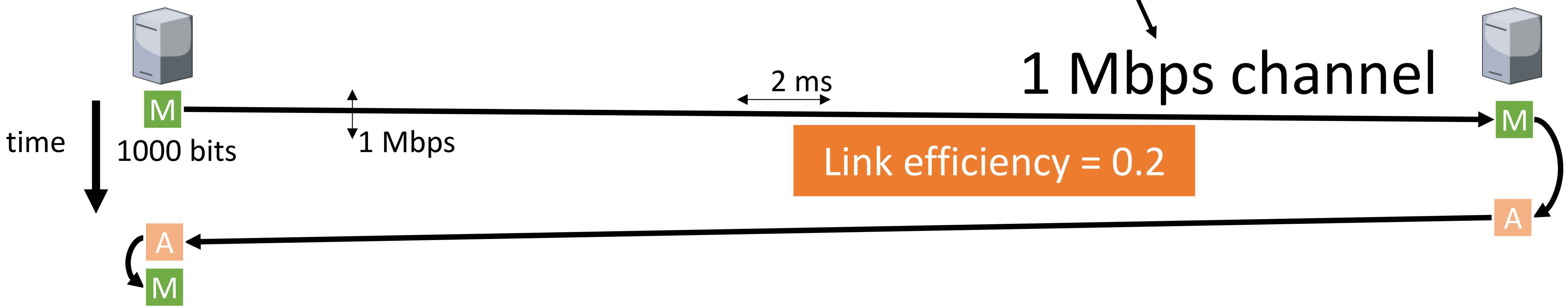
It takes $\frac{1{,}000}{1{,}000{,}000} = 0.001$ seconds to send frame

It takes 2 ms for the frame to arrive at the receiver, takes 2 ms for the (0-bit) acknowledgment to come back at the sender

1 frame per 0.001+0.002+0.002 seconds = 200 kbps

Small window reduces performance

1 Mbps channel



Stop-and-Wait: A 1-Bit Sliding Window Protocol

- Frame size (in bits/bytes): f
- Window size (in frames): w
- Bandwidth (max. data rate of physical channel): B_p
- Bandwidth (frames per second): B_f
- Propagation delay (in seconds): D

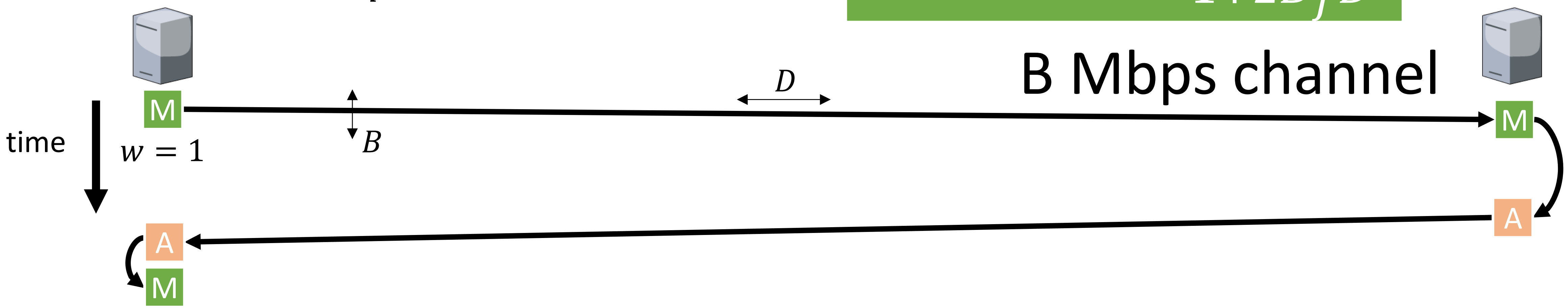
It takes $\frac{f}{B_p}$ seconds to send frame, $\frac{B_p}{f} = B_f$

It takes D s for the frame to arrive at the receiver, takes D s for the (0-bit) acknowledgment to come back at the sender

1 frame per $\frac{f}{B_p} + 2 \times D$ seconds

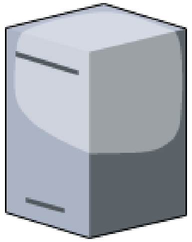
Link utilization $\leq \frac{w}{1+2B_f D}$

B Mbps channel

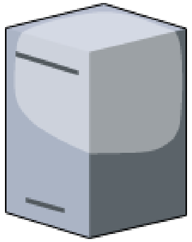


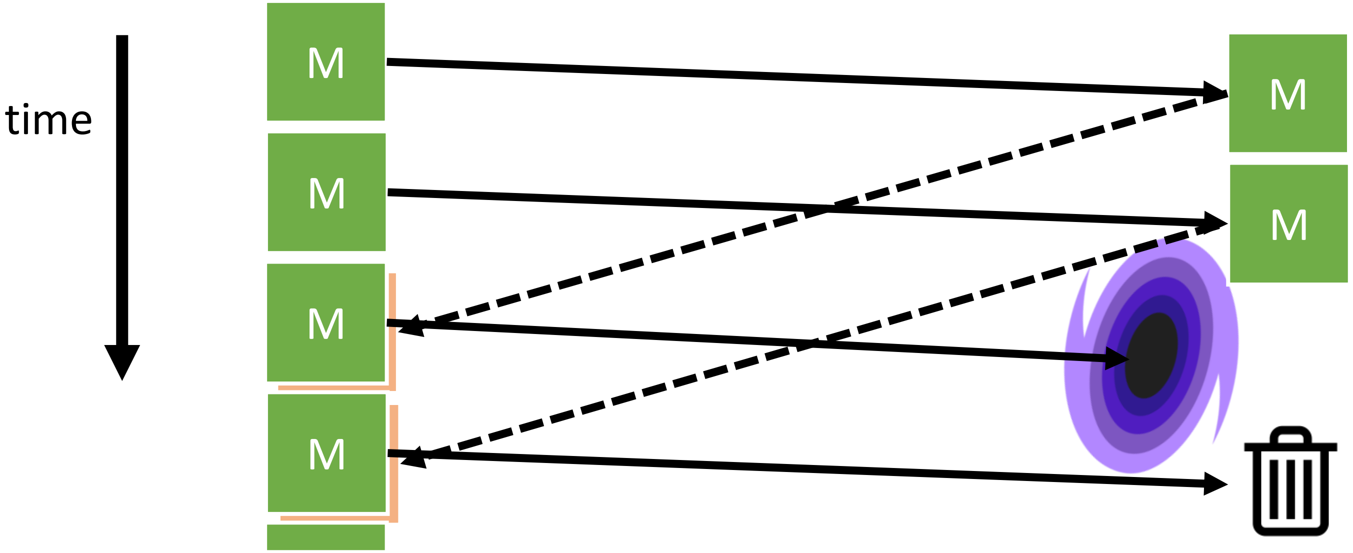
Go-Back-N

IF	ID	EX	MEM	WB				
	IF	ID	EX	MEM	WB			
		IF	ID	EX	MEM	WB		
			IF	ID	EX	MEM	WB	
				IF	ID	EX	MEM	WB



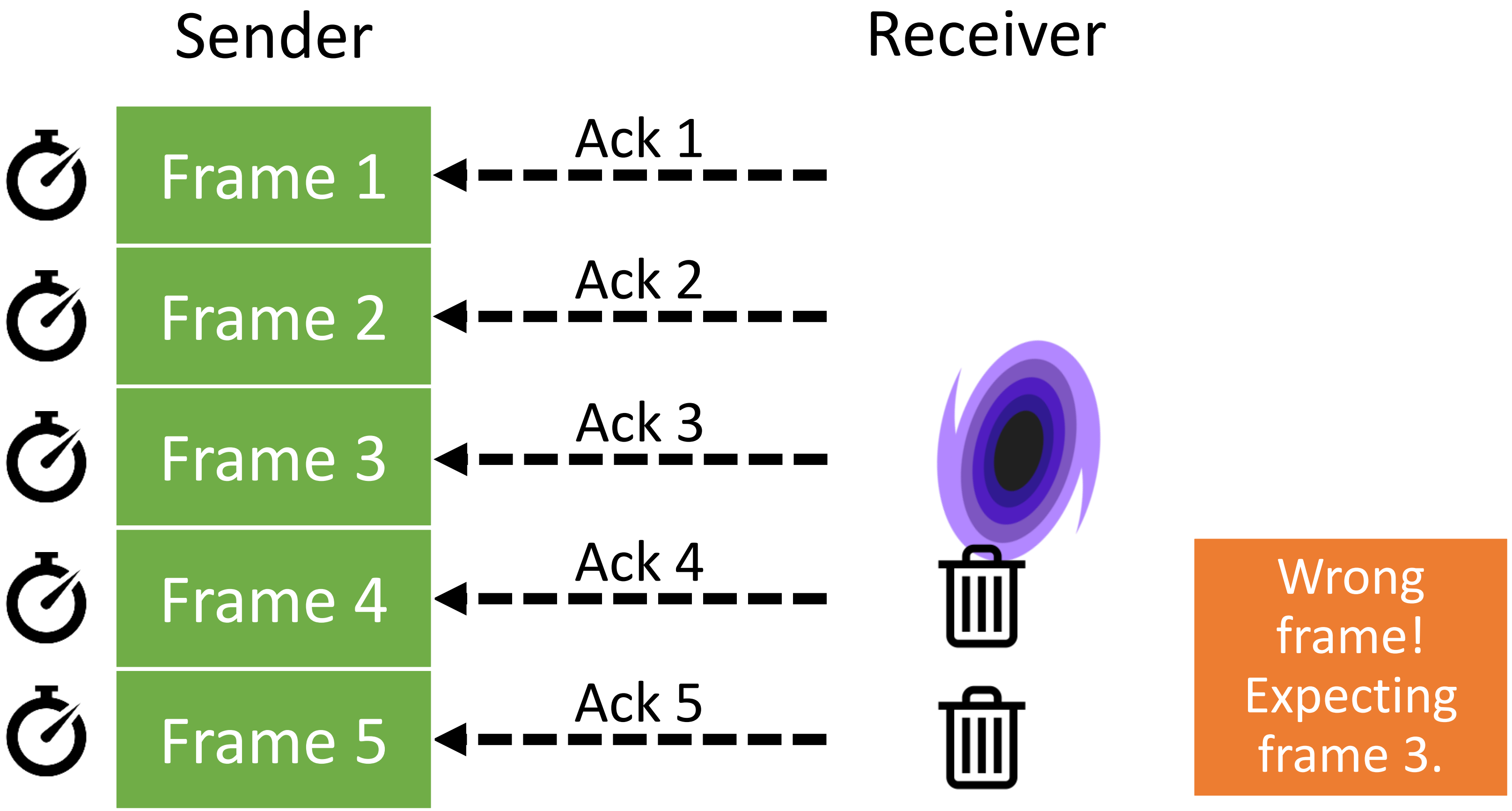
Q: Do you recognize this type of optimization?





Q: What is the size of the receiver window?

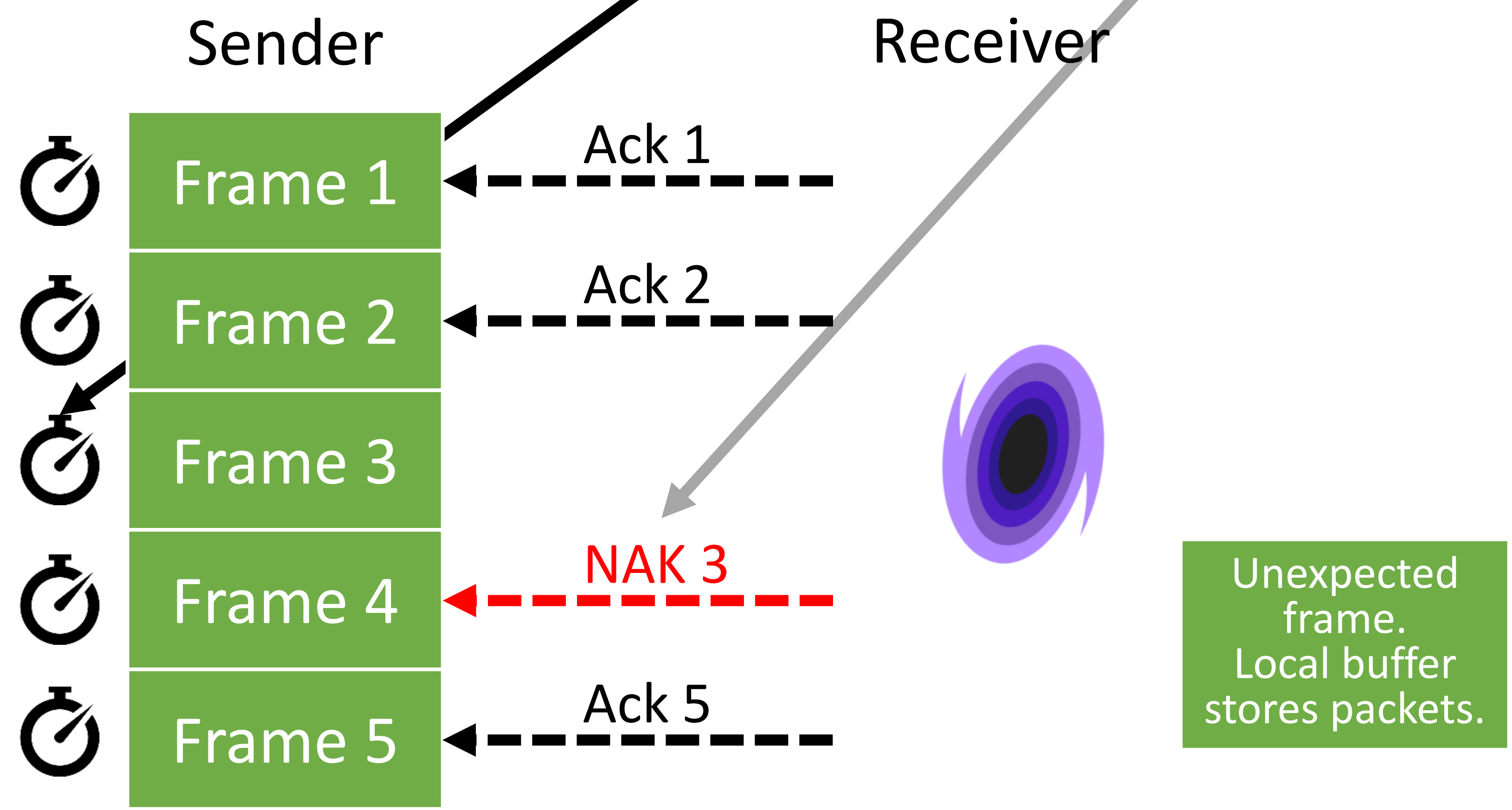
Go-Back-N



Sender
Receiver
Frame 1
Frame 2
Frame 3
Frame 4
Frame 5
Ack 1
Ack 2
Ack 3
Ack 4
Ack 5
Wrong frame! Expecting frame 3.

Selective repeat

Selective repeat can use timers, negative acknowledgements, or both.



Delivering frames in the order in which they were sent is easy after adding sequence numbers.

Data Link Layer — Roadmap

Part 1

- **Framing**
- **Flow Control**
- **Guaranteed Delivery**
- **Sliding Window Protocols**

Part 2

- Error detection
- Error correction