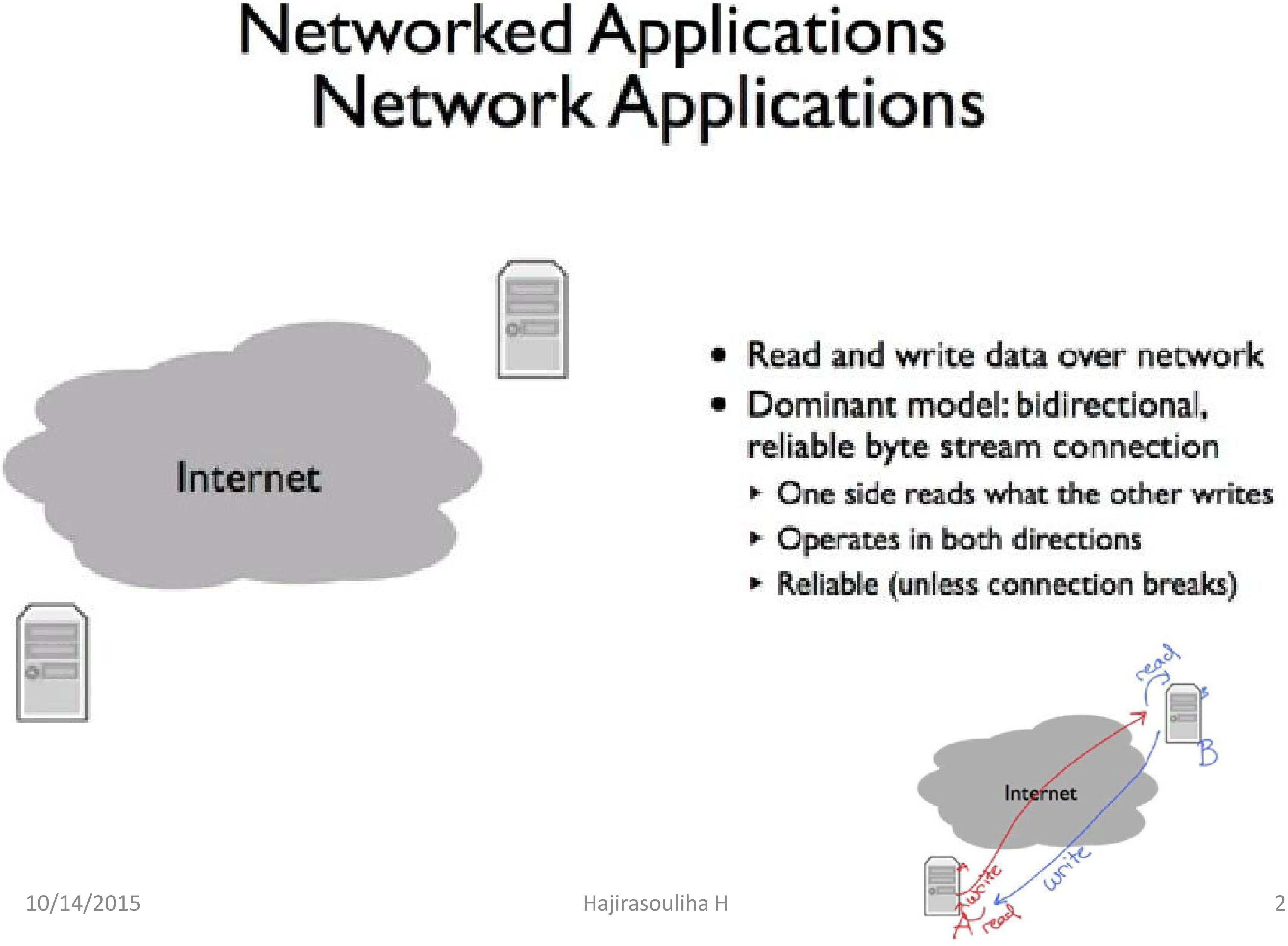
ACN Lecture 2

Critical network infrastructure services:

1. A day in the life of an application
2. The four layer Internet model
3. The IP service model

A day in the life of an application

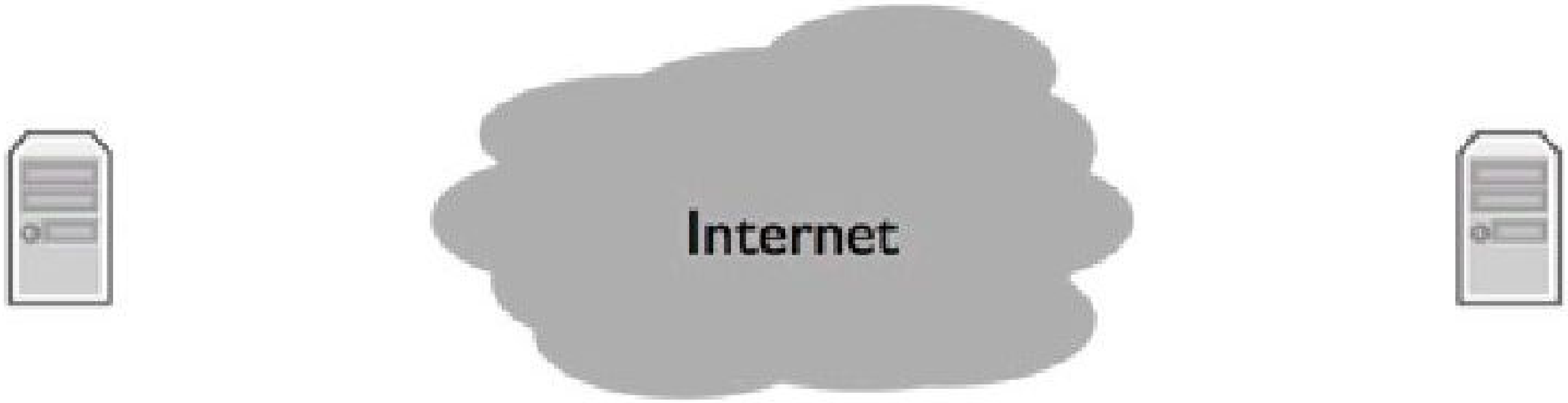


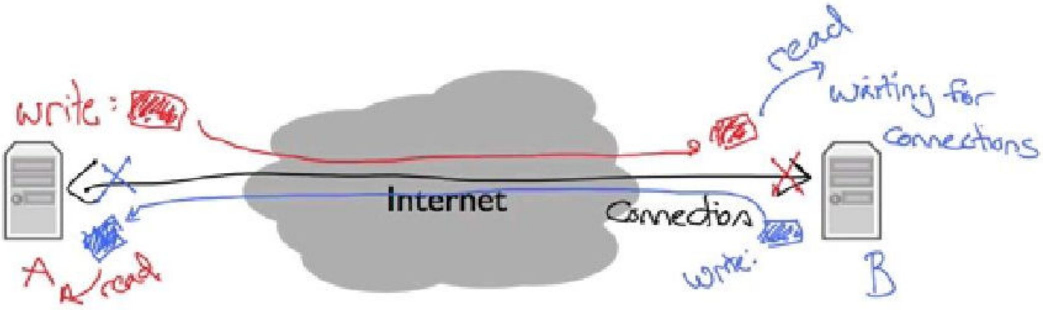
Networked Applications
Network Applications

- Read and write data over network
- Dominant model: bidirectional, reliable byte stream connection
  - One side reads what the other writes
  - Operates in both directions
  - Reliable (unless connection breaks)

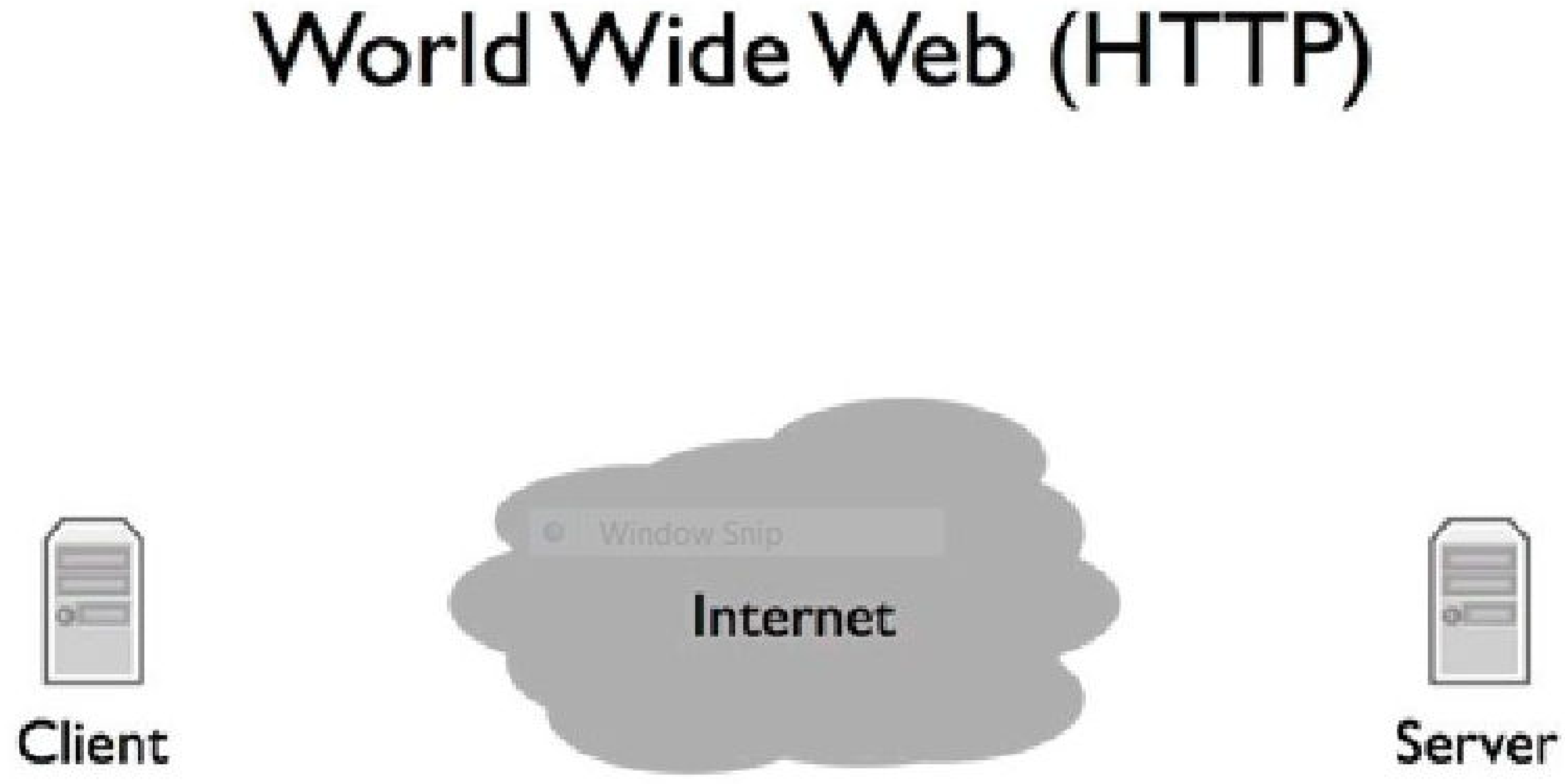
A day in the life of an application

Byte Stream Model

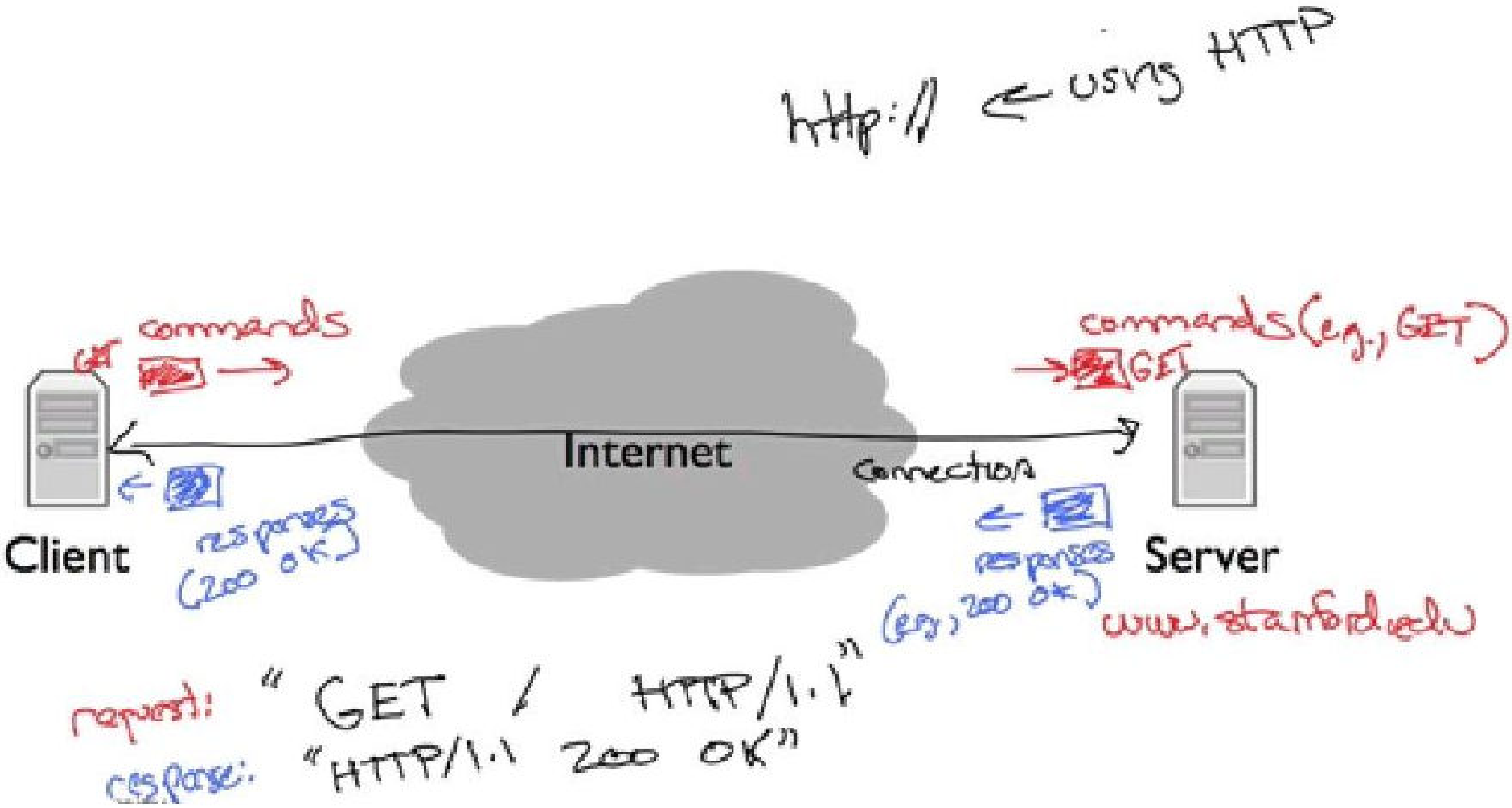




A day in the life of an application



A day in the life of an application



request: "GET / HTTP/1.1"

response: "HTTP/1.1 200 OK"

Two: Types of HTTP connections

1. non-persistent HTTP

- at most one object sent over TCP connection and Then connection is closed.
- downloading multiple objects requires multiple connections.

2. *persistent HTTP*

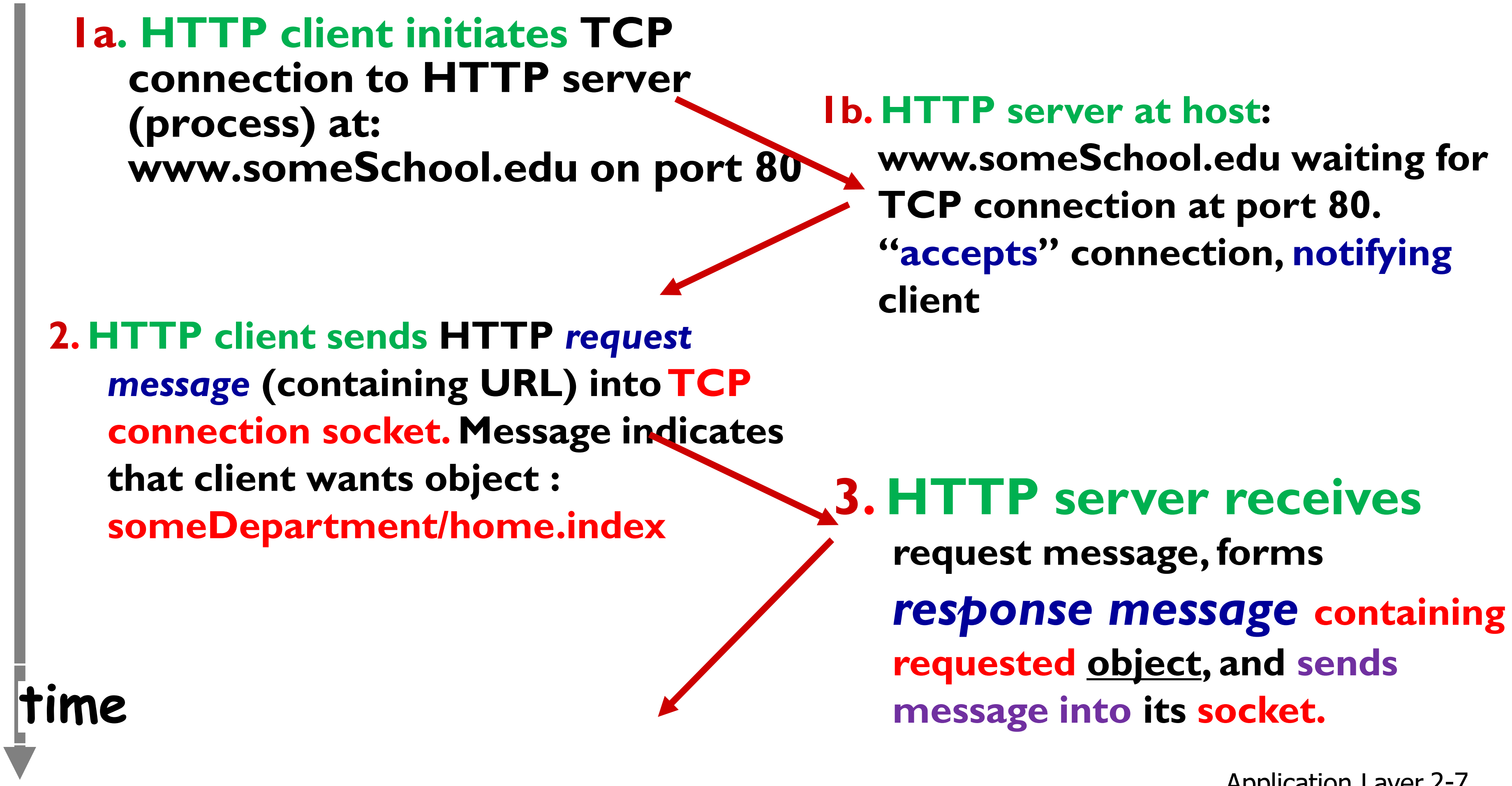
- multiple objects can be sent over single TCP connection between client, and server.

1. Non-persistent HTTP

suppose user enters URL:

www.someSchool.edu/someDepartment/home.index

Assume: it contains text, and references to 10 jpeg images.



1a. HTTP client initiates TCP connection to HTTP server (process) at: www.someSchool.edu on port 80

1b. HTTP server at host: www.someSchool.edu waiting for TCP connection at port 80. "accepts" connection, notifying client

2. HTTP client sends HTTP *request message* (containing URL) into TCP connection socket. Message indicates that client wants object : someDepartment/home.index

3. HTTP server receives request message, forms *response message* containing requested <u>object</u>, and sends message into its socket.

Non-persistent HTTP (cont.)

4. HTTP server closes TCP connection.

5. HTTP client receives response message containing html file, displays html. Parsing html file, finds 10 referenced jpeg objects

6. Steps 1-5 repeated for each of 10 jpeg objects.

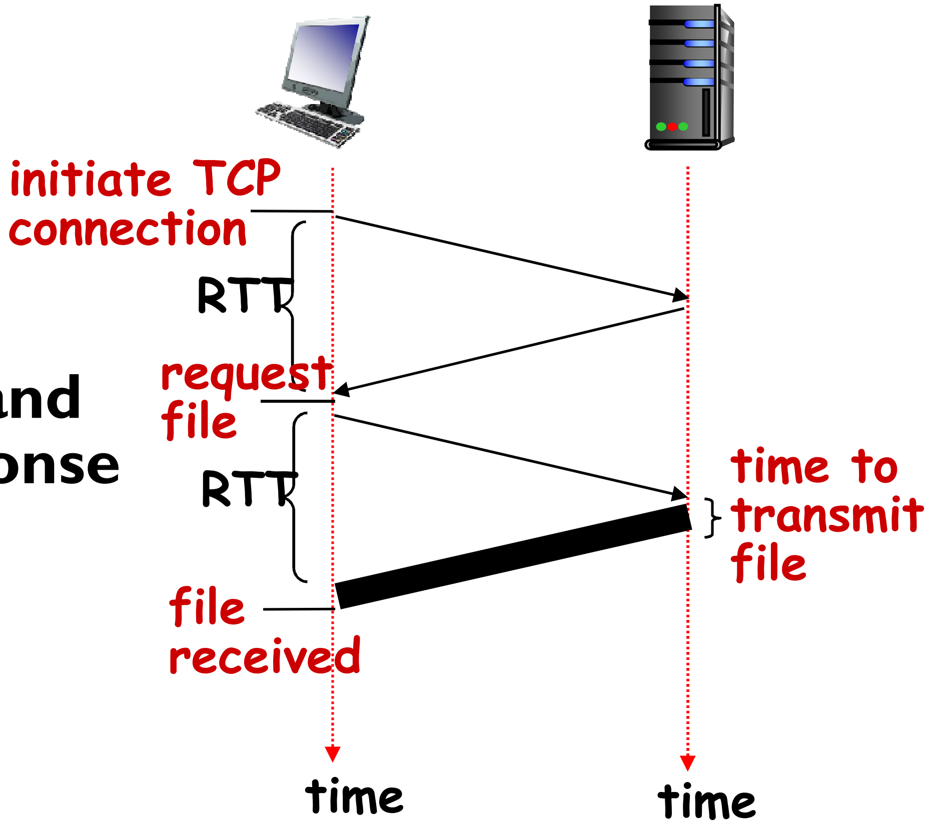
time

Non-persistent HTTP: response time

RTT (definition): time for a small packet to travel from client to server and back.

HTTP response time:

- one RTT to initiate TCP connection +
- one RTT for HTTP request and first few bytes of HTTP response to return +
- file transmission time.



Non-persistent HTTP response time =2RTT+ file transmission time

Non-persistent HTTP issues:

- requires 2 RTTs per object.
- OS overhead for *each* TCP connection.
- browsers often open parallel TCP connections to fetch referenced objects.

Persistent HTTP

persistent HTTP:

- server leaves connection open after ***sending a response.***
- subsequent HTTP messages between same client/server sent over open connection
- client sends requests as soon as it encounters a referenced object.
- As little as one RTT for all the referenced objects.

HTTP request message

- 2 types of HTTP messages: <u>*request*</u>, <u>*response*</u>
- HTTP request message: ASCII (<u>human-readable format</u>)

carriage return character

line-feed character

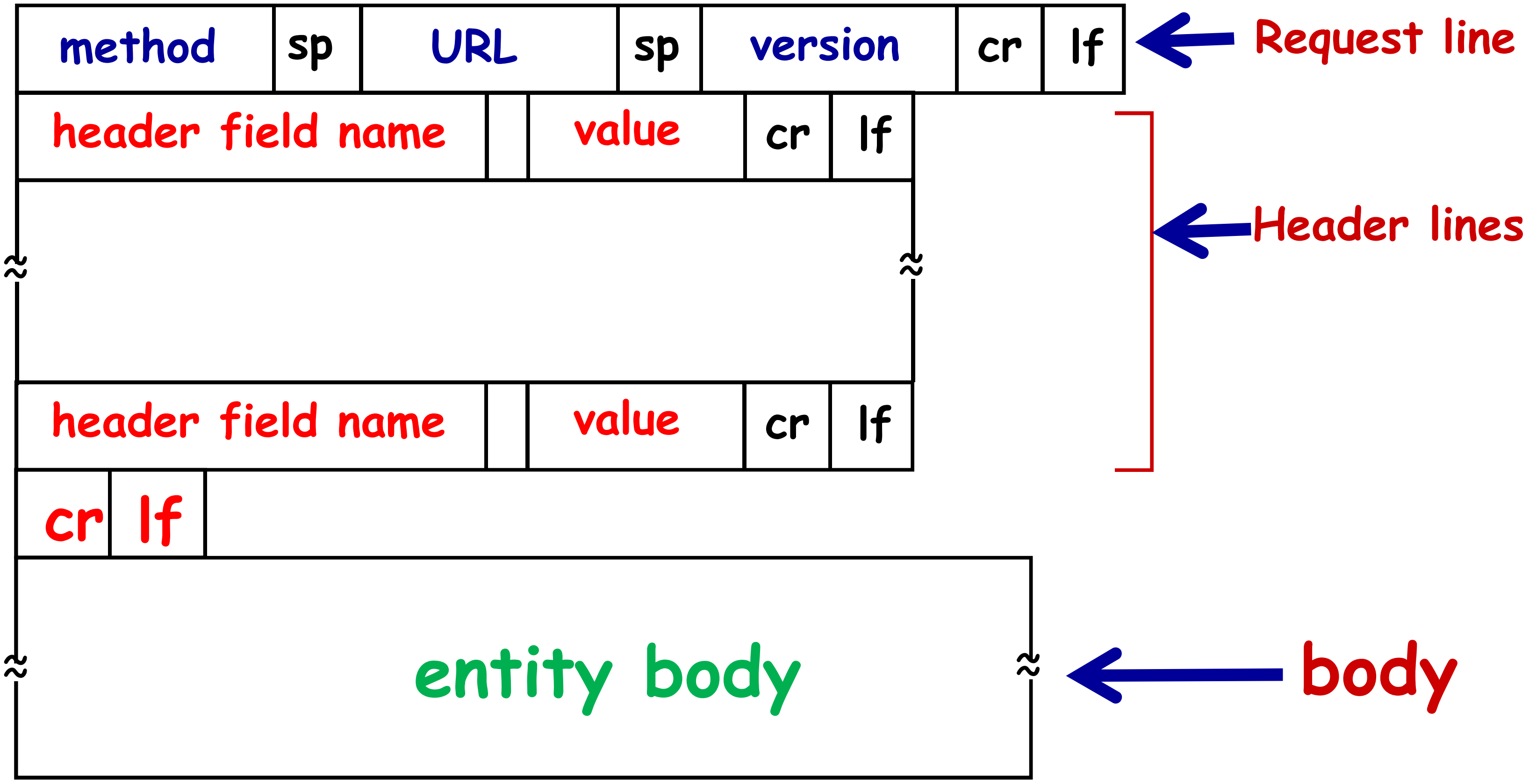
request line
(<u>GET</u>, <u>POST</u>,
<u>HEAD</u> commands)

Header lines

```
GET /index.html HTTP/1.1\r\n
Host: www-net.cs.umass.edu\r\n
User-Agent: Firefox/3.6.10\r\n
Accept: text/html,application/xhtml+xml\r\n
Accept-Language: en-us,en;q=0.5\r\n
Accept-Encoding: gzip,deflate\r\n
Accept-Charset: ISO-8859-1,utf-8;q=0.7\r\n
Keep-Alive: 115\r\n
Connection: keep-alive\r\n
\r\n
```

carriage return, line feed <u>at start of line</u> indicates <u>end of header lines</u>

HTTP request message: general format



Uploading form input

POST method:

- web pages often includes form inputs.
- input is uploaded to server in entity body

URL method:

- uses GET method
- input is uploaded in URL field of request line: e.g.:

```
www.somesite.com/animalsearch?monkeys&banana
```

Method types:

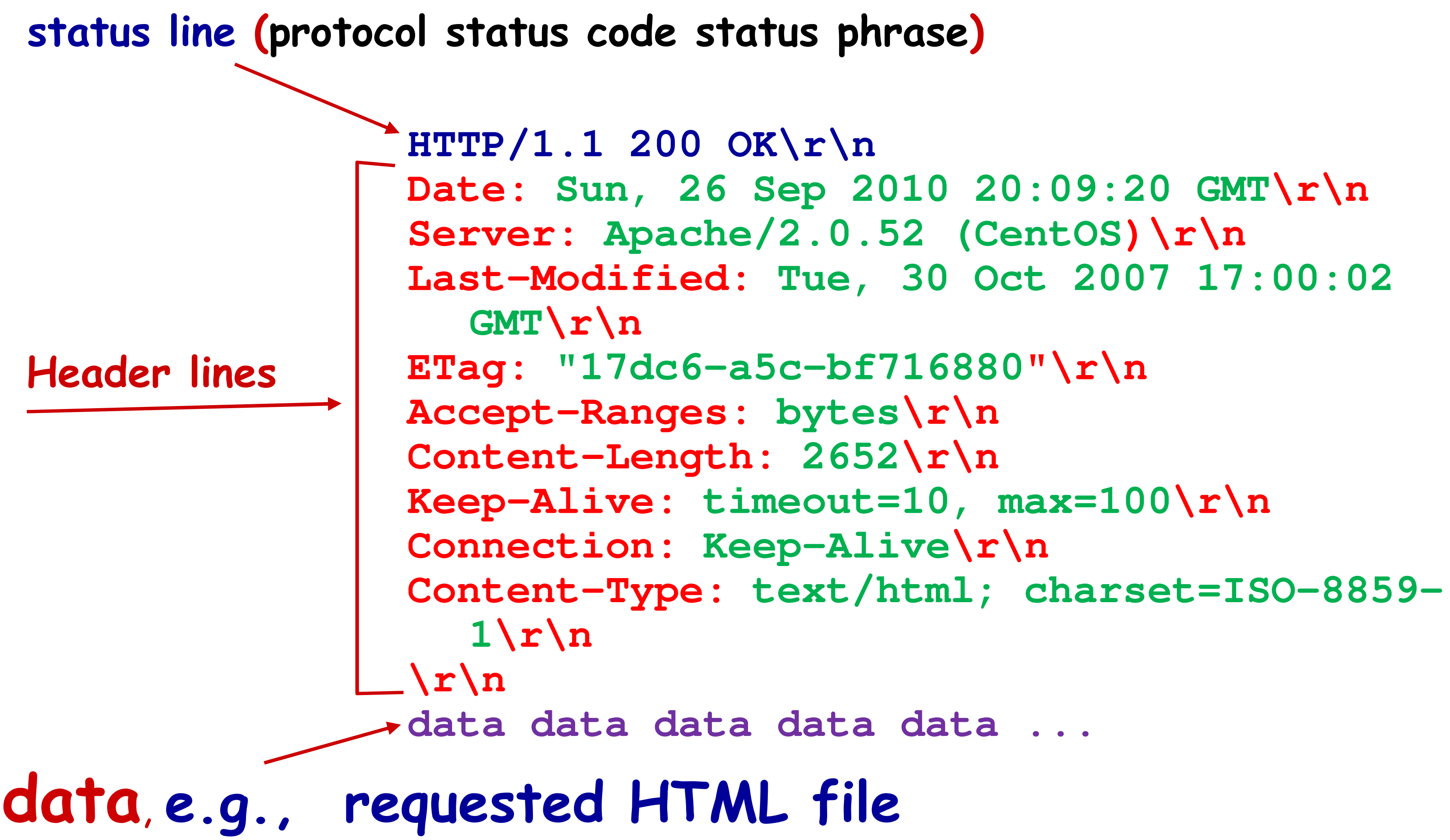
HTTP/1.0:

- GET
- POST
- HEAD
  - asks server to leave requested object out of response

HTTP/1.1:

- GET, POST, HEAD
- PUT
  - uploads file in entity body to path specified in URL field
- DELETE
  - deletes file specified in the URL field

HTTP response message:



status line (protocol status code status phrase)

```
HTTP/1.1 200 OK\r\n
Date: Sun, 26 Sep 2010 20:09:20 GMT\r\n
Server: Apache/2.0.52 (CentOS)\r\n
Last-Modified: Tue, 30 Oct 2007 17:00:02
   GMT\r\n
ETag: "17dc6-a5c-bf716880"\r\n
Accept-Ranges: bytes\r\n
Content-Length: 2652\r\n
Keep-Alive: timeout=10, max=100\r\n
Connection: Keep-Alive\r\n
Content-Type: text/html; charset=ISO-8859-
   1\r\n
\r\n
data data data data data ...
```

Header lines

data, e.g., requested HTML file

HTTP response status codes

- status code appears in 1st line in server-to-client response message.
- some sample codes are:

200 OK

- request succeeded, requested object later in this msg

301 Moved Permanently

- requested object moved, new location specified later in this msg (Location:)

400 Bad Request

- request msg not understood by server

404 Not Found

- requested document not found on this server

505 HTTP Version Not Supported

Trying out HTTP (client side) for yourself

1. Telnet to your favorite Web server: e.g.:

```
telnet cis.poly.edu 80
```

opens TCP connection to port 80 (default HTTP server port) at cis.poly.edu. anything typed is sent to port 80 at cis.poly.edu

2. type in a GET HTTP request:

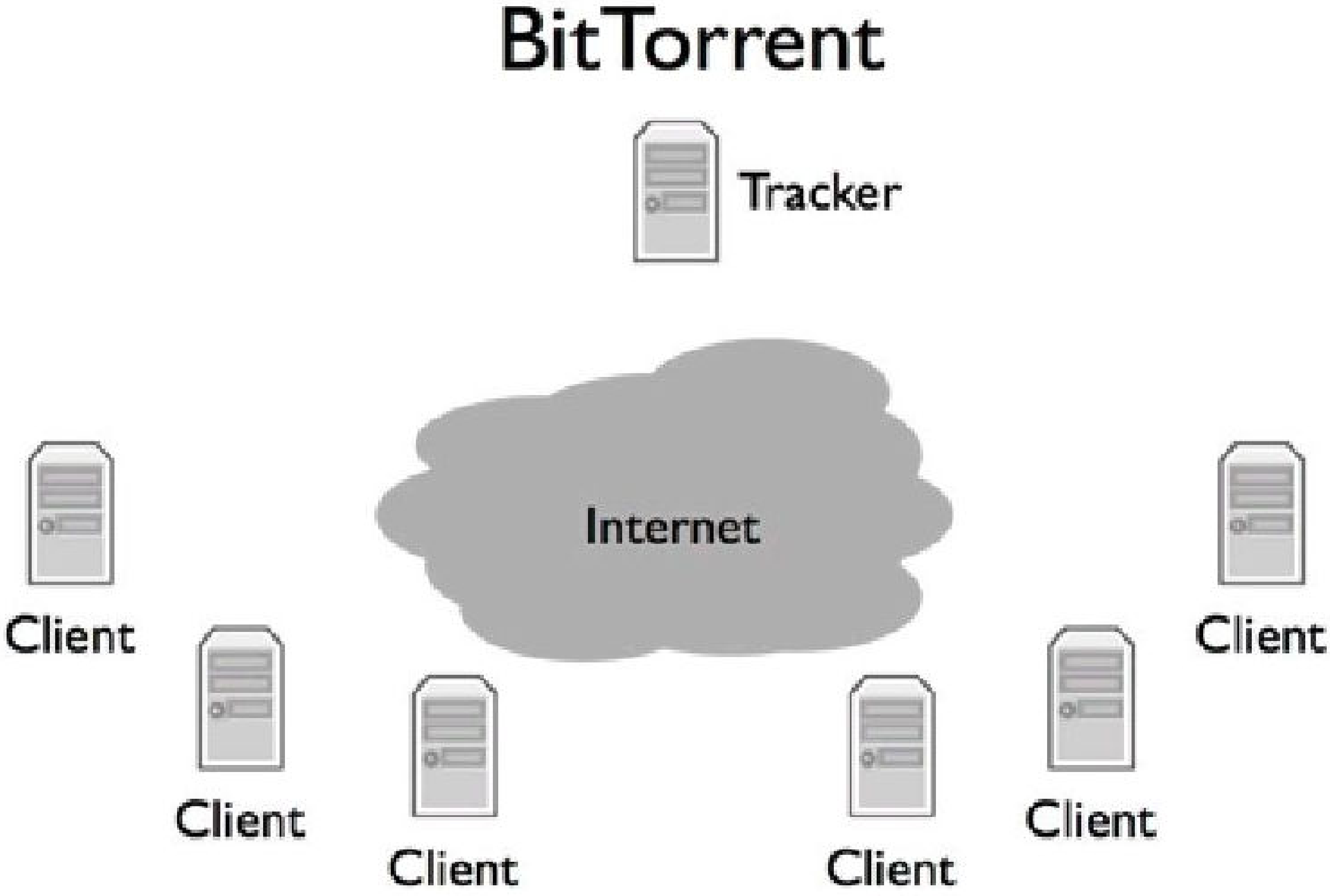
```
GET /~ross/ HTTP/1.1
Host: cis.poly.edu
```

By typing this in (hit carriage return twice), you send this minimal (but complete) GET request to HTTP server

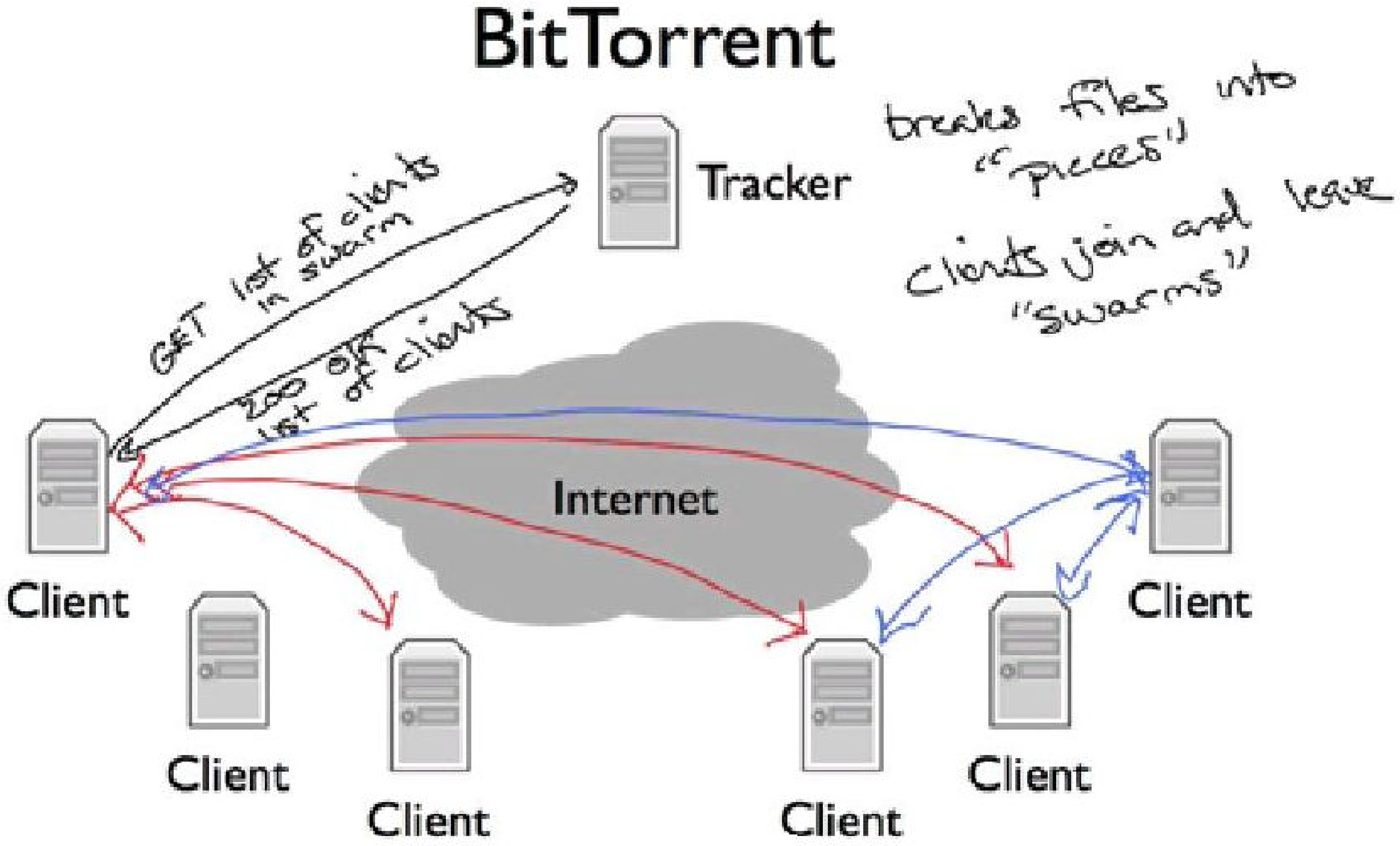
3. look at response message sent by HTTP server!

(or use **Wireshark** to **look at captured HTTP request/response**)

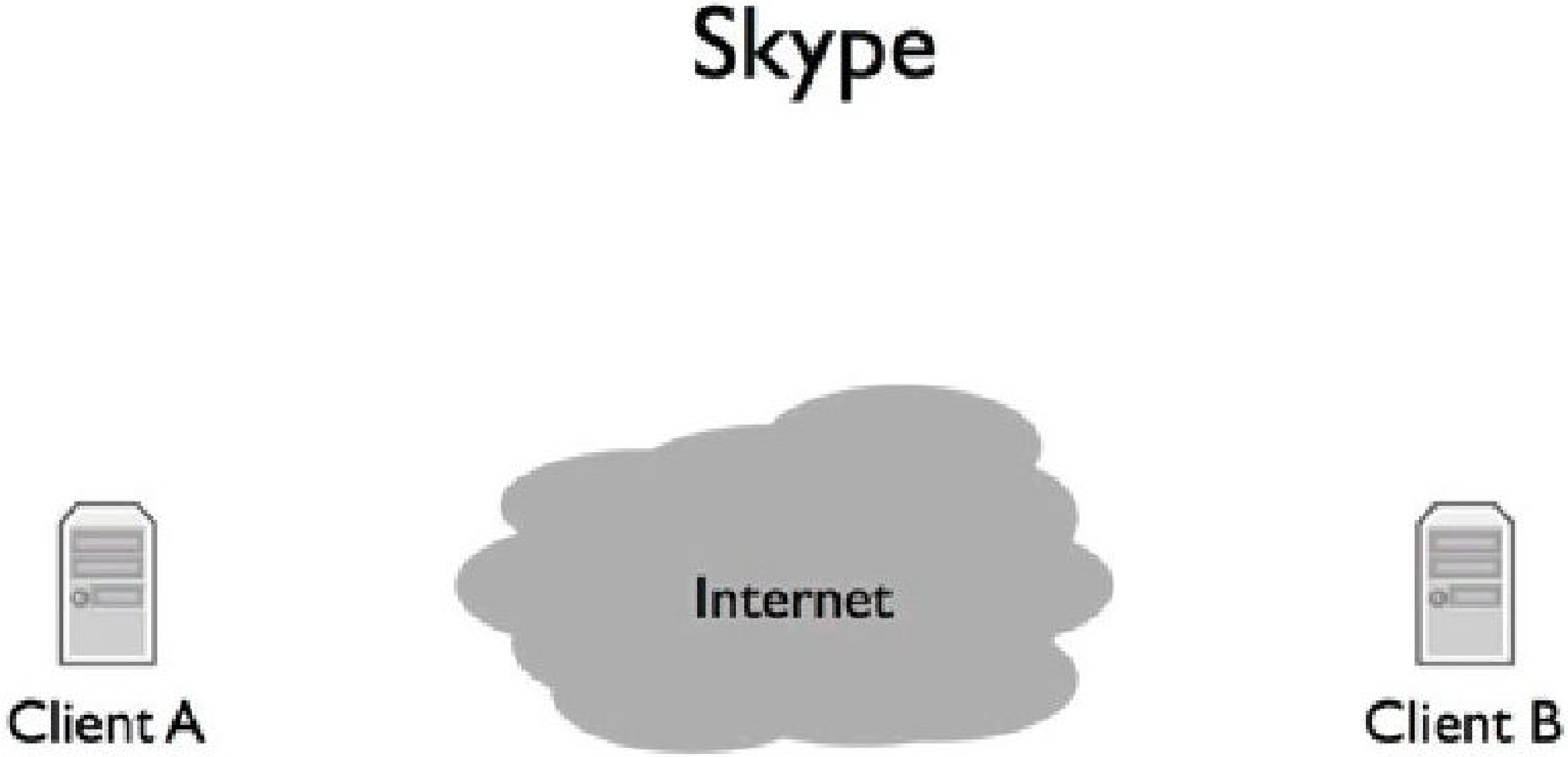
A day in the life of an application



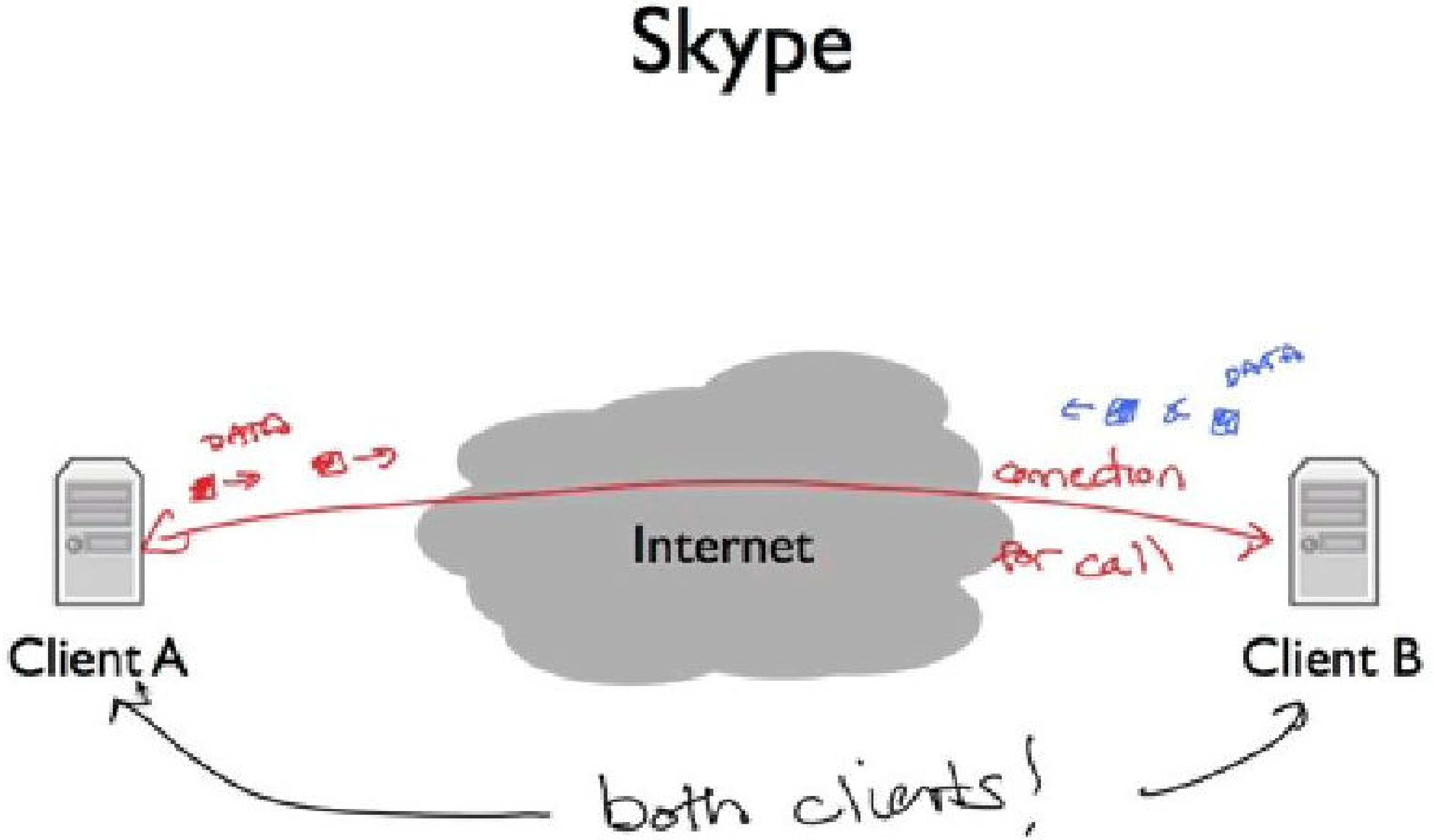
A day in the life of an application


BitTorrent
Tracker
Internet
Client
Client
Client
Client
Client
Client
GET list of clients in swarm
200 OK list of clients
breaks files into "pieces"
clients join and leave "swarms"

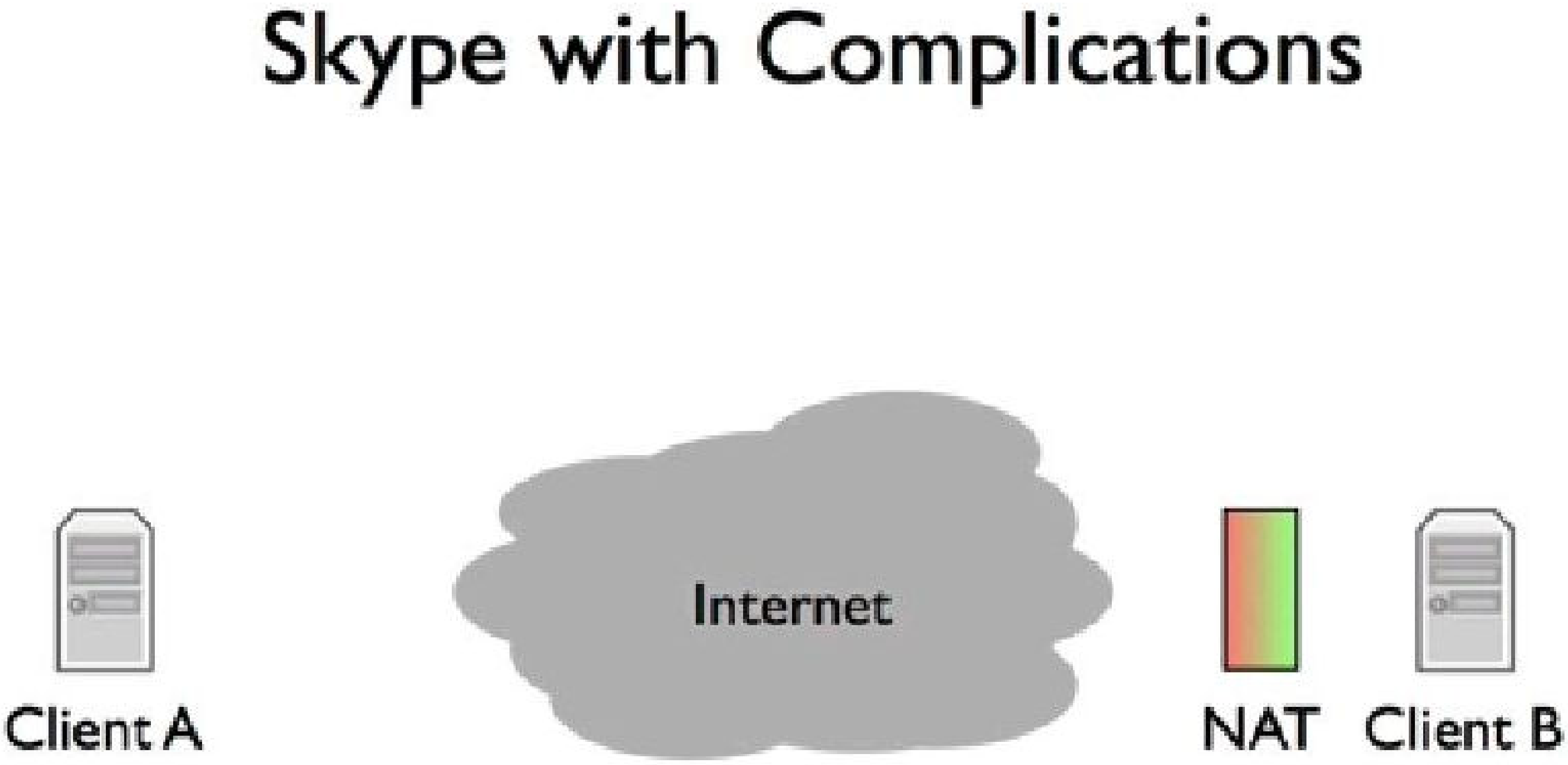
A day in the life of an application



A day in the life of an application

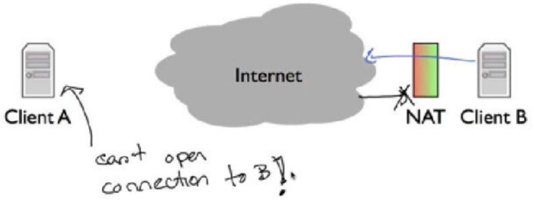


A day in the life of an application



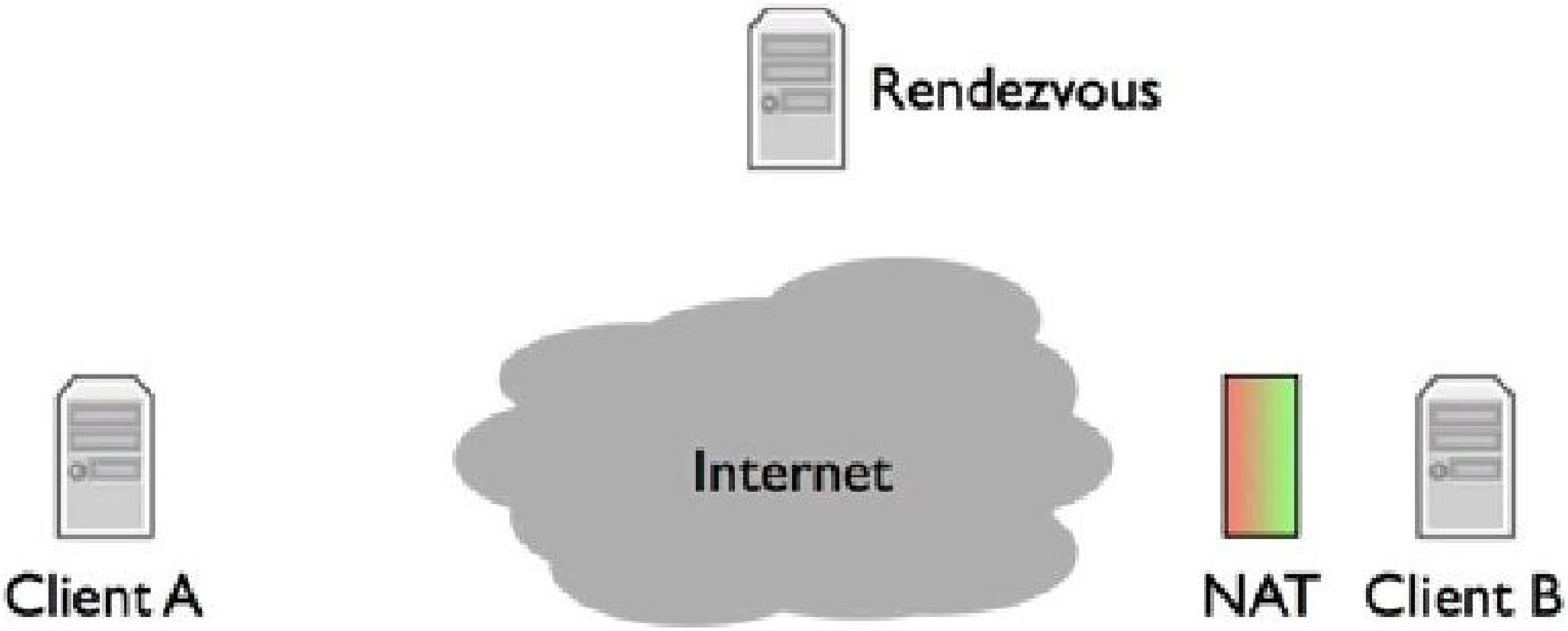
Skype with Complications

Skype with Complications

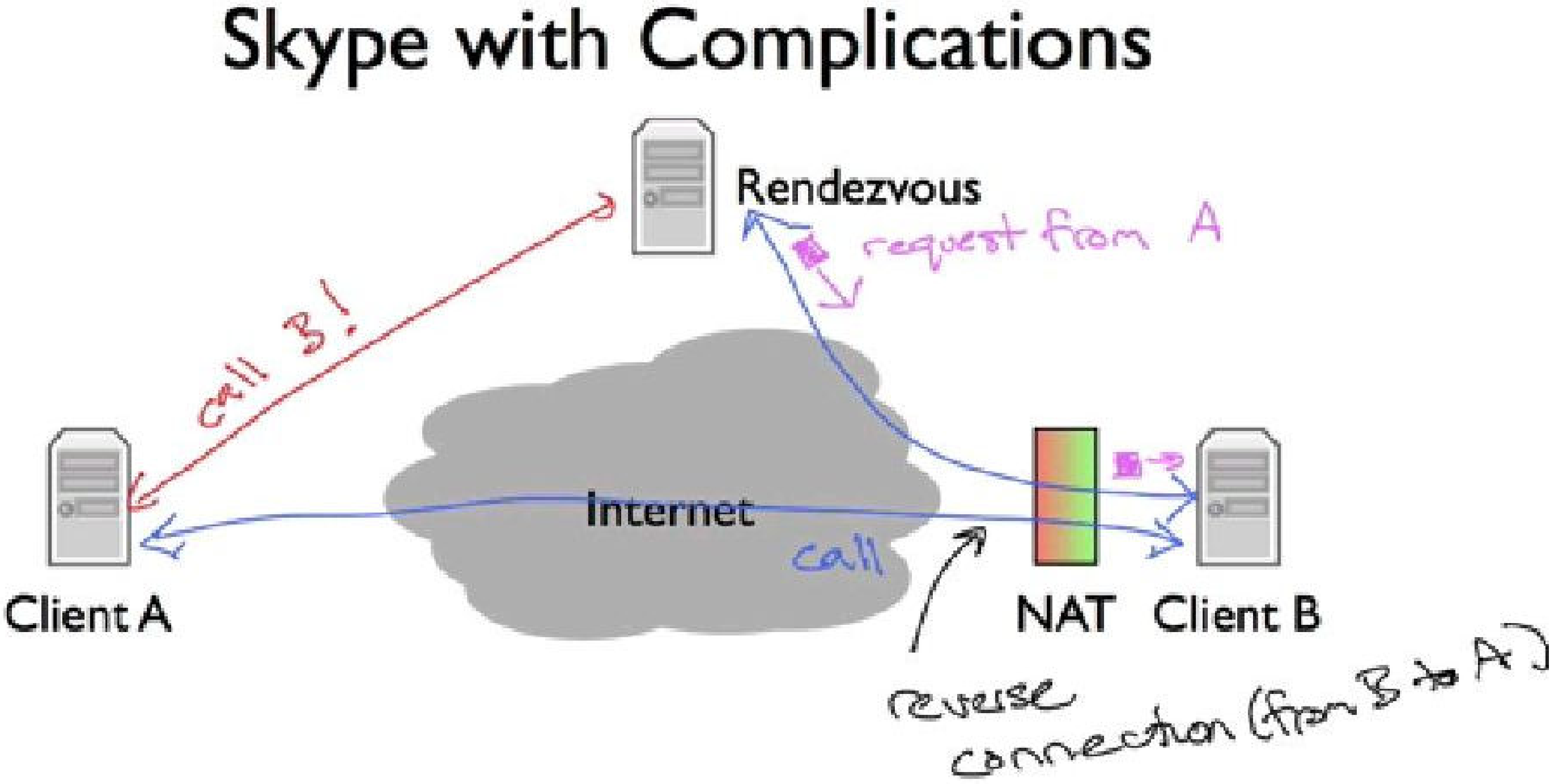


A day in the life of an application

Skype with Complications

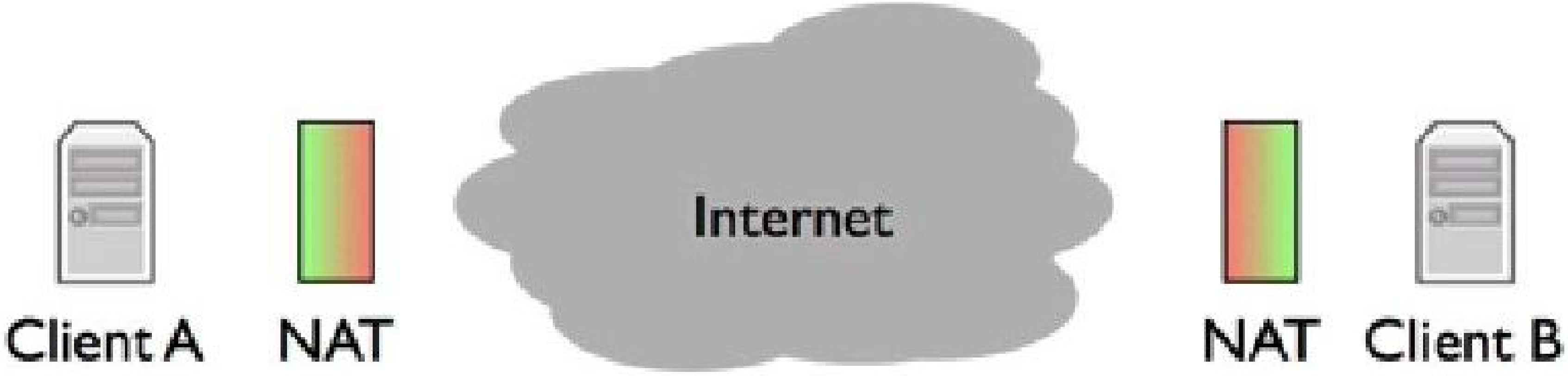


A day in the life of an application

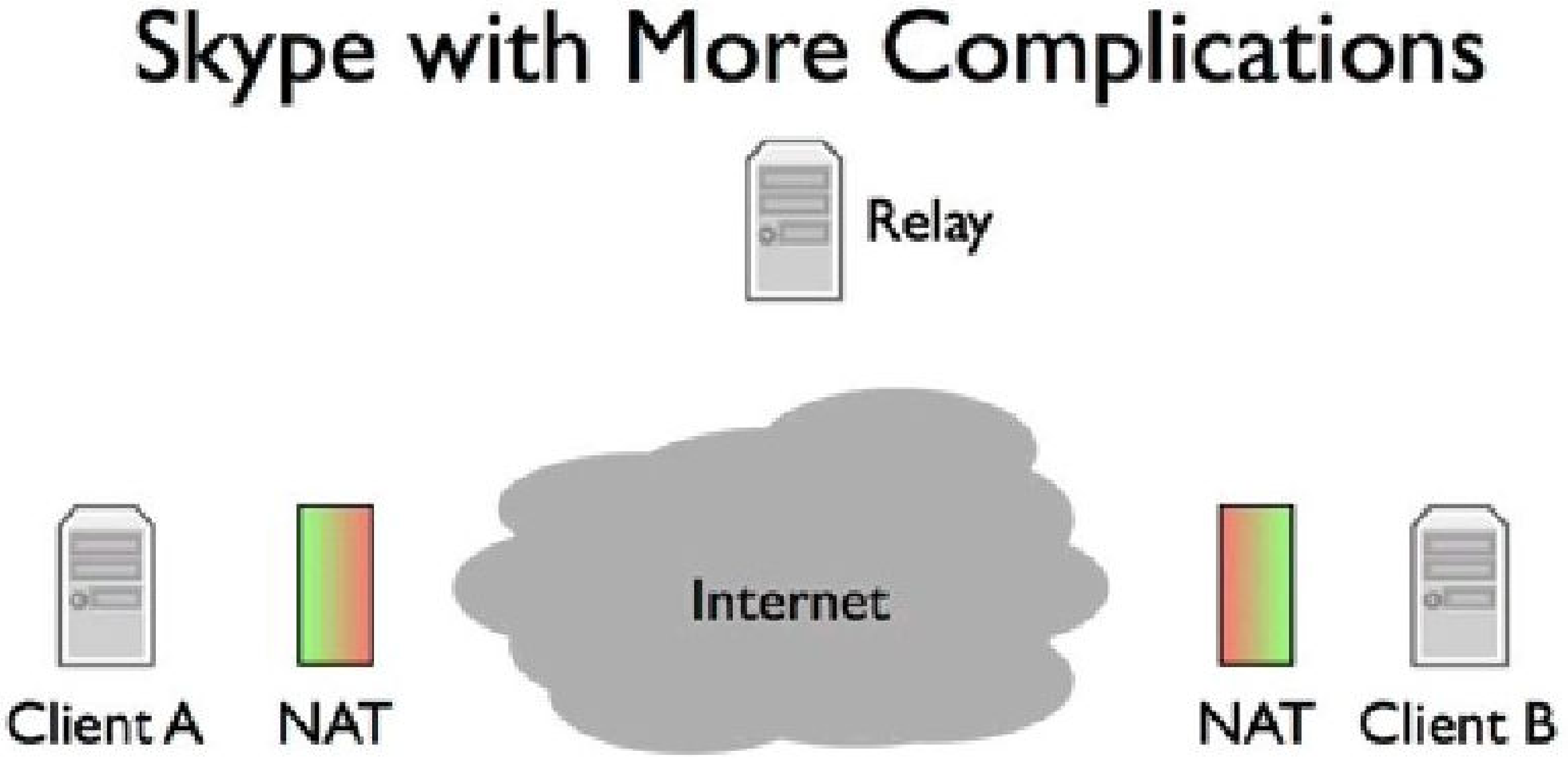


A day in the life of an application

Skype with More Complications



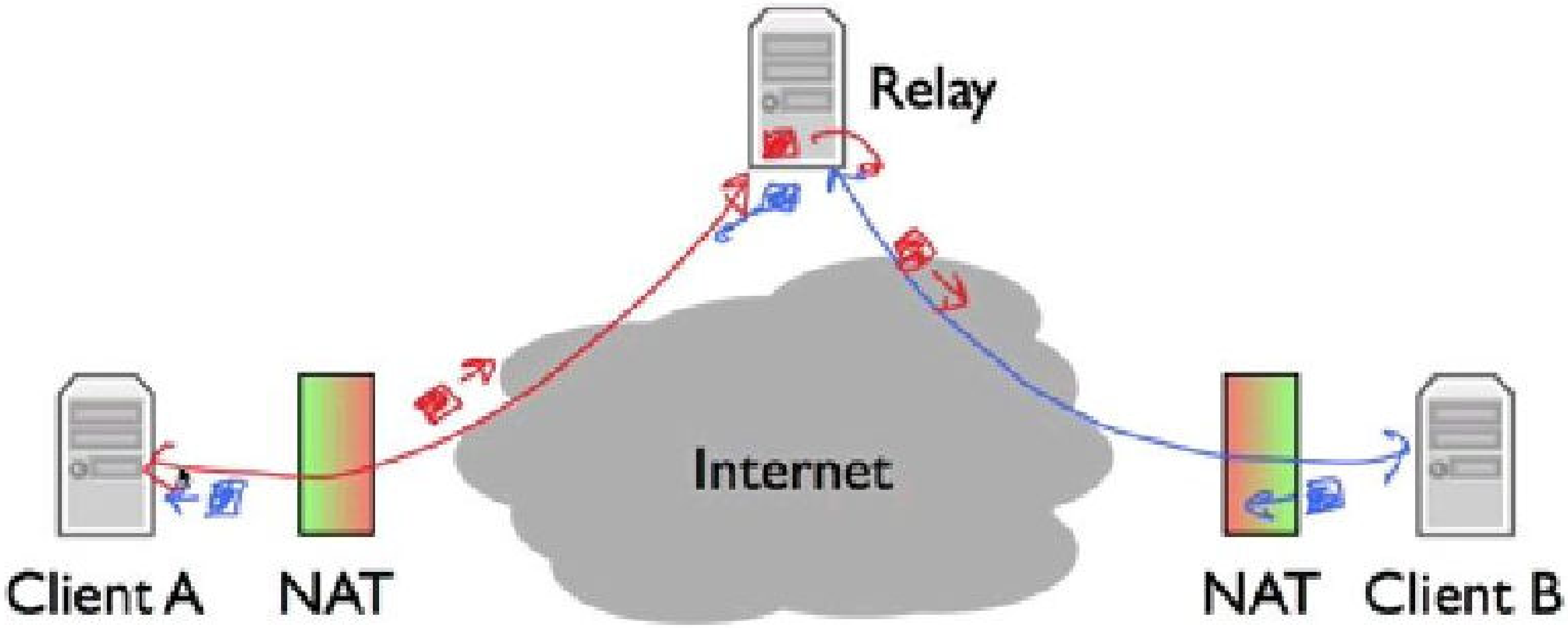
A day in the life of an application



Skype with More Complications

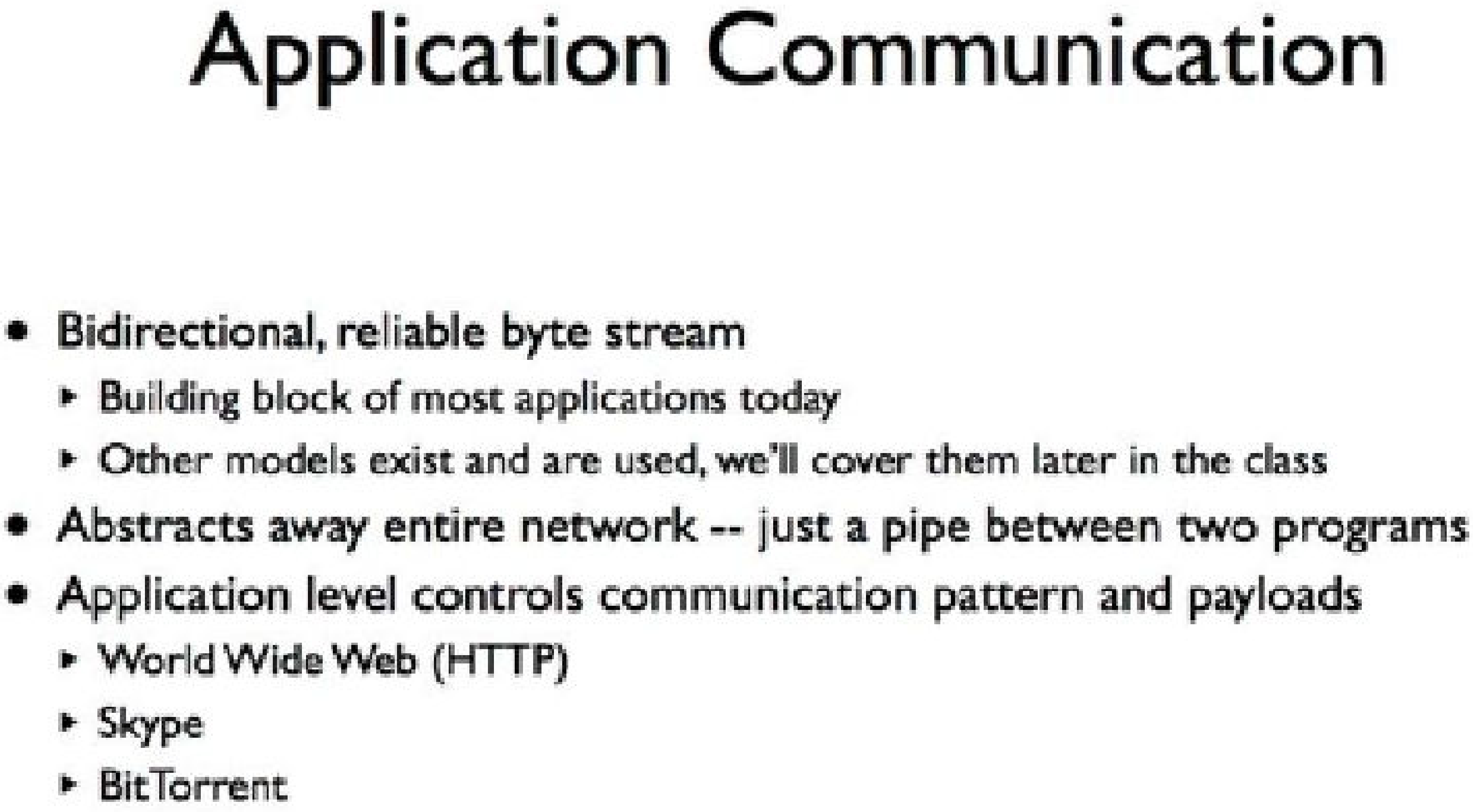
A day in the life of an application

Skype with More Complications



A day in the life of an application

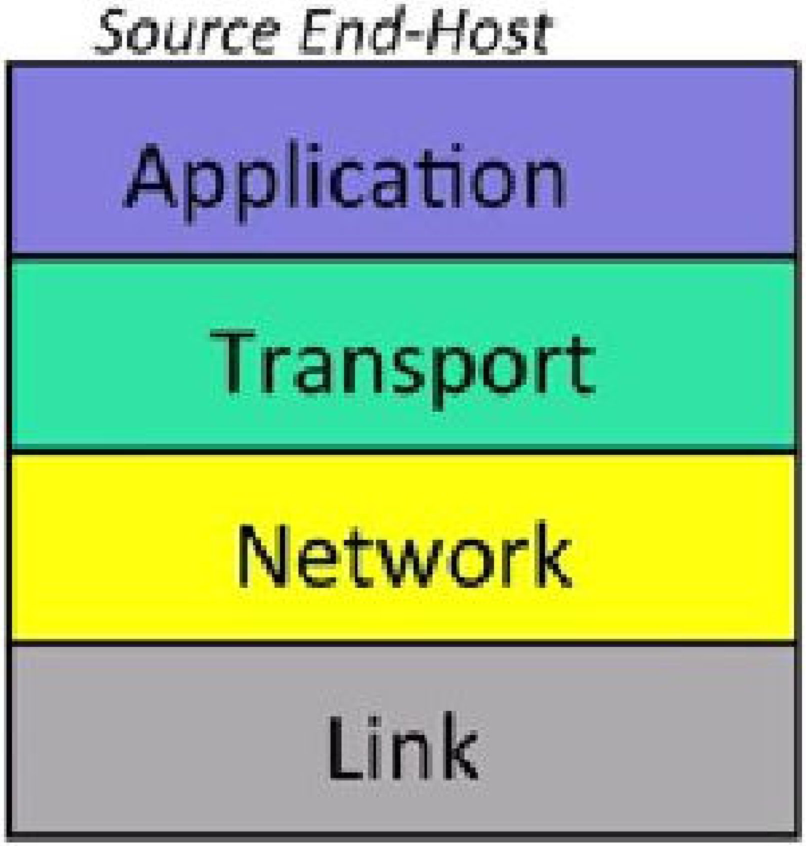
Summary:



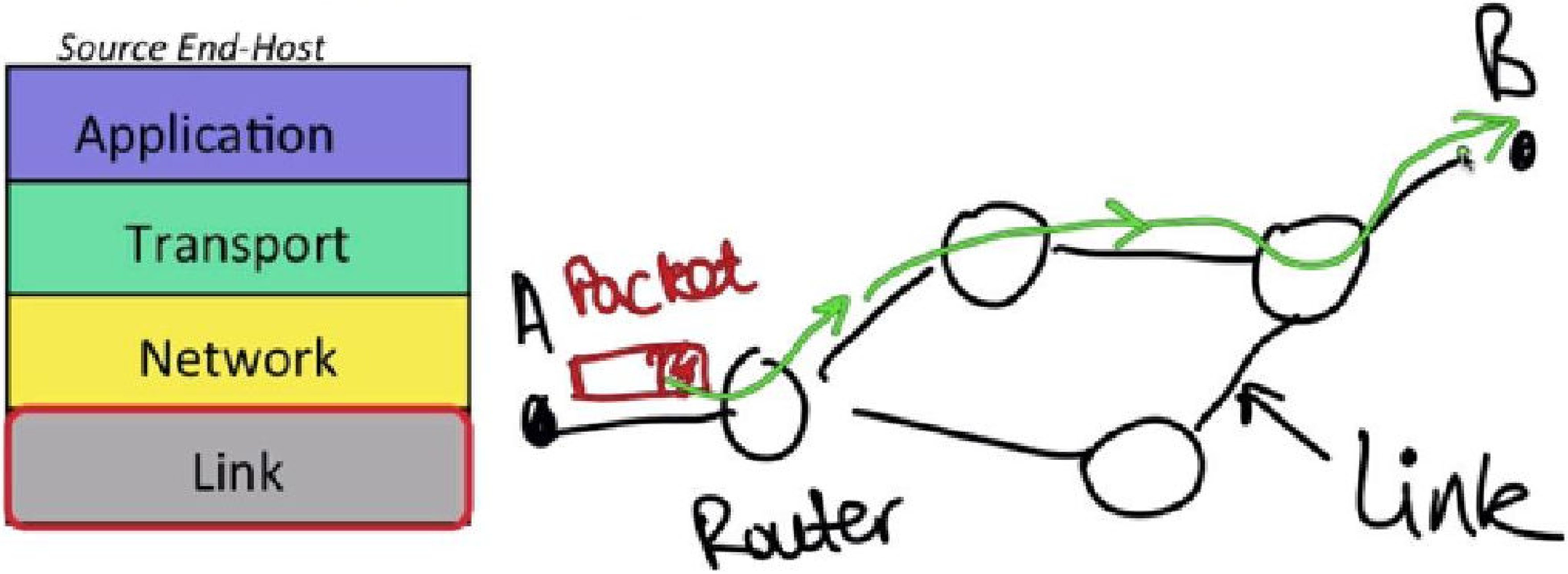
Application Communication

- Bidirectional, reliable byte stream
  - Building block of most applications today
  - Other models exist and are used, we'll cover them later in the class
- Abstracts away entire network -- just a pipe between two programs
- Application level controls communication pattern and payloads
  - World Wide Web (HTTP)
  - Skype
  - BitTorrent

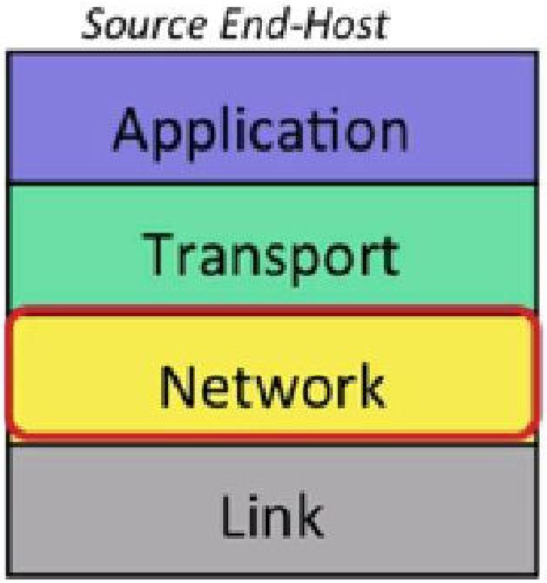
The four layer Internet model

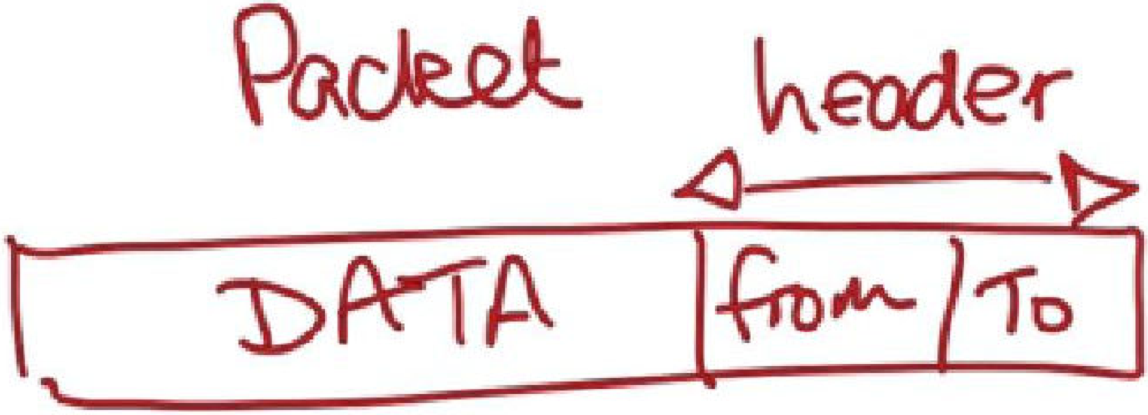


The four layer Internet model

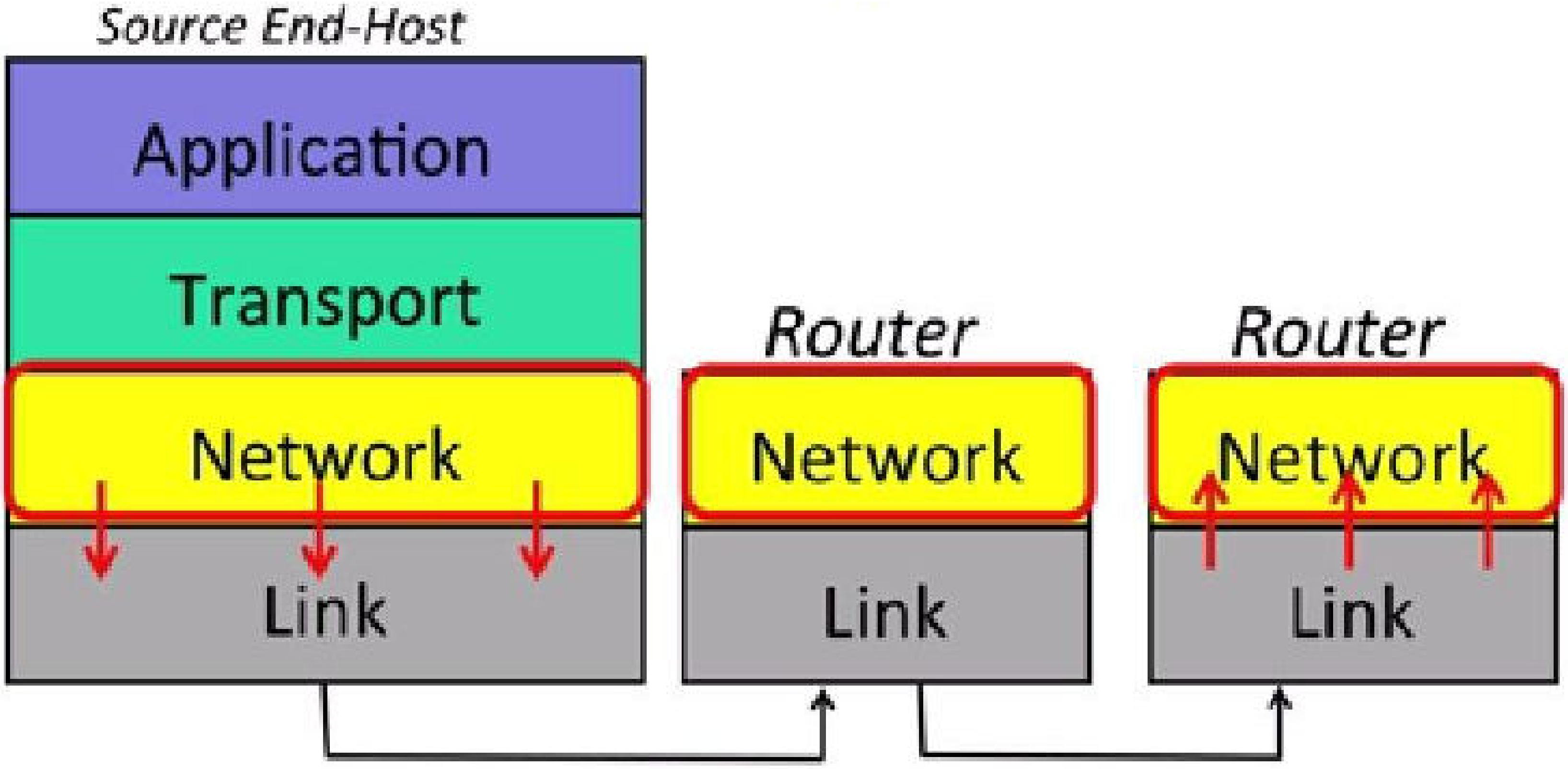


The four layer Internet model

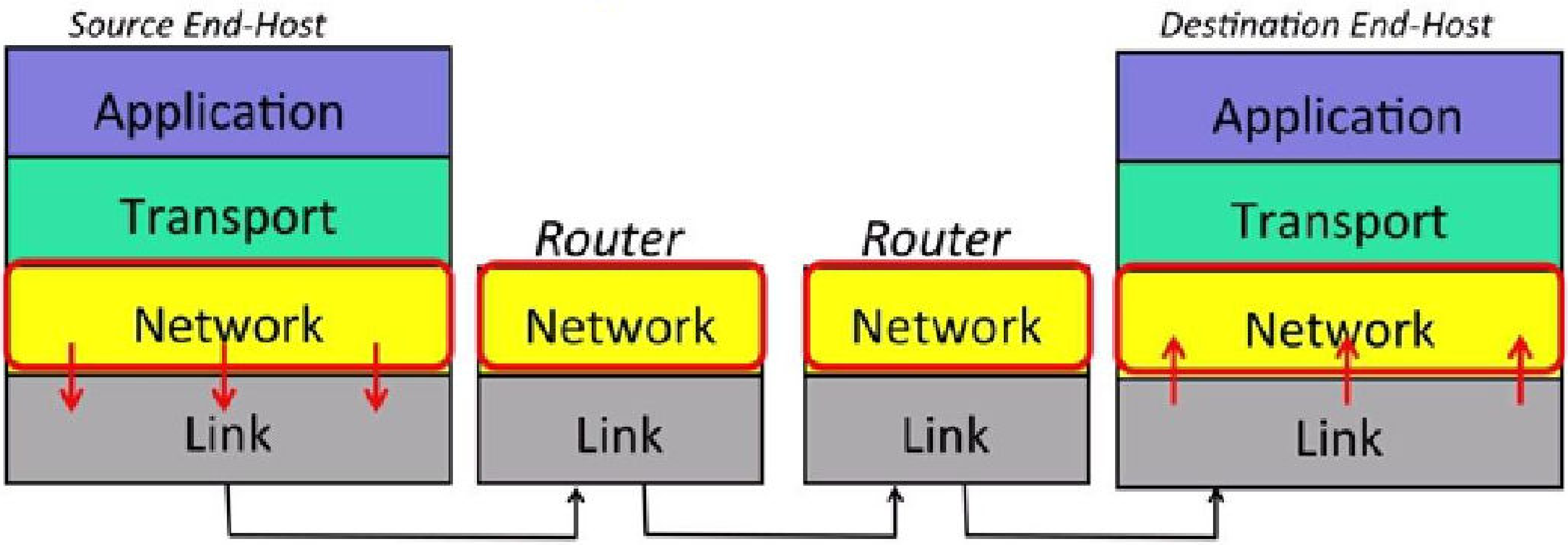




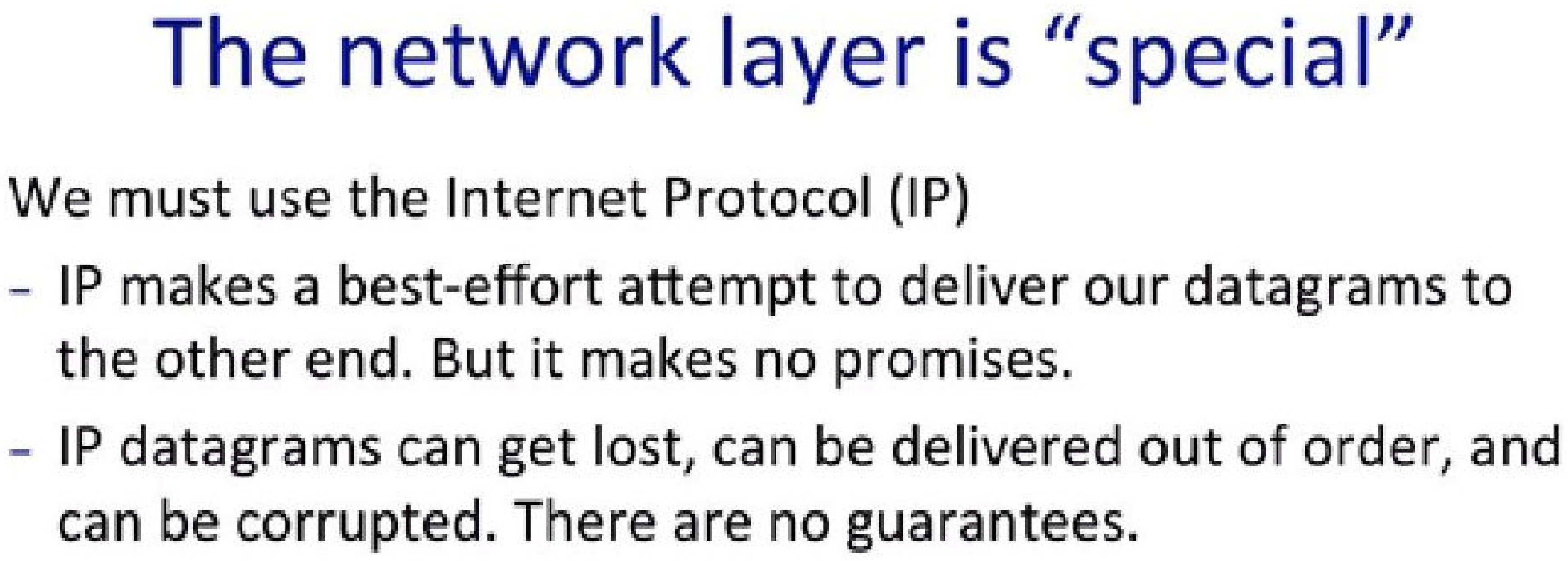
The four layer Internet model



The four layer Internet model



The four layer Internet model

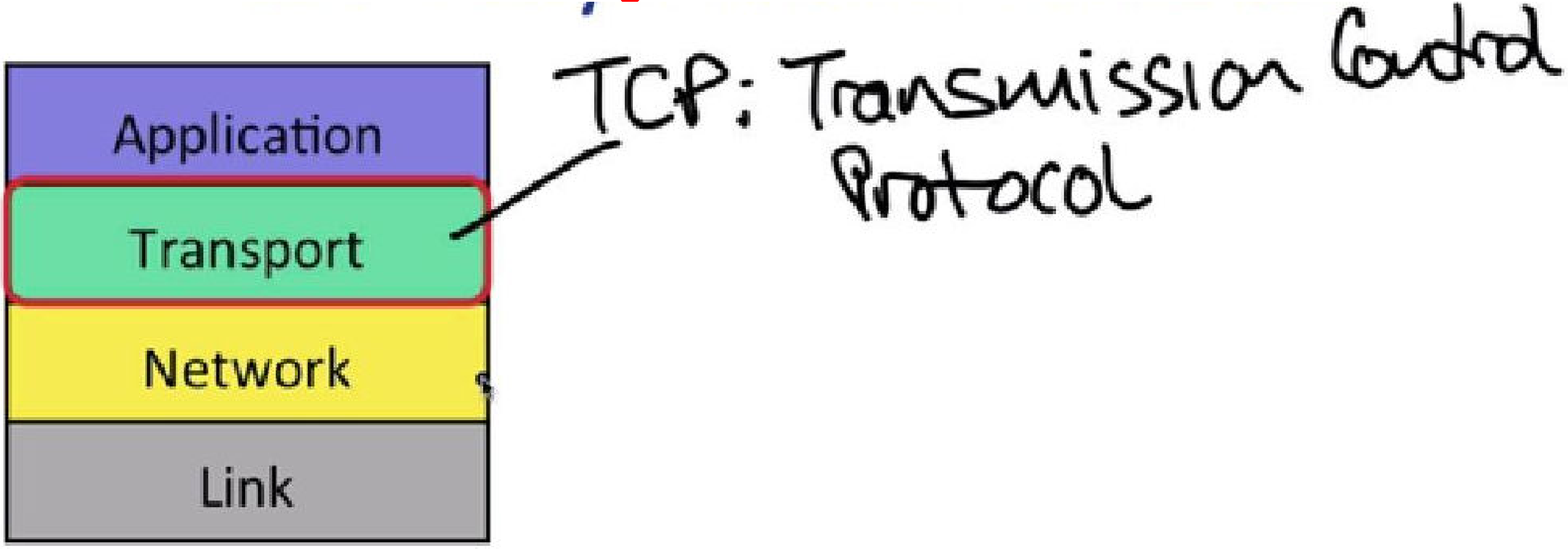


The network layer is "special"

We must use the Internet Protocol (IP)

- IP makes a best-effort attempt to deliver our datagrams to the other end. But it makes no promises.
- IP datagrams can get lost, can be delivered out of order, and can be corrupted. There are no guarantees.

The four layer Internet model



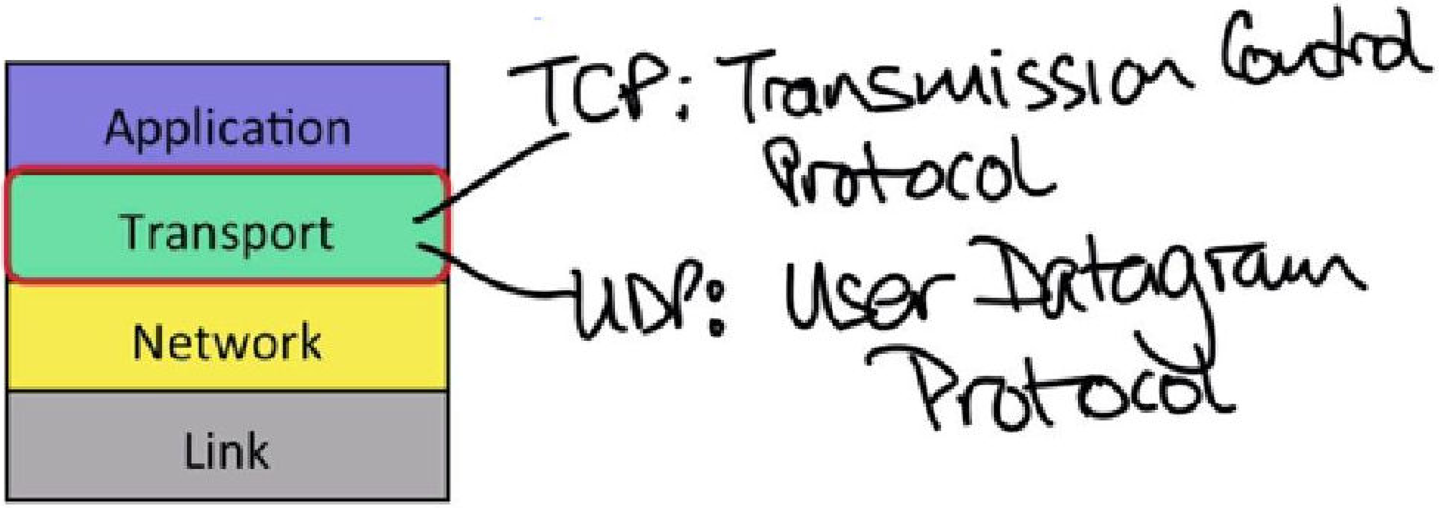
Application

Transport

Network

Link

TCP: Transmission Control Protocol



Application

Transport

Network

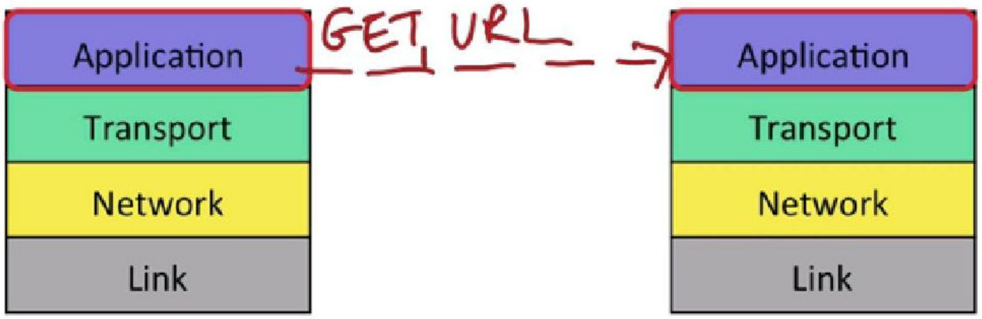
Link

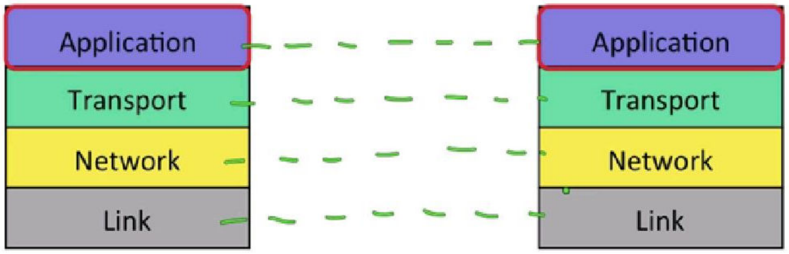
TCP: Transmission Control Protocol

UDP: User Datagram Protocol

The four layer Internet model

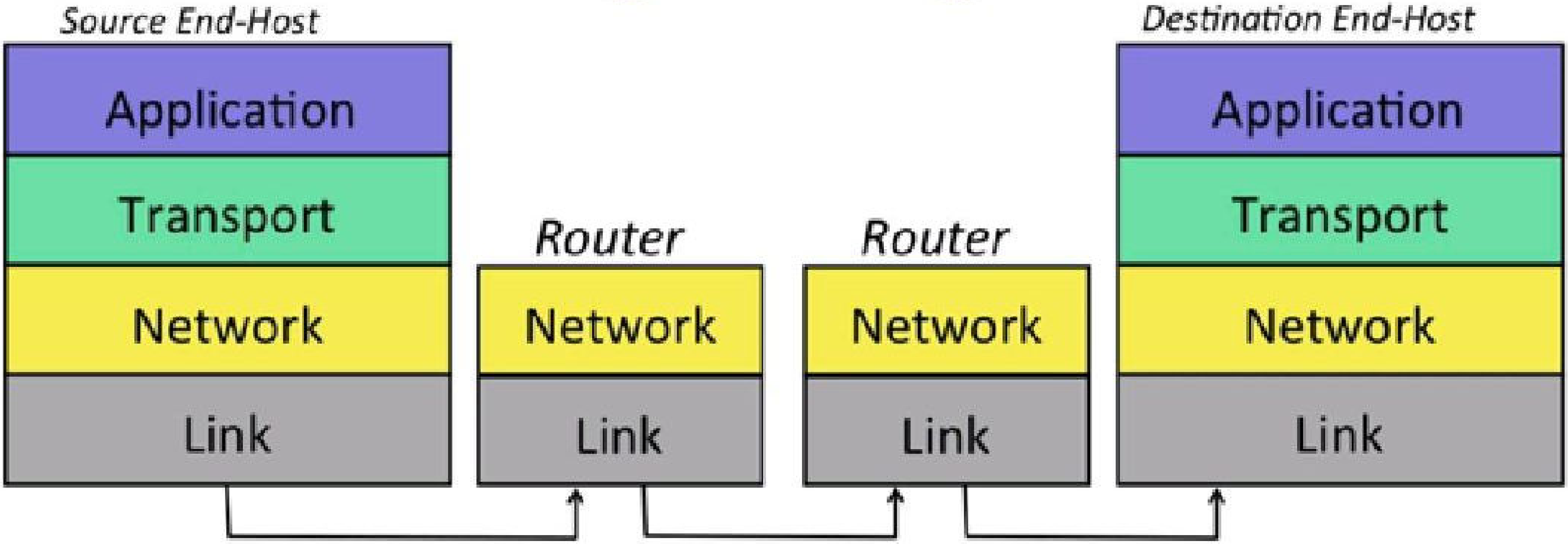
Application
Transport
Network
Link
Application
Transport
Network
Link



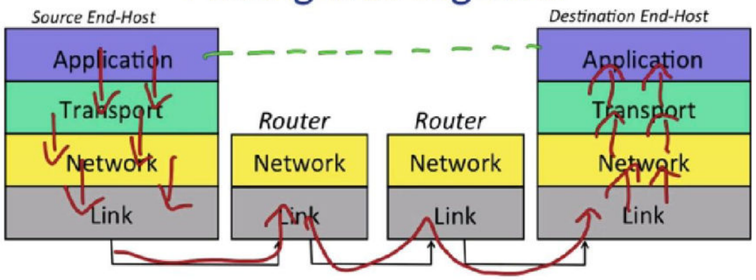


The four layer Internet model

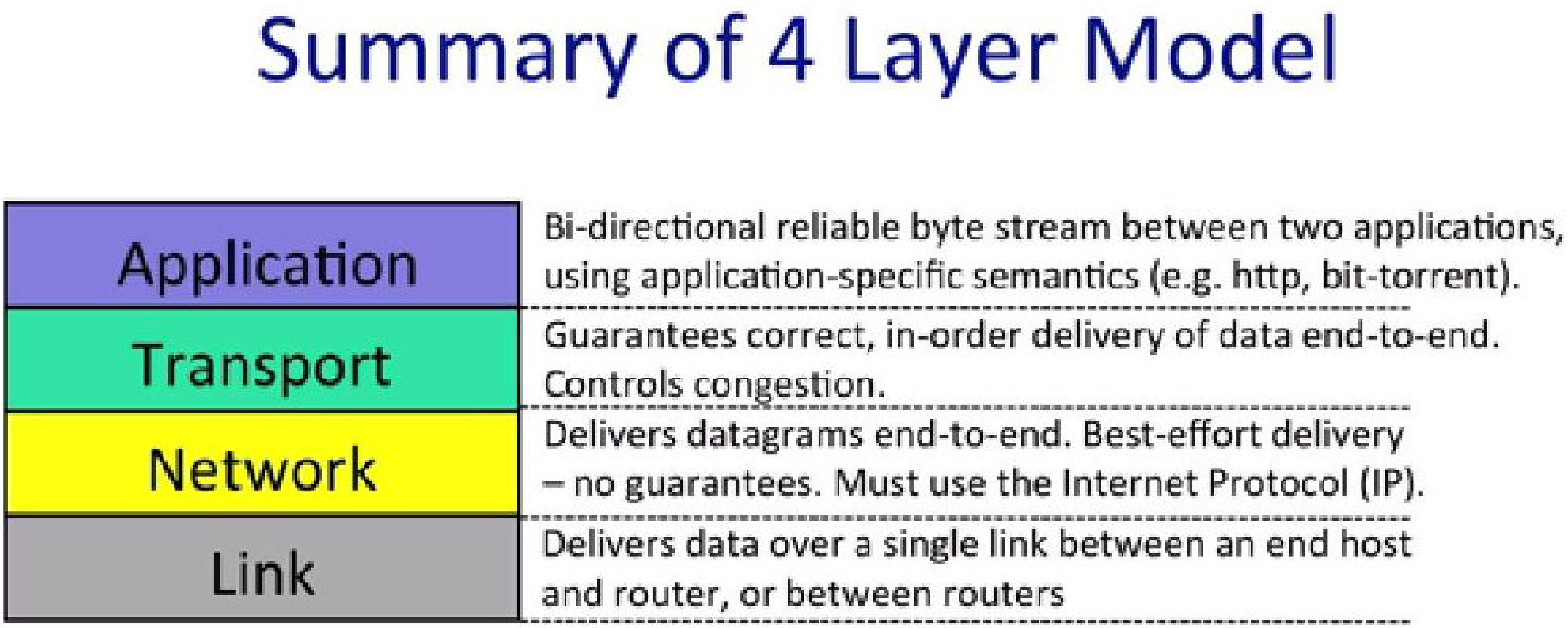
Putting it all together



Putting it all together



The four layer Internet model



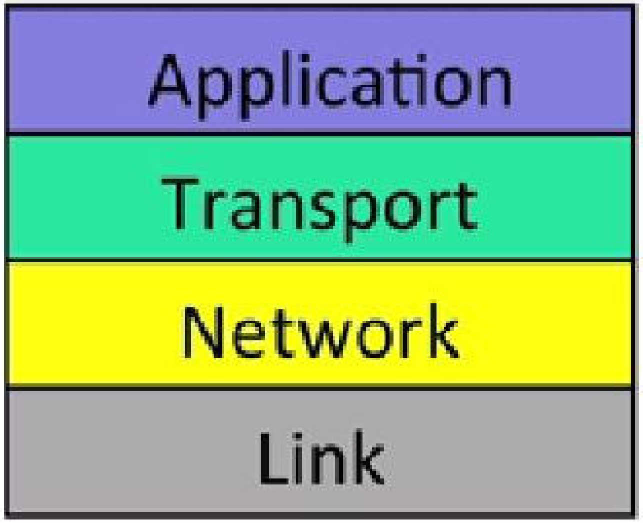
Summary of 4 Layer Model

Application	Bi-directional reliable byte stream between two applications, using application-specific semantics (e.g. http, bit-torrent).
Transport	Guarantees correct, in-order delivery of data end-to-end. Controls congestion.
Network	Delivers datagrams end-to-end. Best-effort delivery – no guarantees. Must use the Internet Protocol (IP).
Link	Delivers data over a single link between an end host and router, or between routers

The four layer Internet model

Two extra things you need to know…

IP is the “thin waist”



Application

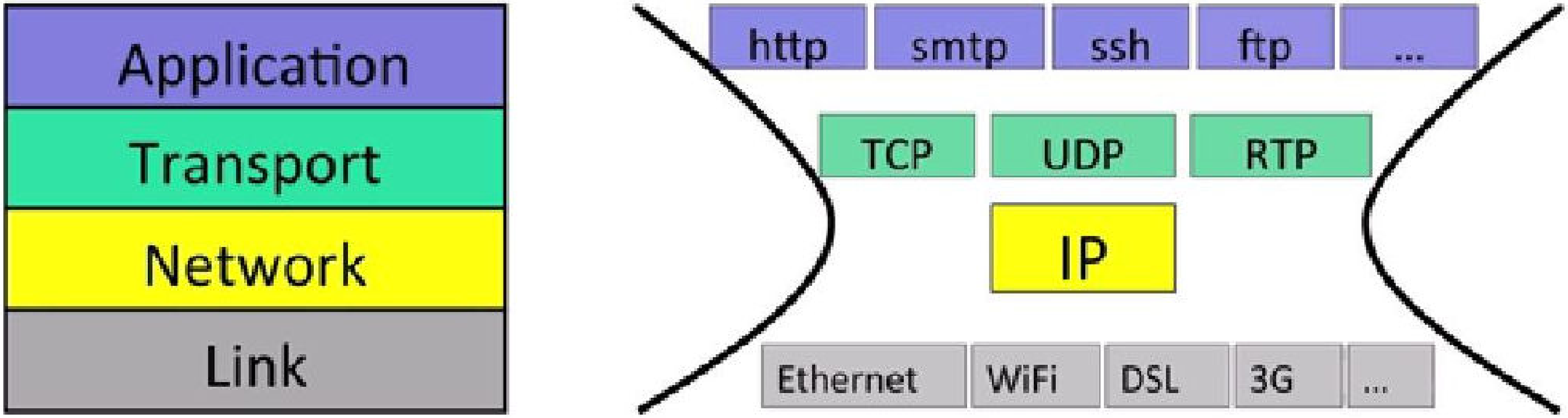
Transport

Network

Link

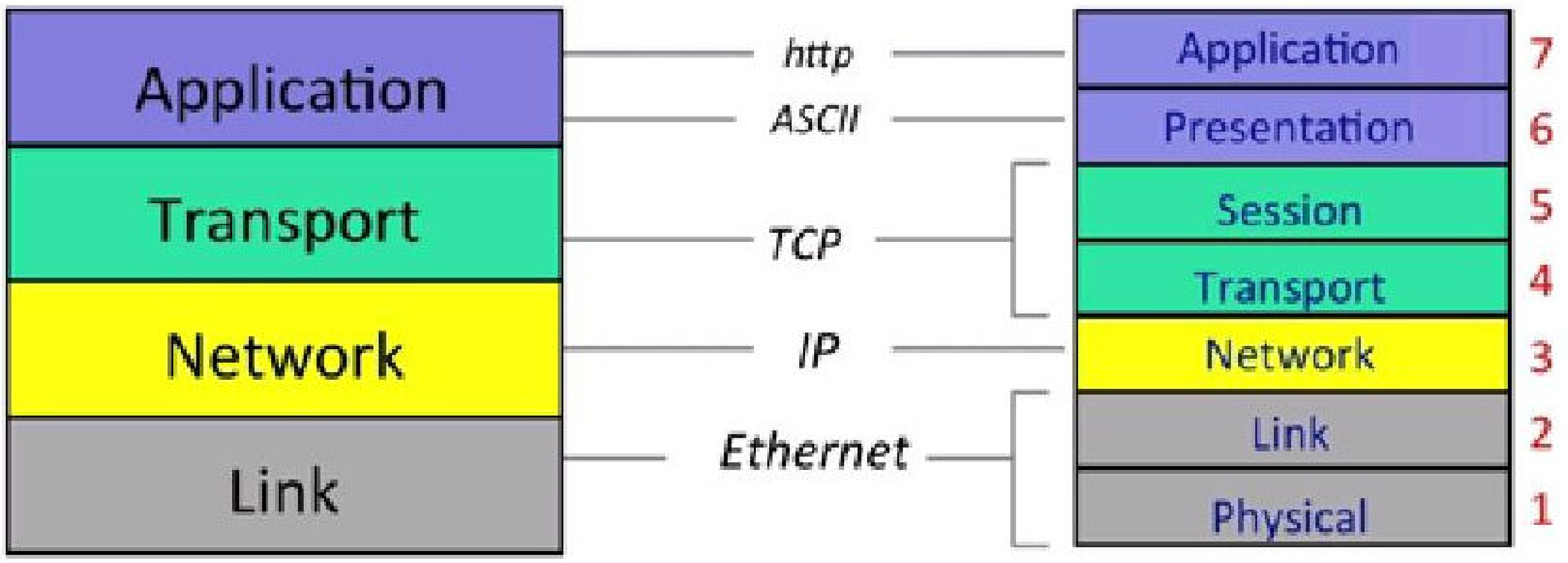
The four layer Internet model

IP is the “thin waist”



The four layer Internet model

The 7-layer OSI Model



The 7-layer OSI Model

Link layer Services: next

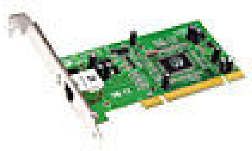
our goals:

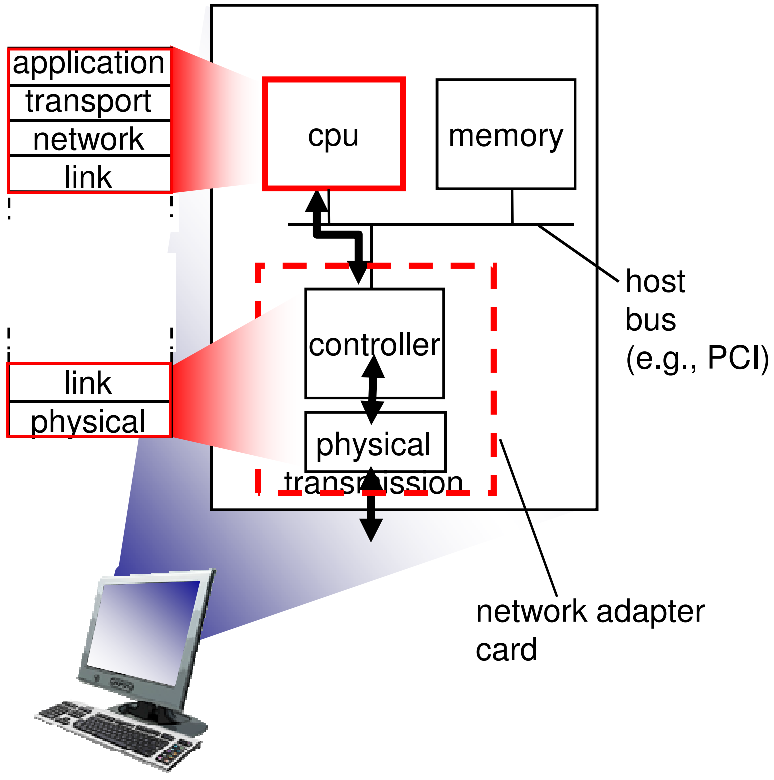
- understand principles behind link layer services:
  - **Framing**
  - error detection, correction
  - sharing a broadcast channel: multiple access
  - link layer addressing
  - local area networks: Ethernet, VLANs

Where is the link layer implemented?

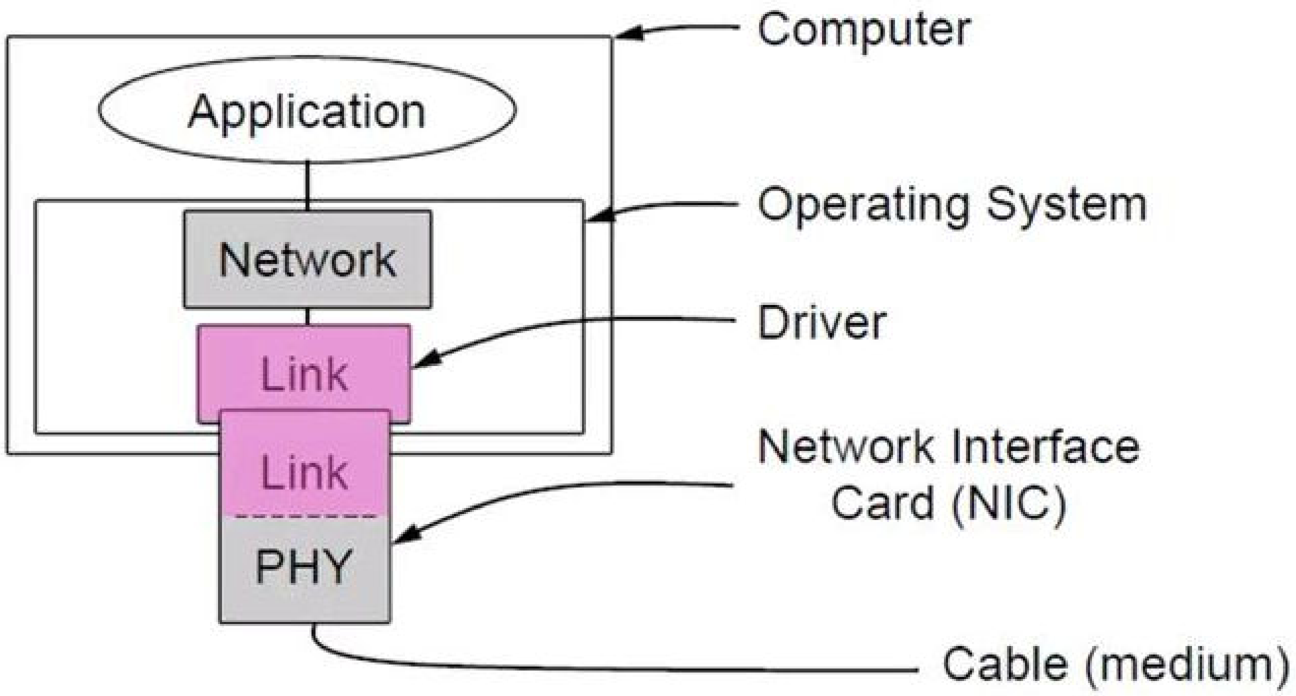
- **in each and every host**
- **link layer implemented in:** “**adaptor**” (aka *network interface card* **NIC**) or on a chip
  - **Ethernet card, 802.11 card; Ethernet chipset implements:**
    - **both link and physical layers**
- **attaches into host’ s system buses**

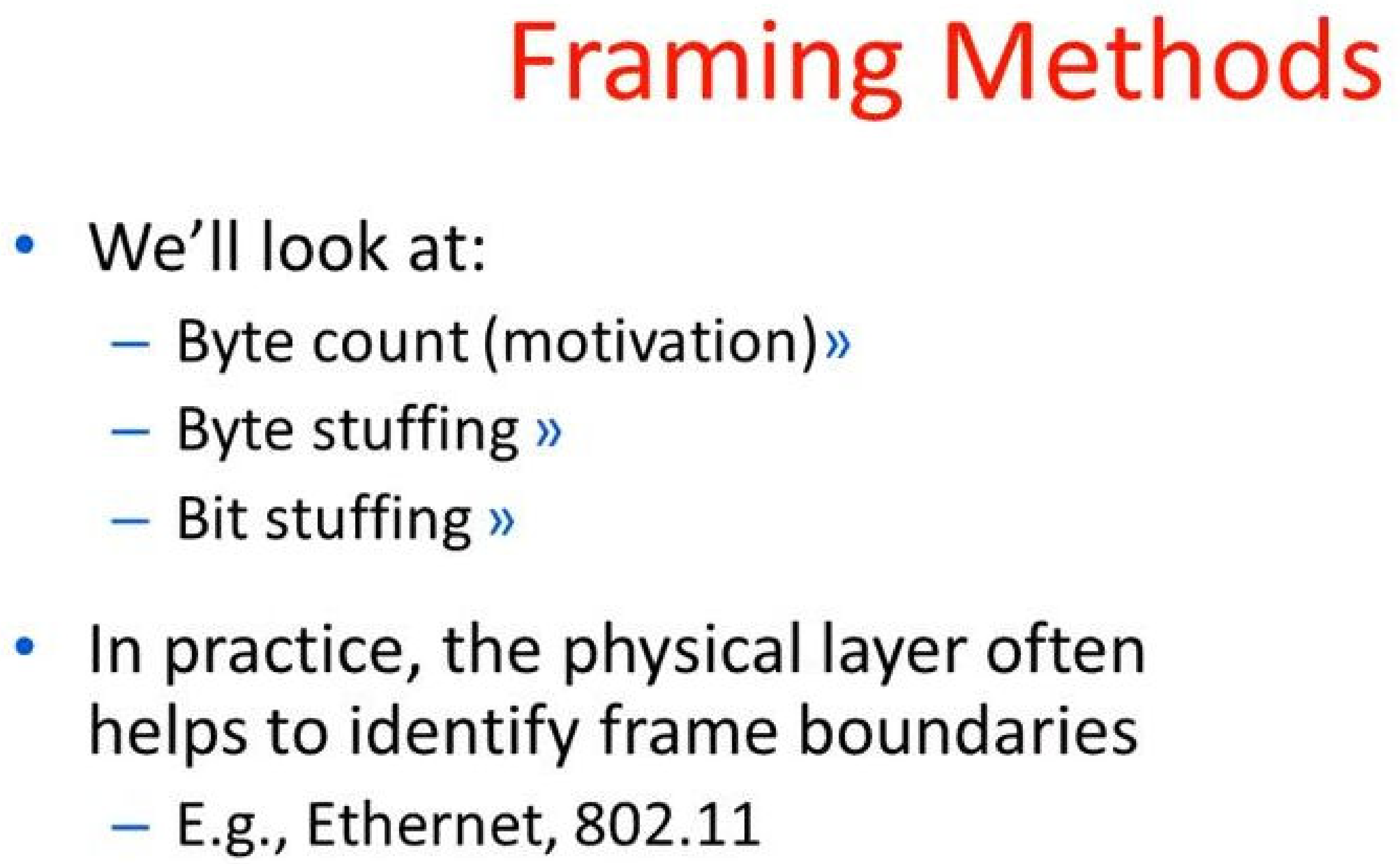
combination of hardware, software, firmware





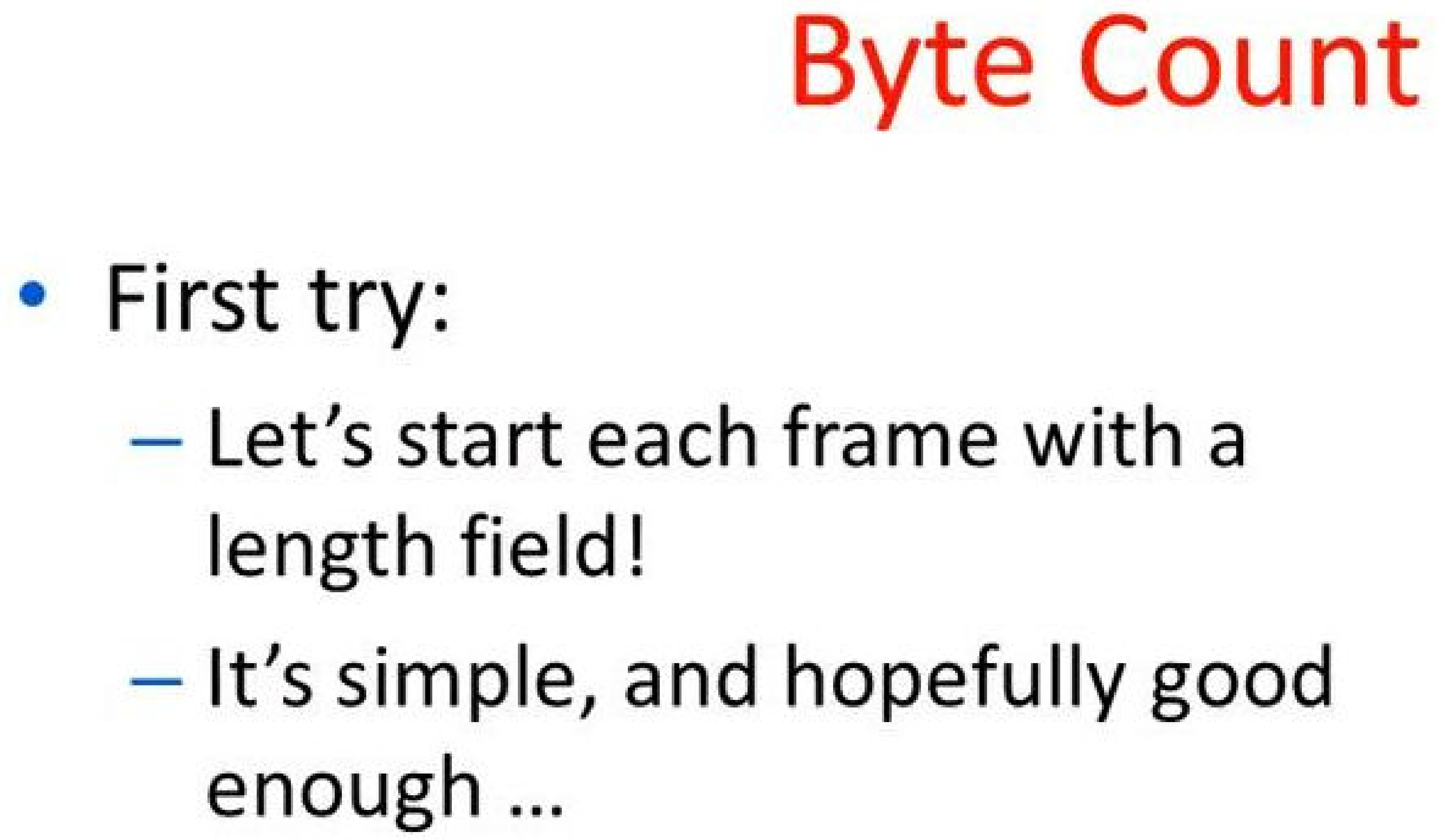
Typical Implementation of Layers (2)





Framing Methods

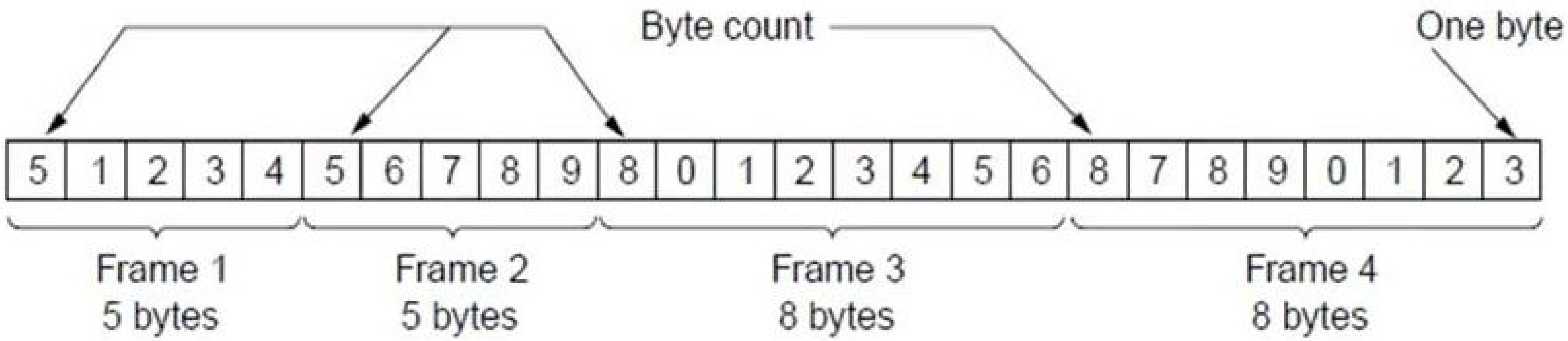
- We'll look at:
  - Byte count (motivation)»
  - Byte stuffing »
  - Bit stuffing »
- In practice, the physical layer often helps to identify frame boundaries
  - E.g., Ethernet, 802.11



Byte Count

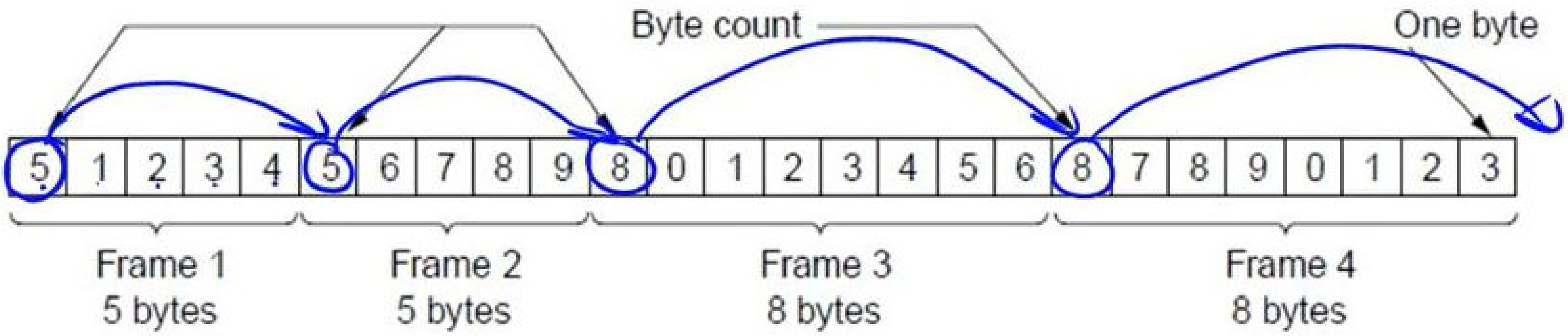
- First try:
  - Let’s start each frame with a length field!
  - It’s simple, and hopefully good enough …

Byte Count (2)



- How well do you think it works?

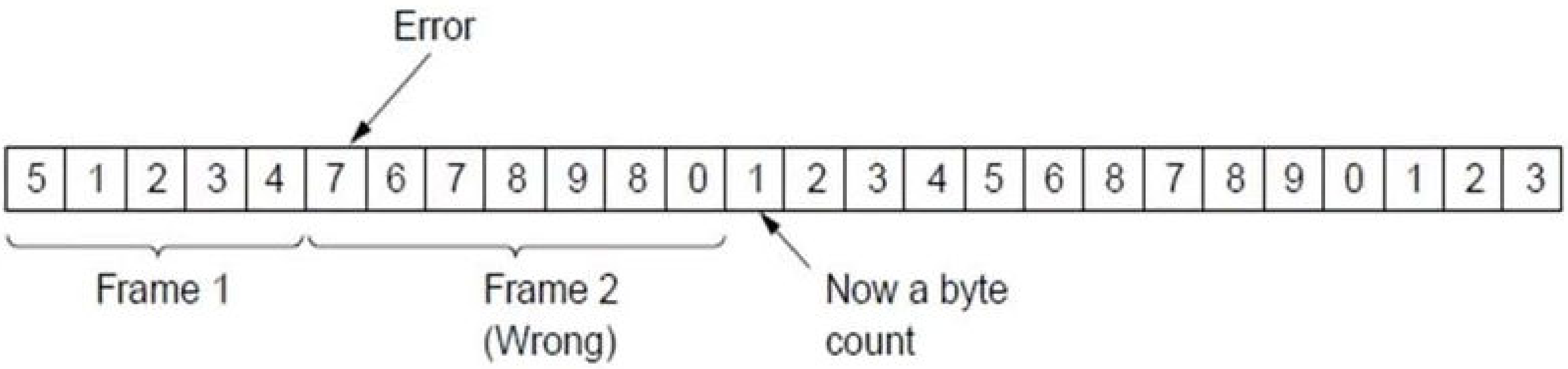
Byte Count (2)



- How well do you think it works?

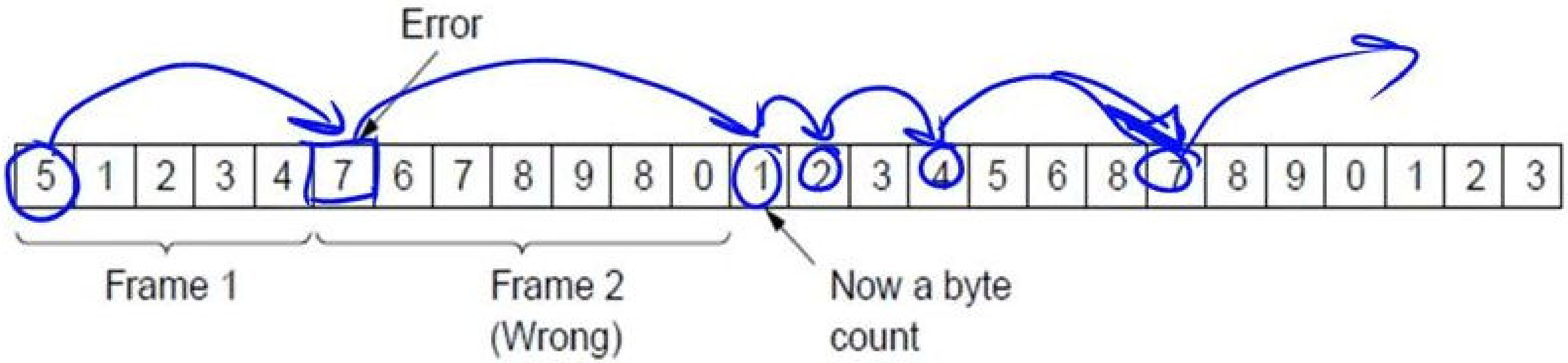
Byte Count (3)

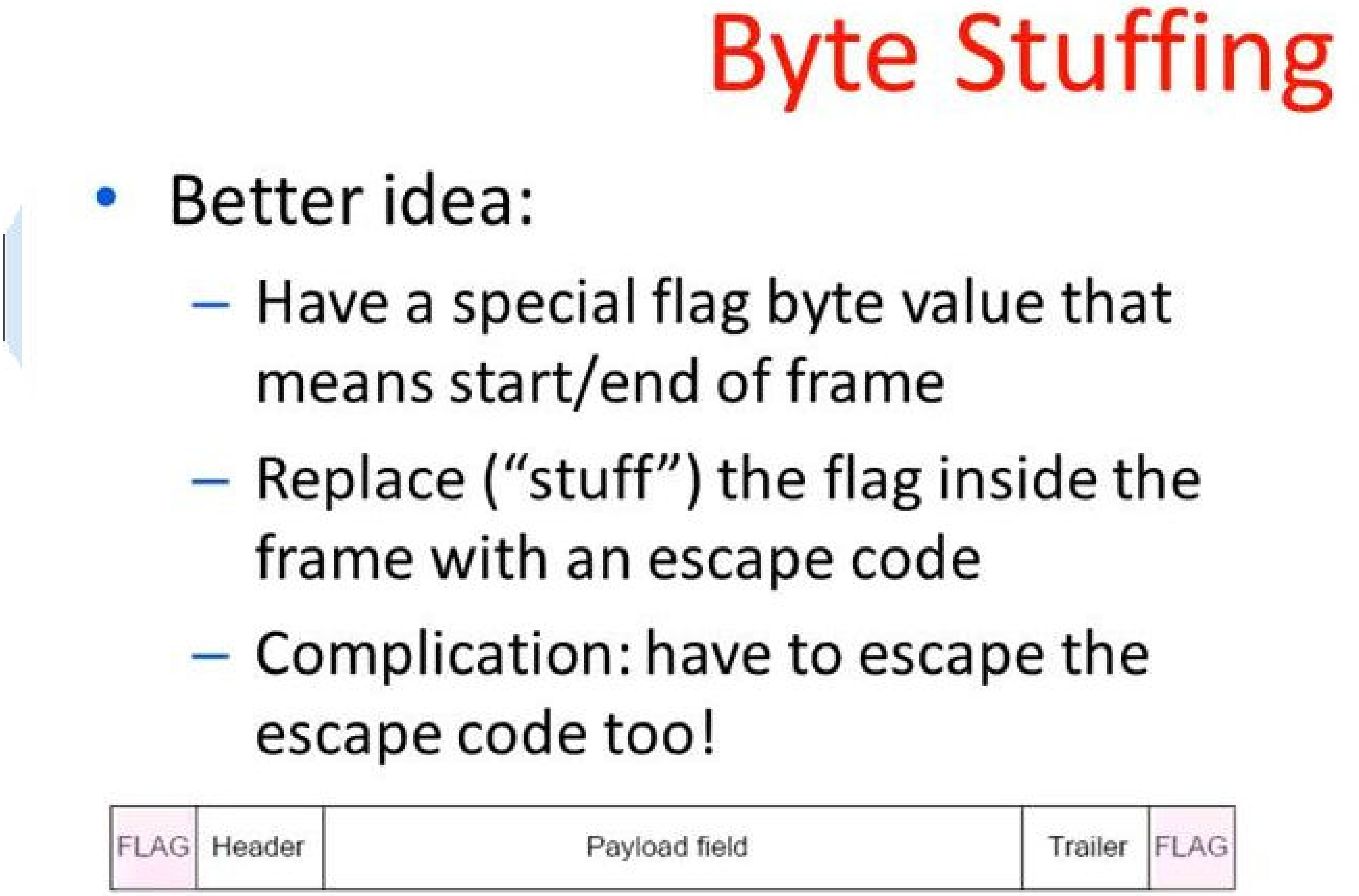
- Difficult to re-synchronize after framing error
  - Want a way to scan for a start of frame



Byte Count (3)

- Difficult to re-synchronize after framing error
  - Want a way to scan for a start of frame





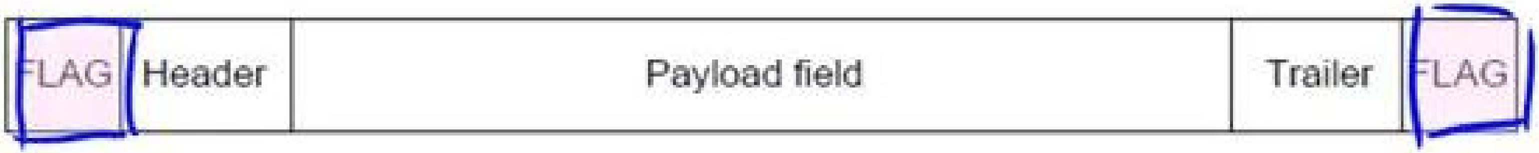
Byte Stuffing

- Better idea:
  - Have a special flag byte value that means start/end of frame
  - Replace ("stuff") the flag inside the frame with an escape code
  - Complication: have to escape the escape code too!

FLAG	Header	Payload field	Trailer	FLAG

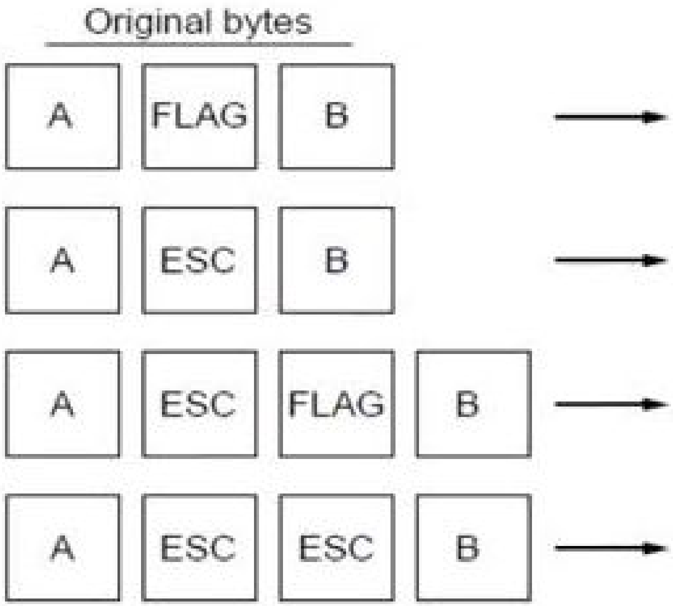
Byte Stuffing

- Better idea:
  - Have a special flag byte value that means start/end of frame
  - Replace (“stuff”) the flag inside the frame with an escape code
  - Complication: have to escape the escape code too!



Byte Stuffing (2)

- Rules:
  - Replace each FLAG in data with ESC FLAG
  - Replace each ESC in data with ESC ESC



Original bytes

A FLAG B →

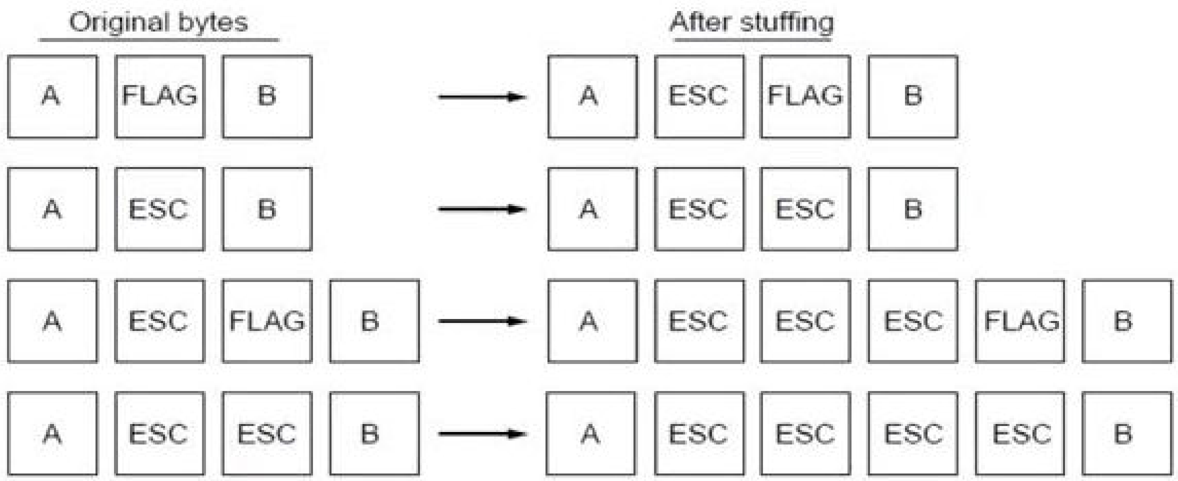
A ESC B →

A ESC FLAG B →

A ESC ESC B →

Byte Stuffing (3)

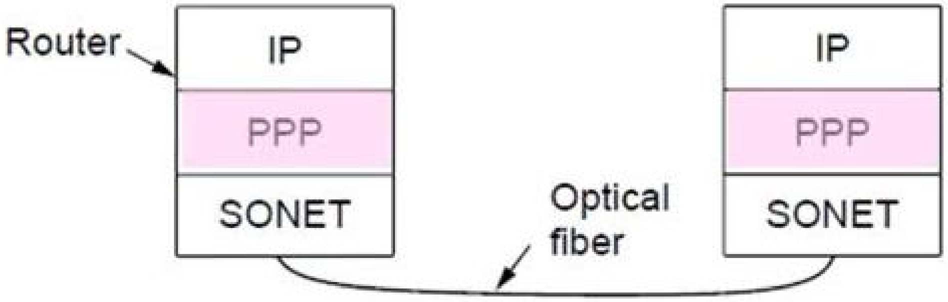
- Now any unescaped FLAG is the start/end of a frame



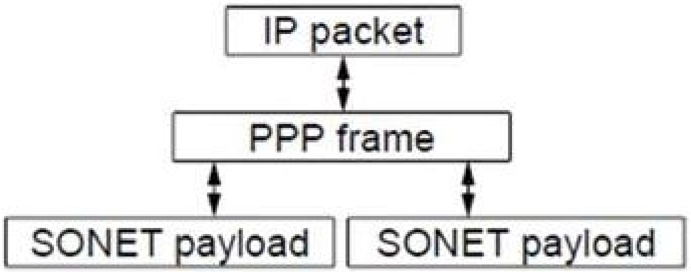
Original bytes		After stuffing
A FLAG B	→	A ESC FLAG B
A ESC B	→	A ESC ESC B
A ESC FLAG B	→	A ESC ESC ESC FLAG B
A ESC ESC B	→	A ESC ESC ESC ESC B

Link Example: PPP over SONET (2)

- Think of SONET as a bit stream, and PPP as the framing that carries an IP packet over the link



Protocol stacks



PPP frames may be split over SONET payloads

Link Example: PPP over SONET (3)

- Framing uses byte stuffing
  - FLAG is 0x7E and ESC is 0x7D

Bytes	1	1	1	1 or 2	Variable	2 or 4	1
	Flag 01111110	Address 11111111	Control 00000011	Protocol	Payload	Checksum	Flag 01111110

Link Example: PPP over SONET (4)

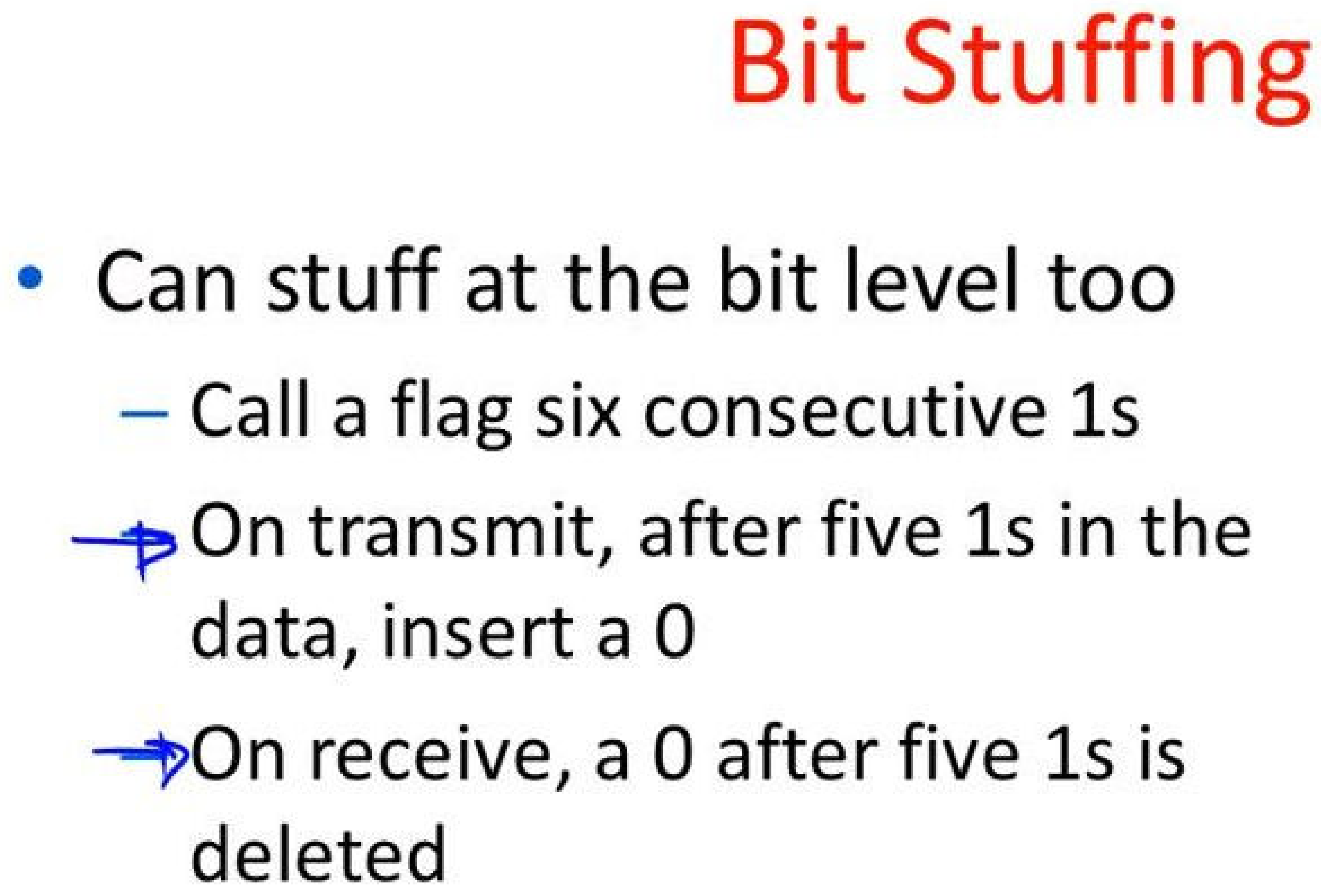
- Byte stuffing method:
  - To stuff (unstuff) a byte, add (remove) ESC (0x7D), and XOR byte with 0x20
  - Removes FLAG from the contents of the frame

Link Example: PPP over SONET (4)

- Byte stuffing method:
  - To stuff (unstuff) a byte, add (remove) ESC (0x7D), and XOR byte with 0x20 ~ toggle 5th bit
  - Removes FLAG from the contents of the frame

0x7E → 0x7D5E

0x7D → 0x7D5D



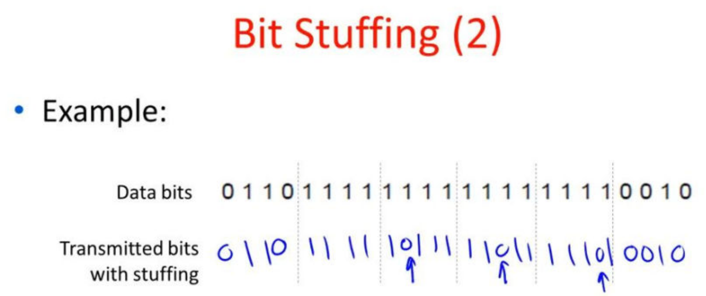
Bit Stuffing

- Can stuff at the bit level too
  - Call a flag six consecutive 1s
  - → On transmit, after five 1s in the data, insert a 0
  - → On receive, a 0 after five 1s is deleted

Bit Stuffing (2)

- Example:

Data bits	0 1 1 0	1 1 1 1	1 1 1 1	1 1 1 1	1 1 1 1	0 0 1 0
Transmitted bits with stuffing						



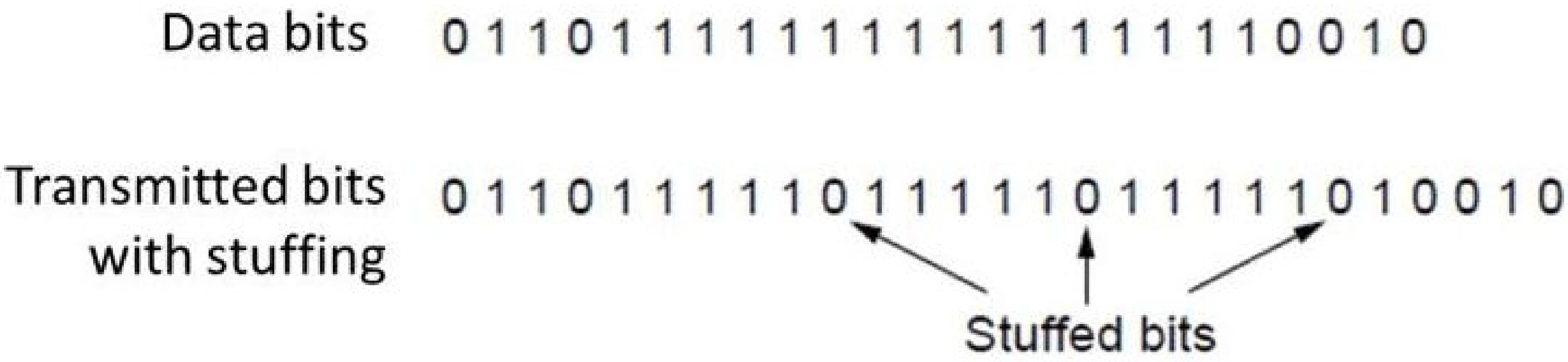
Bit Stuffing (2)

- Example:

Data bits	0 1 1 0	1 1 1 1	1 1 1 1	1 1 1 1	1 1 1 1	0 0 1 0
Transmitted bits with stuffing	0 1 1 0	1 1 1 1	1 0 1 1 1	1 1 0 1 1	1 1 1 0 1	0 0 1 0

Bit Stuffing (3)

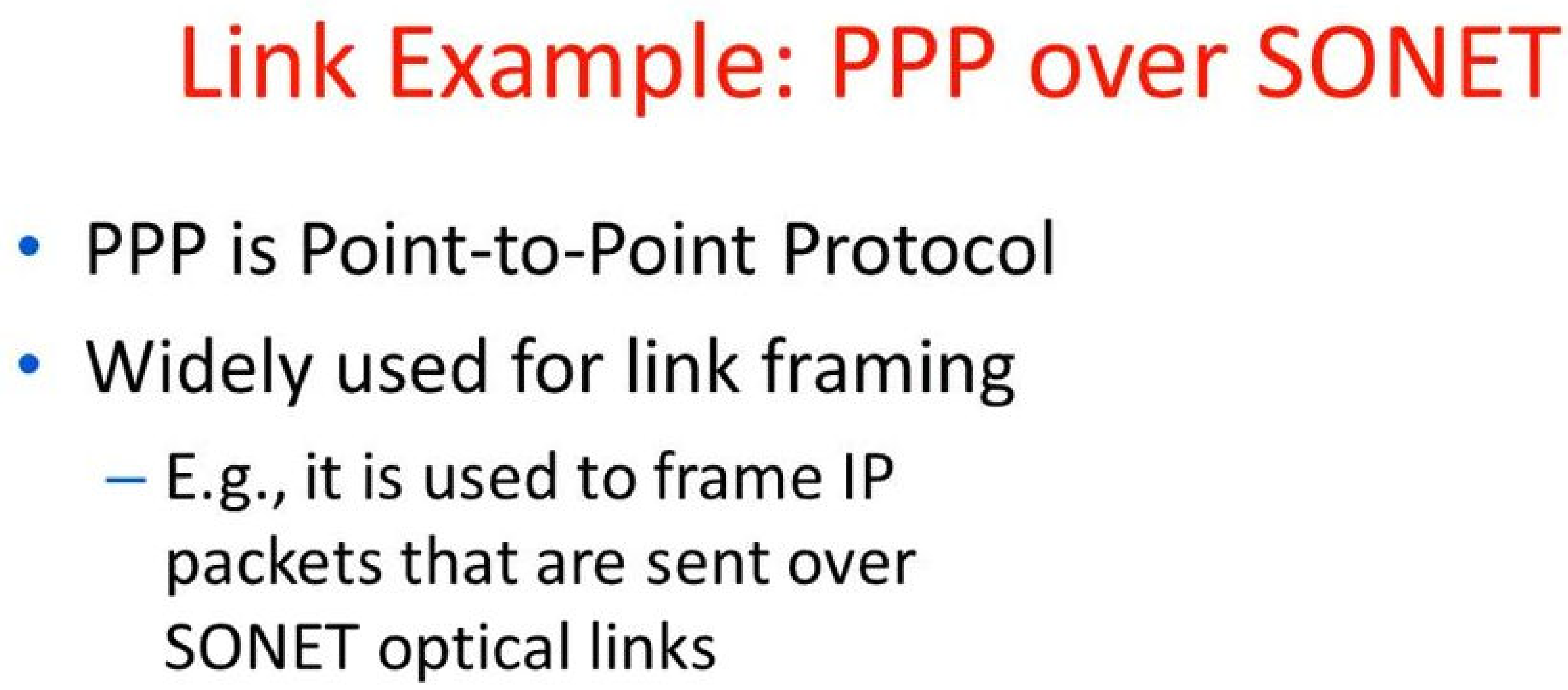
- So how does it compare with byte stuffing?



Data bits 0 1 1 0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 0 0 1 0

Transmitted bits with stuffing 0 1 1 0 1 1 1 1 1 0 1 1 1 1 1 0 1 1 1 1 1 0 1 0 0 1 0

Stuffed bits



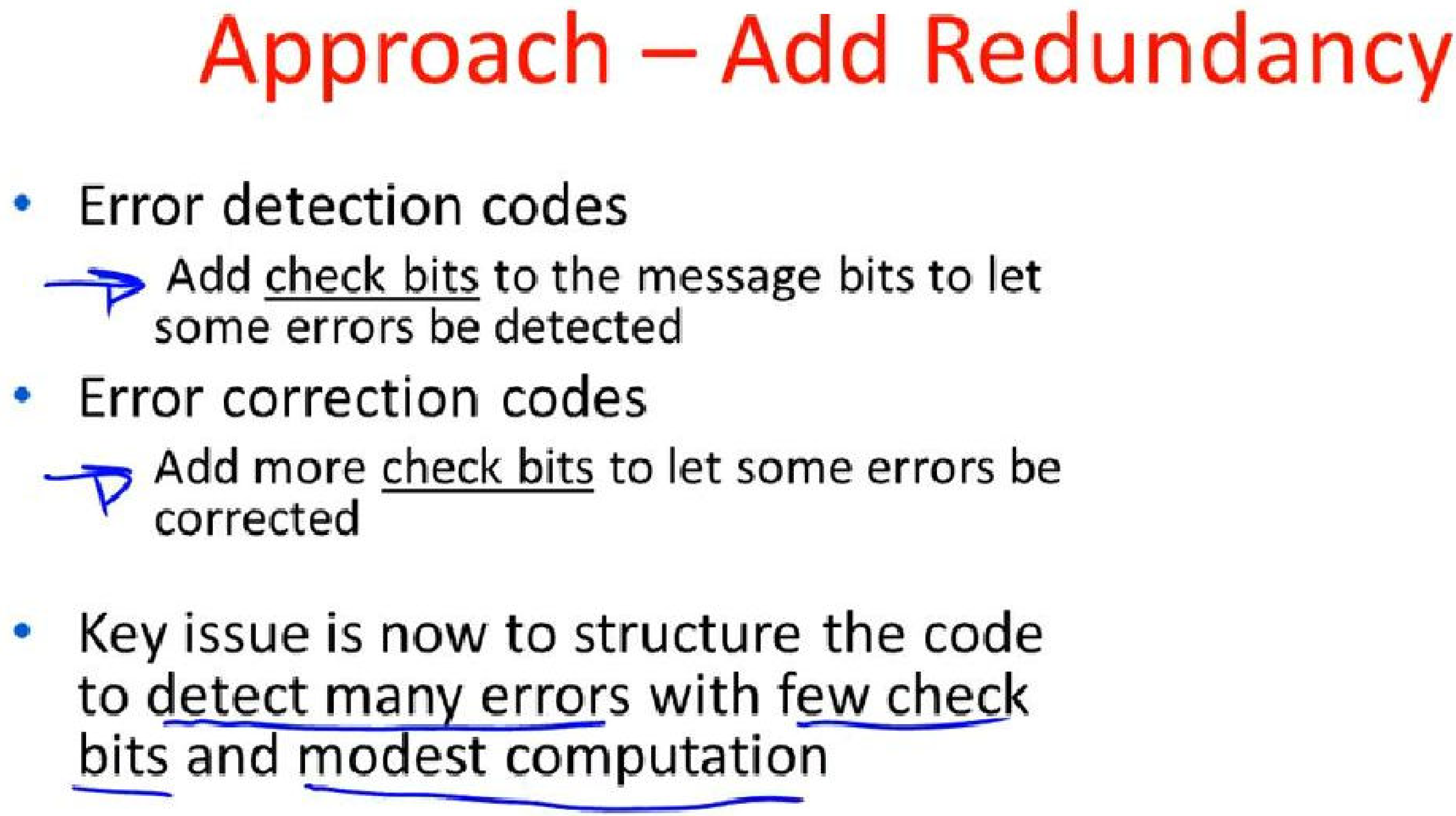
Link Example: PPP over SONET

- PPP is Point-to-Point Protocol
- Widely used for link framing
  - E.g., it is used to frame IP packets that are sent over SONET optical links

Link layer Services: next

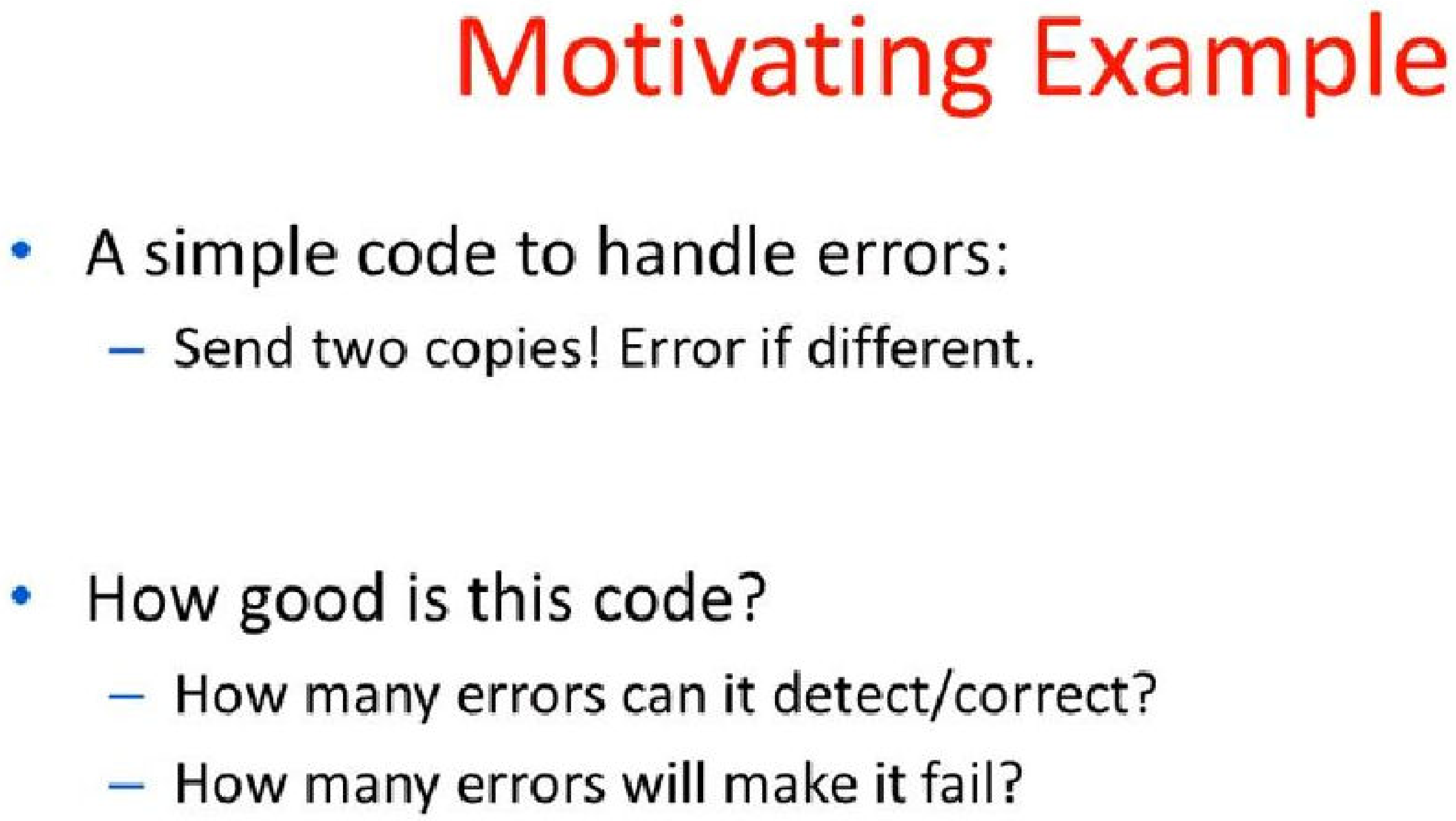
our goals:

- understand principles behind link layer services:
  - Framing
  - **error detection, correction**
  - sharing a broadcast channel: multiple access
  - link layer addressing
  - local area networks: Ethernet, VLANs



Approach – Add Redundancy

- Error detection codes
  - → Add check bits to the message bits to let some errors be detected
- Error correction codes
  - → Add more check bits to let some errors be corrected
- Key issue is now to structure the code to detect many errors with few check bits and modest computation



Motivating Example

- A simple code to handle errors:
  - Send two copies! Error if different.

- How good is this code?
  - How many errors can it detect/correct?
  - How many errors will make it fail?

Motivating Example

- A simple code to handle errors:
  - Send two copies! Error if different.

    010 010

- How good is this code?
  - How many errors can it detect/correct?
  - How many errors will make it fail?

- A simple code to handle errors:
  - Send two copies! Error if different.

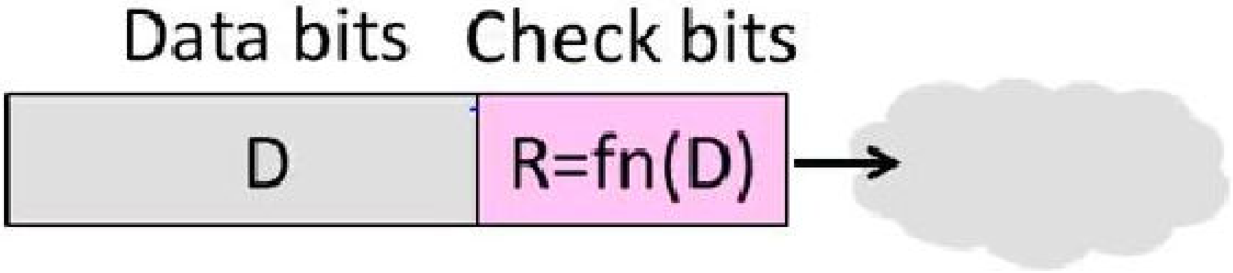
0 1 ~~0~~ : 0 1 ~~0~~

1 1

- How good is this code? 1 0
  - How many errors can it detect/correct?
  - How many errors will make it fail? 2

Using Error Codes

- Codeword consists of D data plus R check bits (=systematic block code)



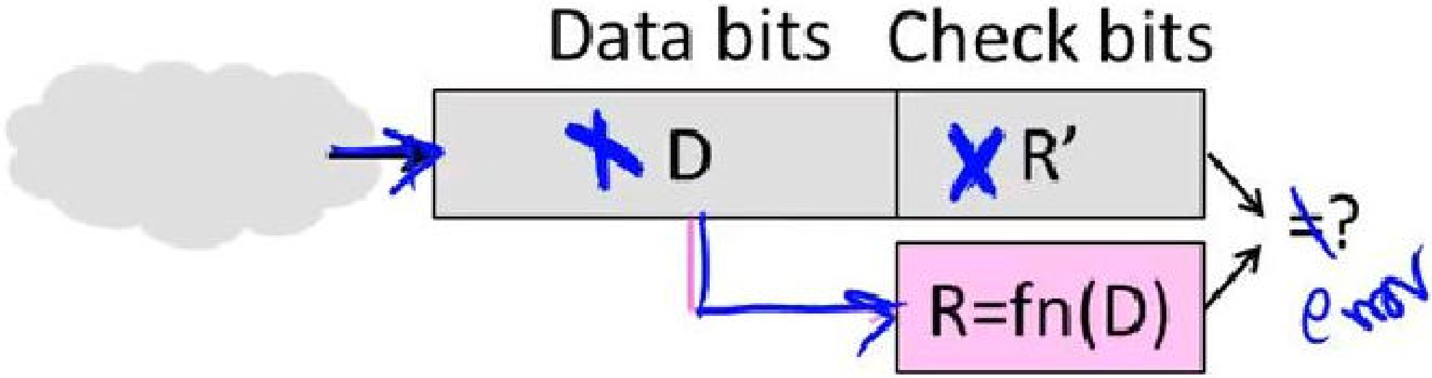
Data bits Check bits

D R=fn(D)

- Sender:
  - Compute R check bits based on the D data bits; send the codeword of D+R bits

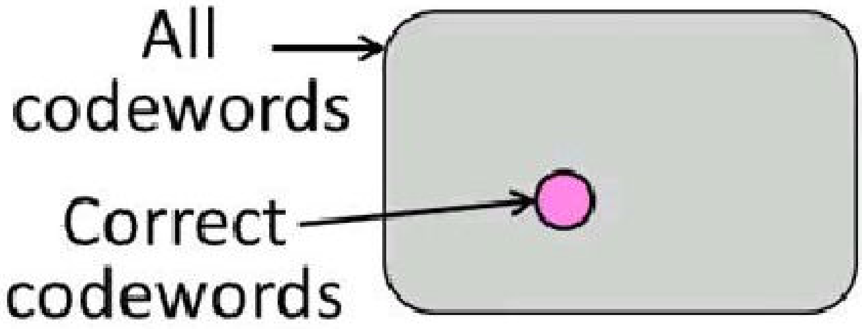
Using Error Codes (2)

- Receiver:
  - Receive D+R bits with unknown errors
  - Recompute R check bits based on the D data bits; error if R doesn't match R'



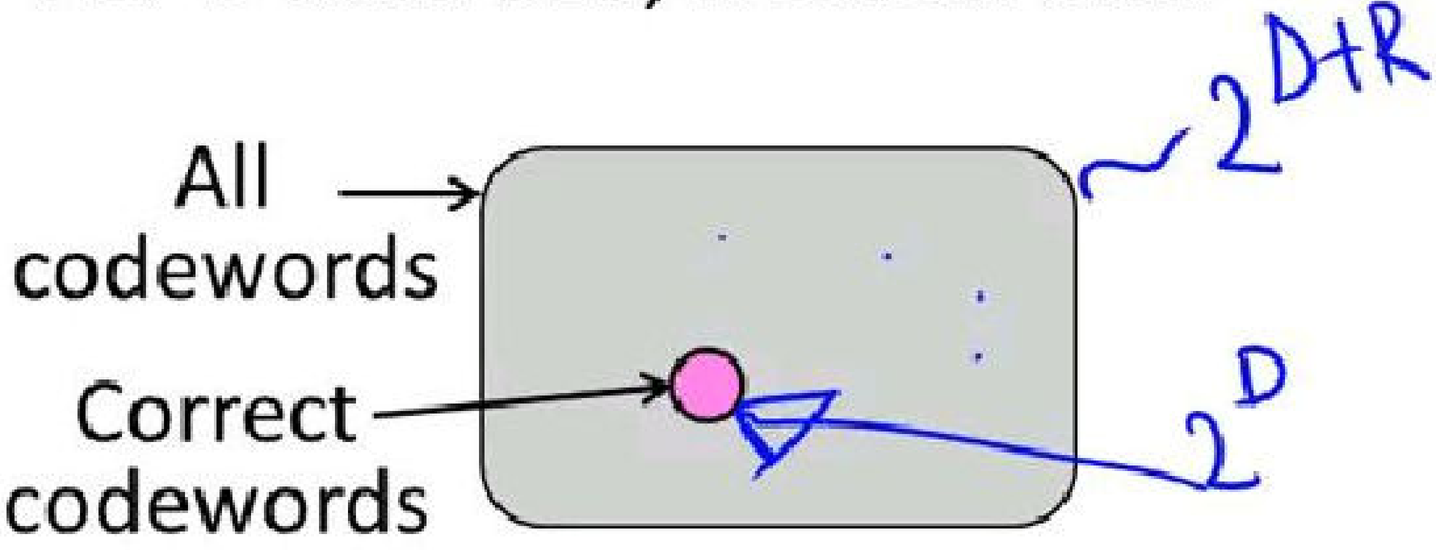
Intuition for Error Codes

- For D data bits, R check bits:



- Randomly chosen codeword is unlikely to be correct; overhead is low

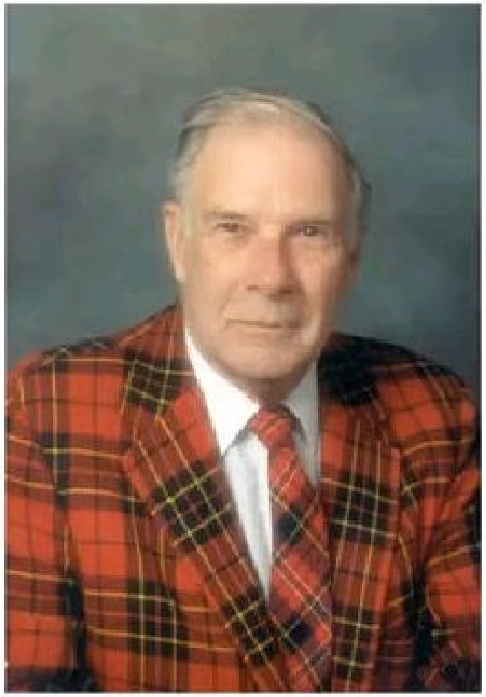
- For D data bits, R check bits:



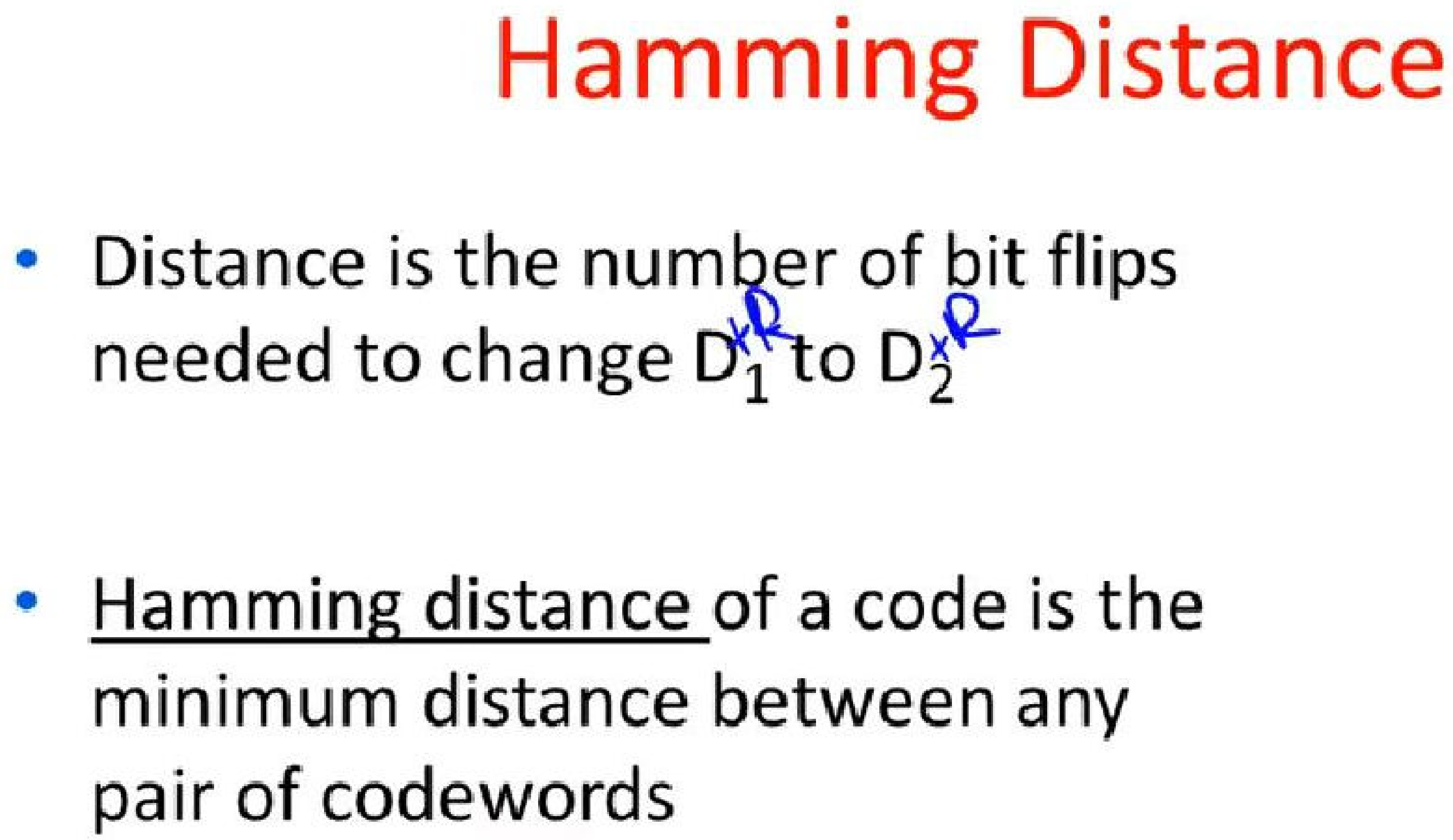
- Randomly chosen codeword is unlikely $\frac{1}{2^R}$ to be correct; overhead is low

R.W. Hamming (1915-1998)

- Much early work on codes:
  - "Error Detecting and Error Correcting Codes", BSTJ, 1950
- See also:
  - "You and Your Research", 1986



Source: IEEE GHN, © 2009 IEEE



Hamming Distance

- Distance is the number of bit flips needed to change D_1^{xR} to D_2^{xR}
- Hamming distance of a code is the minimum distance between any pair of codewords

Hamming Distance

- Distance is the number of bit flips needed to change D_1^{xR} to D_2^{xR}

  $1 \rightarrow 111$, $0 \rightarrow 000$ distance $= 3$

- Hamming distance of a code is the minimum distance between any pair of codewords $HD = 3$

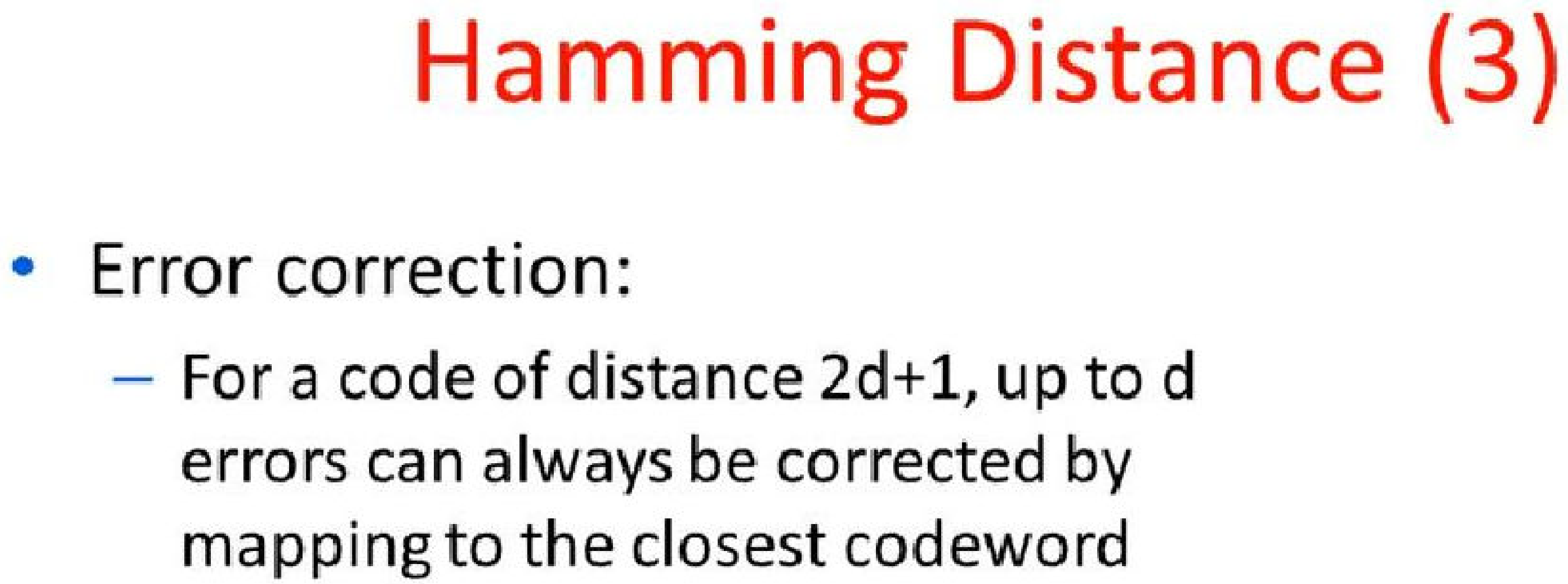
Hamming Distance (2)

- Error detection:
  - For a code of distance d+1, up to d errors will always be detected

$d+1=3 \Rightarrow d=2$

000 111

001 010
100 011
101 110



Hamming Distance (3)

- Error correction:
  - For a code of distance 2d+1, up to d errors can always be corrected by mapping to the closest codeword

Hamming Distance (3)

- Error correction:
  - For a code of distance 2d+1, up to d errors can always be corrected by mapping to the closest codeword

HD=3   2d+1=3

d=1

000 ← 010

110 ↓ 111

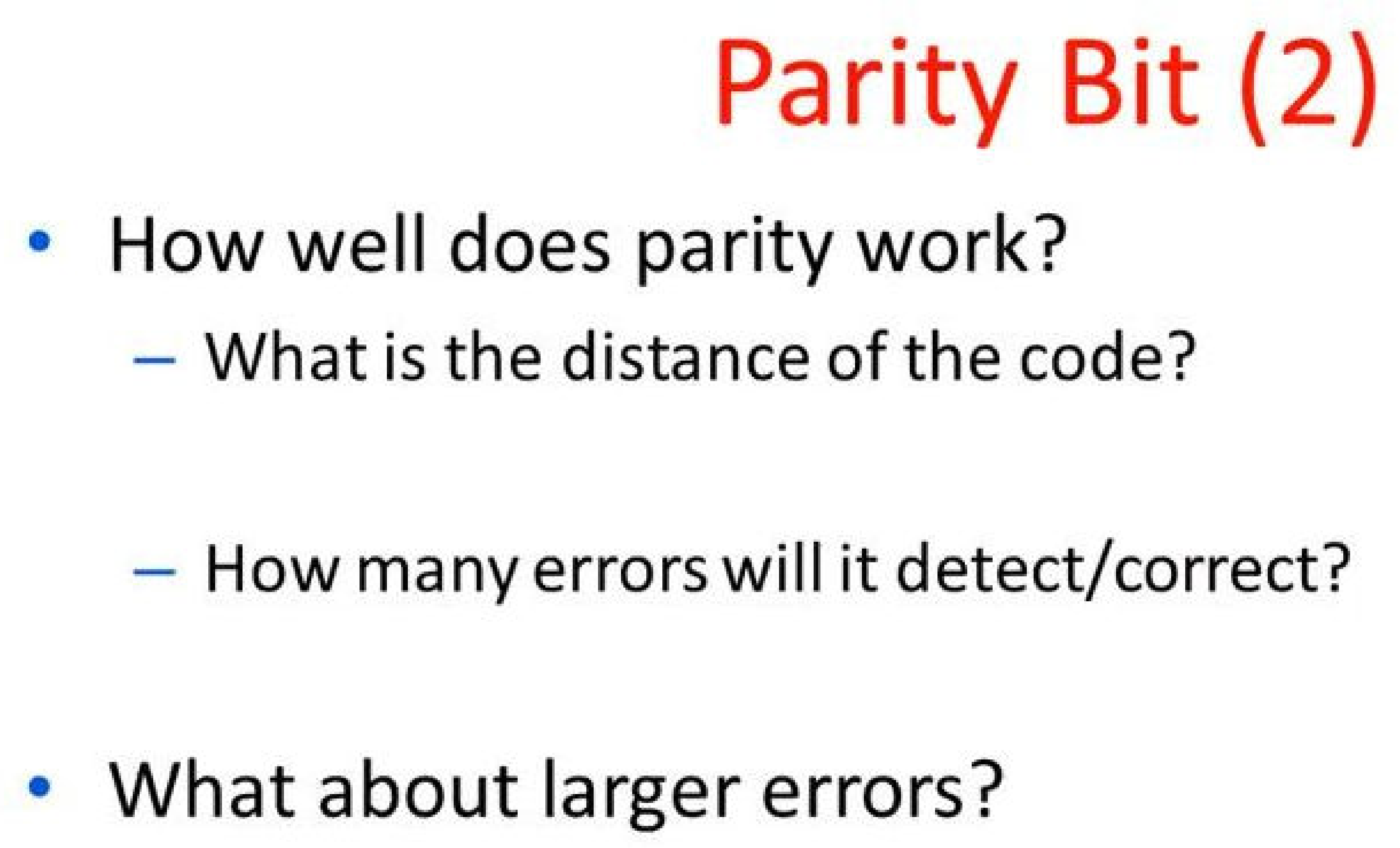
Error Detection 2:

1. Parity Codes
2. Checksum Codes
3. CRC Codes

Simple Error Detection – Parity Bit

- Take D data bits, add 1 check bit that is the sum of the D bits
  - Sum is modulo 2 or XOR

1001100



Parity Bit (2)

- How well does parity work?
  - What is the distance of the code?
  - How many errors will it detect/correct?
- What about larger errors?

Parity Bit (2)

- How well does parity work?
  - What is the distance of the code?

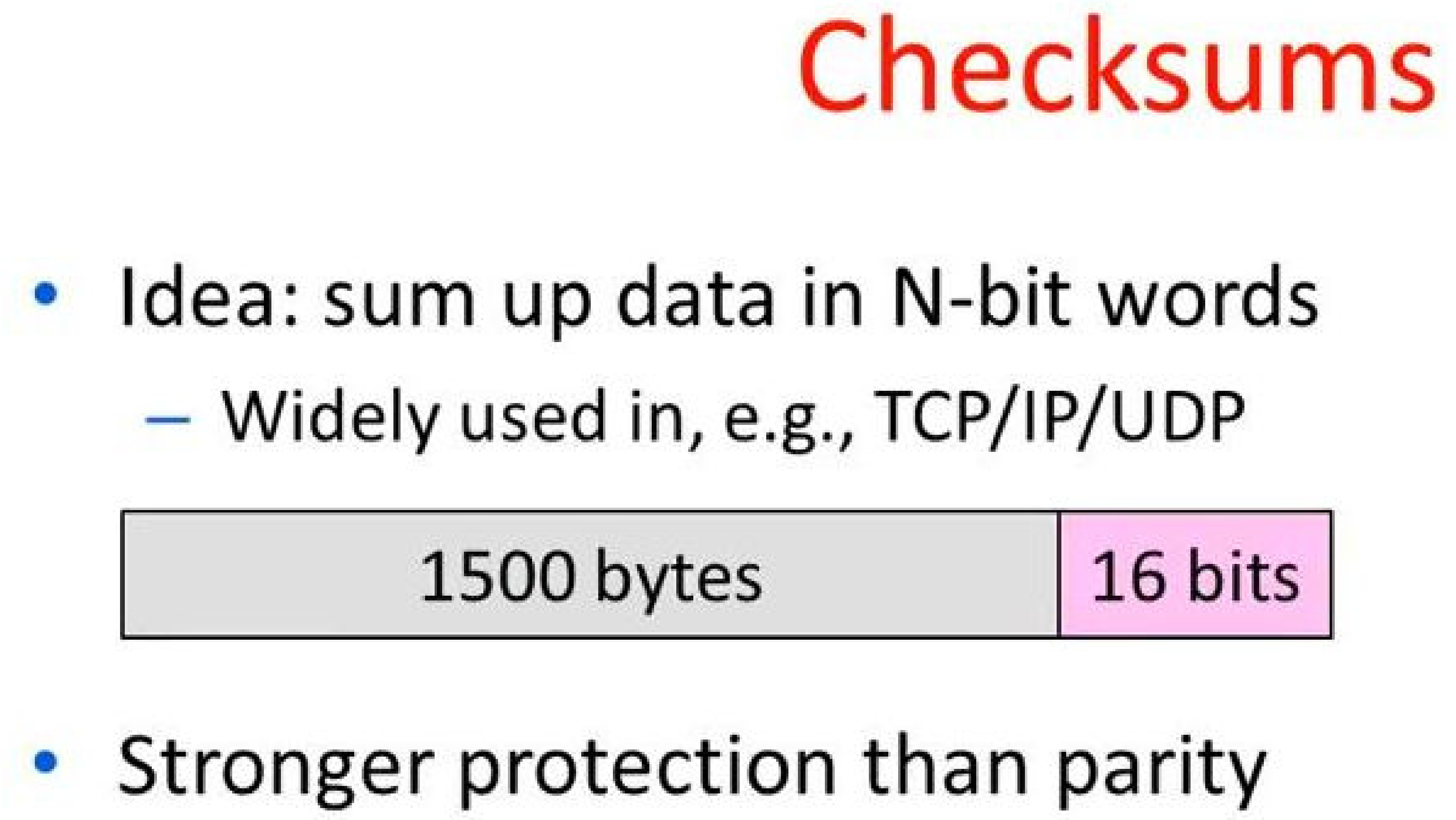
    2

  - How many errors will it detect/correct?

    1 0

- What about larger errors?

  odd # errors



Checksums

- Idea: sum up data in N-bit words
  - Widely used in, e.g., TCP/IP/UDP

1500 bytes	16 bits

- Stronger protection than parity

Internet Checksum

- Sum is defined in 1s complement arithmetic (must add back carries)
  - And it's the negative sum
- *"The checksum field is the 16 bit one's complement of the one's complement sum of all 16 bit words ..."* – RFC 791

1s comp 001 – "1"
110 – "-1"

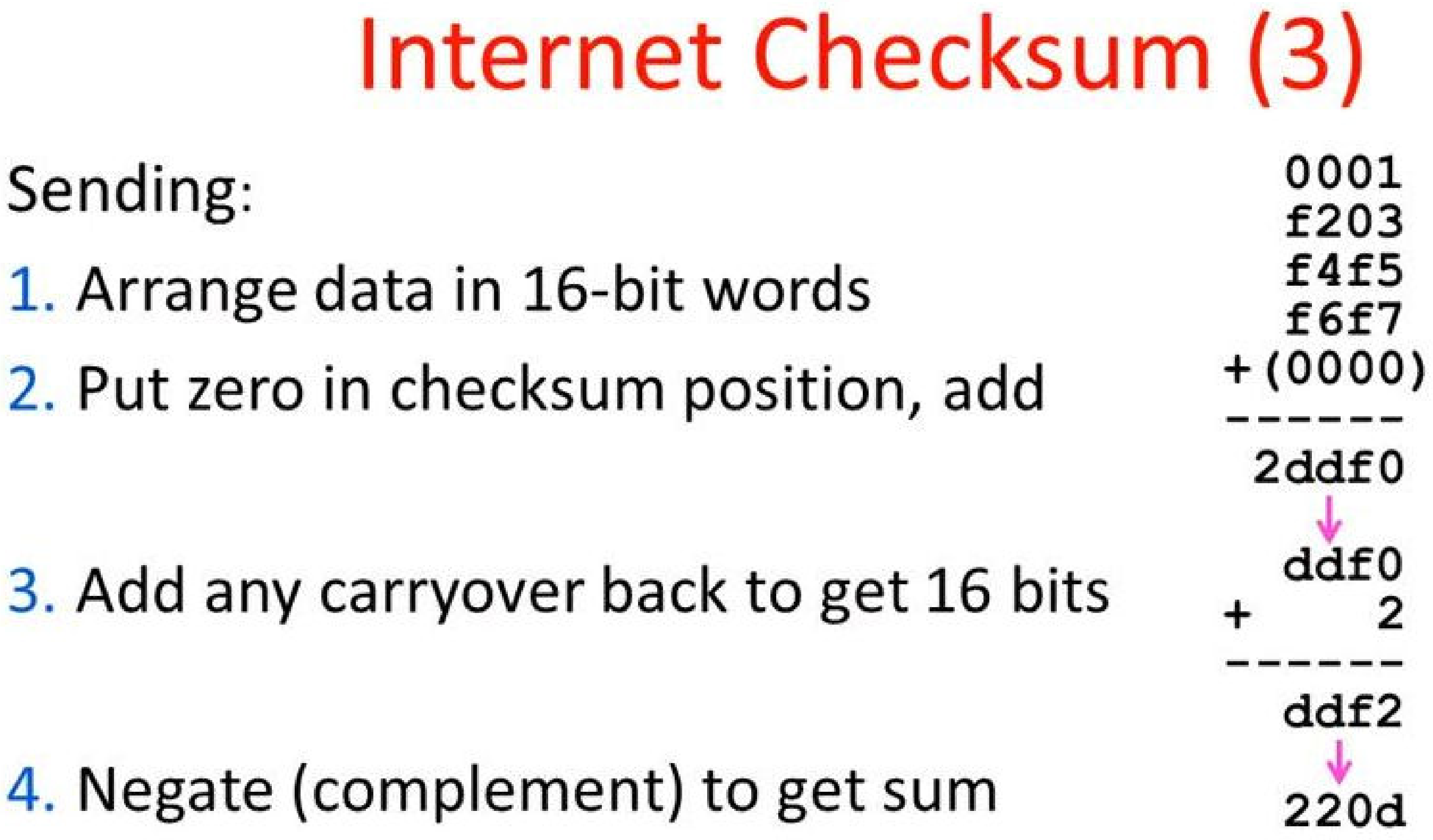
-1 in 2s comp 111

Internet Checksum (2)

Sending:

1. Arrange data in 16-bit words
2. Put zero in checksum position, add
3. Add any carryover back to get 16 bits
4. Negate (complement) to get sum

```
0001
f203
f4f5
f6f7
```

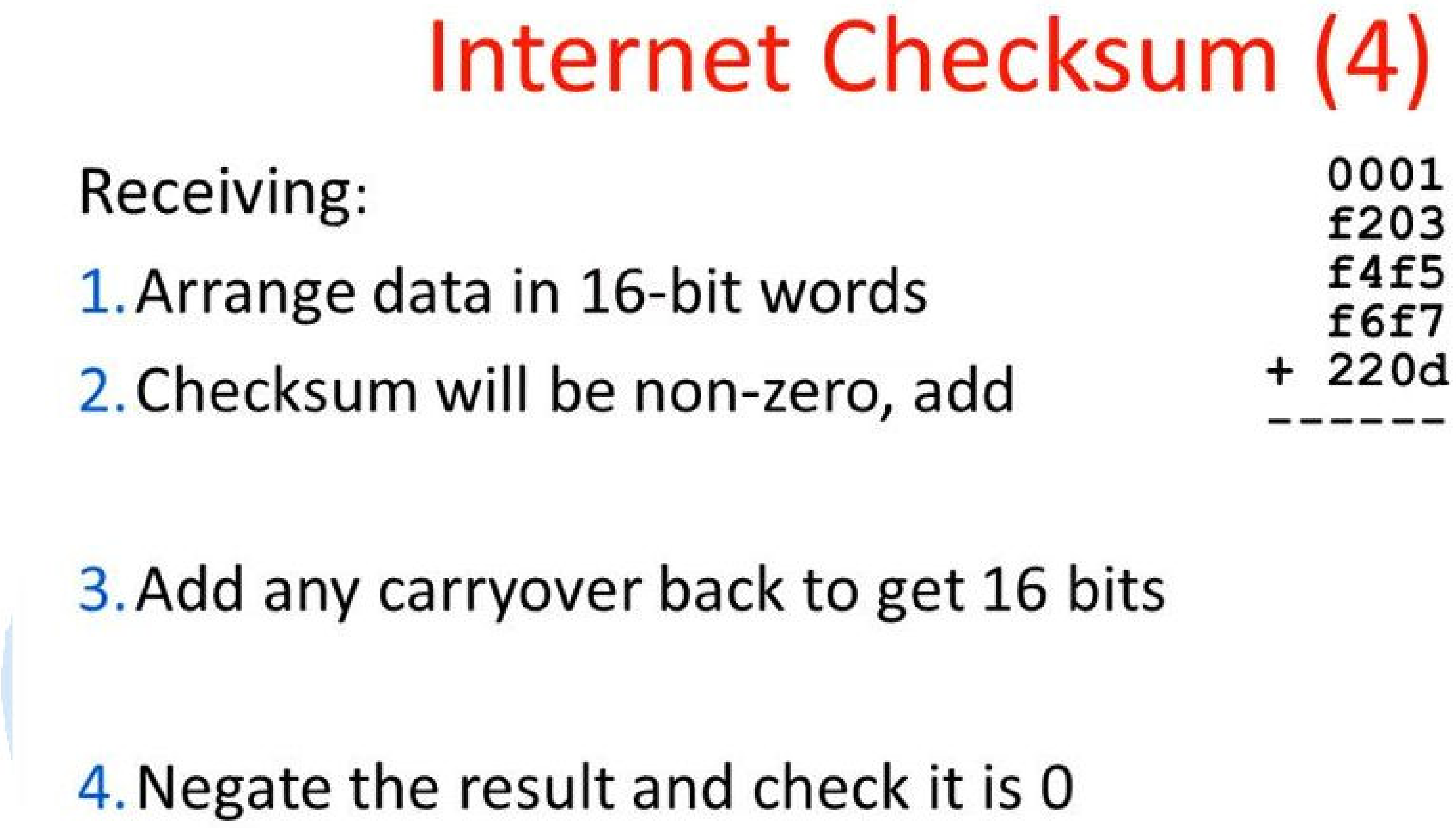


Internet Checksum (3)

Sending:

1. Arrange data in 16-bit words
2. Put zero in checksum position, add
3. Add any carryover back to get 16 bits
4. Negate (complement) to get sum

```
   0001
   f203
   f4f5
   f6f7
 +(0000)
 ------
  2ddf0
    ↓
   ddf0
 +    2
 ------
   ddf2
    ↓
   220d
```



Internet Checksum (4)

Receiving:

1. Arrange data in 16-bit words
2. Checksum will be non-zero, add
3. Add any carryover back to get 16 bits
4. Negate the result and check it is 0

```
   0001
   f203
   f4f5
   f6f7
+  220d
 ------
```

Internet Checksum (5)

Receiving:

1. Arrange data in 16-bit words
2. Checksum will be non-zero, add
3. Add any carryover back to get 16 bits
4. Negate the result and check it is 0

```
   0001
   f203
   f4f5
   f6f7
+  220d
------
  2fffd
    ↓
   fffd
+     2
------
   ffff
    ↓
   0000
```

Internet Checksum (6)

- How well does the checksum work?
  - What is the distance of the code? 2
  - How many errors will it detect/correct?

1 0

- What about larger errors?

Cyclic Redundancy Check (CRC)

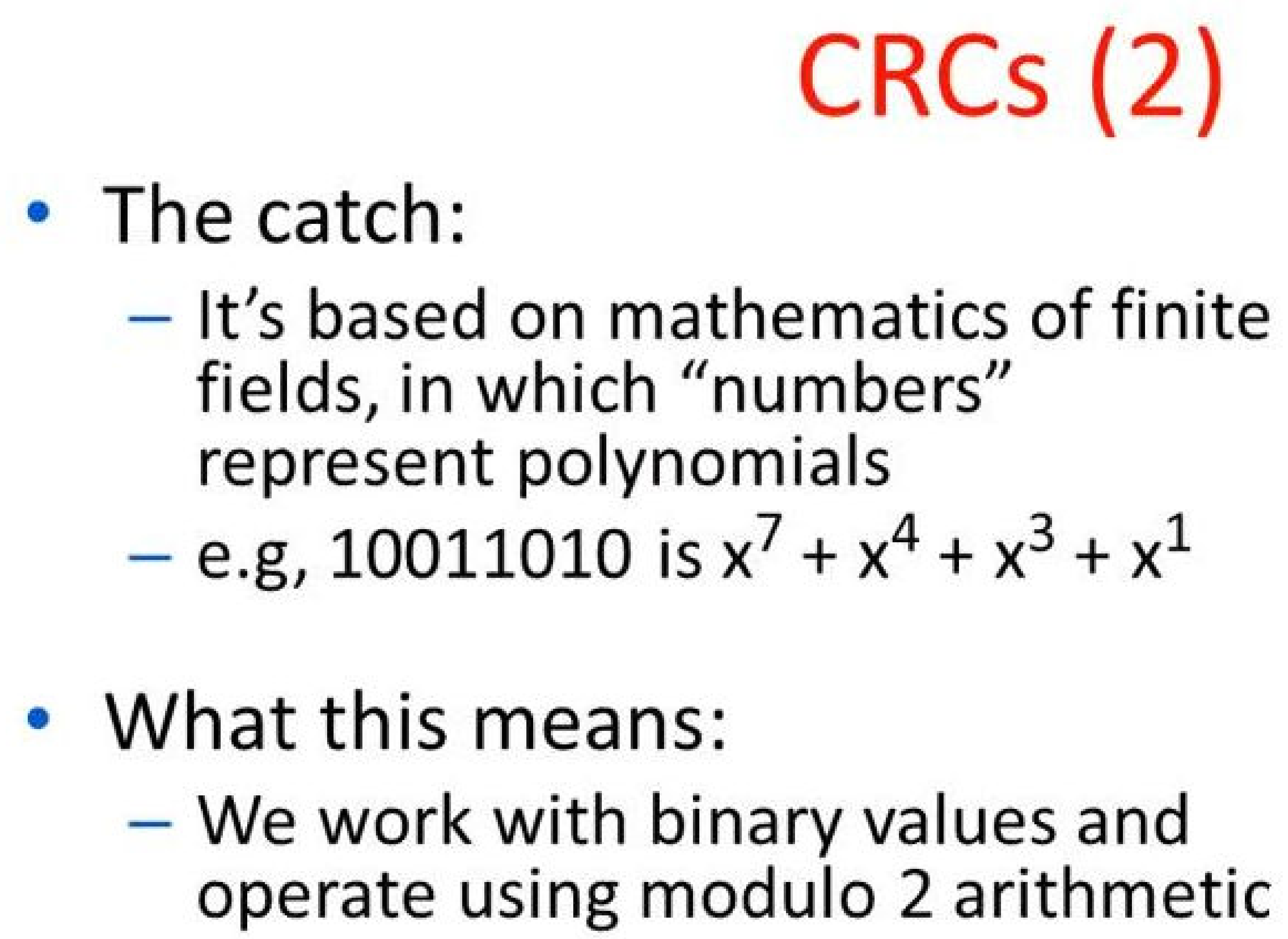
- Even stronger protection
  - Given n data bits, generate k check bits such that the n+k bits are evenly divisible by a generator C
- Example with numbers:
  - n = 302, k = one digit, C = 3

Cyclic Redundancy Check (CRC)

- Even stronger protection
  - Given n data bits, generate k check bits such that the n+k bits are evenly divisible by a generator C
- Example with numbers:
  - n = 302, k = one digit, C = 3

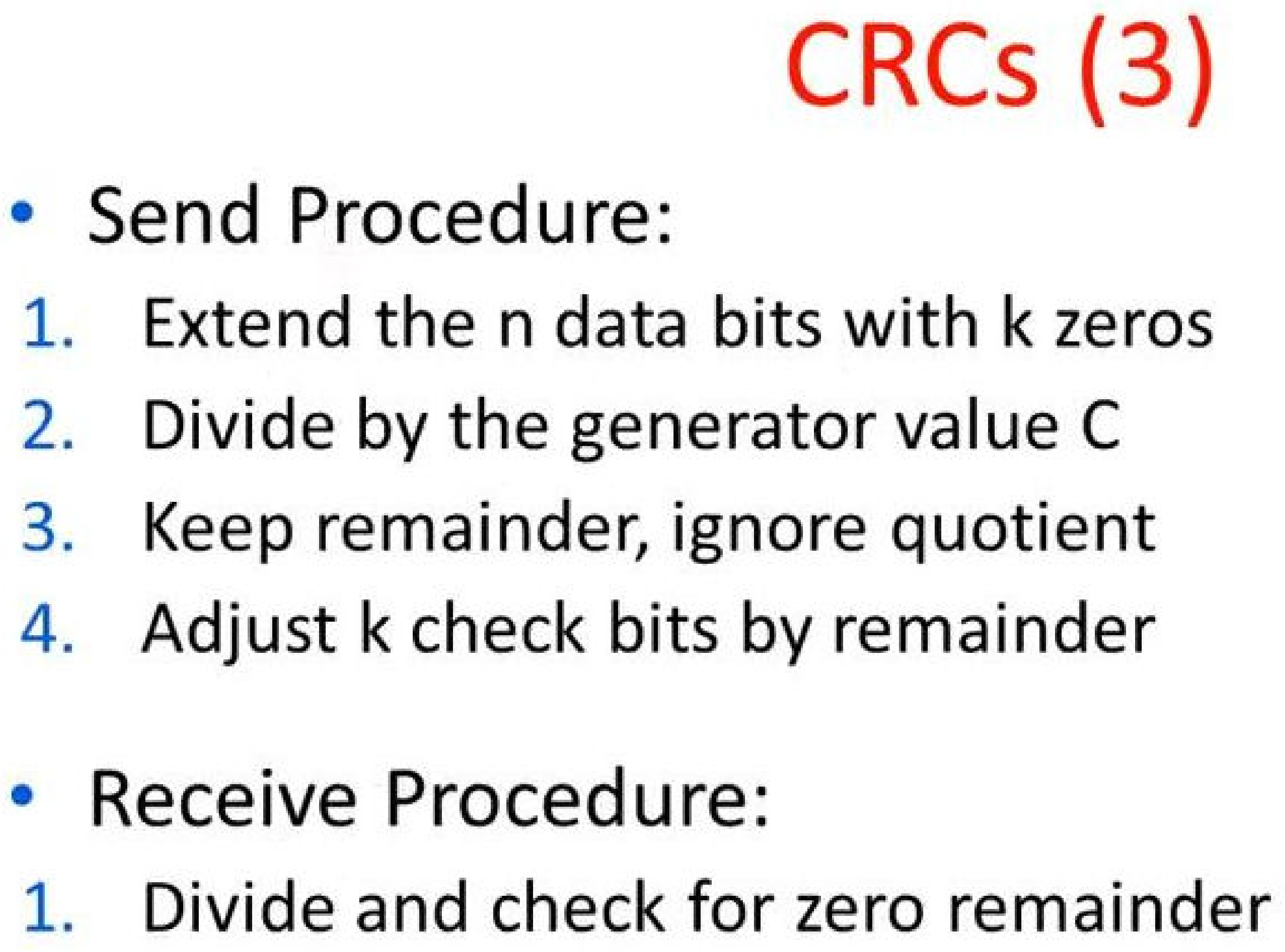
302$\underline{1}$

3020 % 3
= 2



CRCs (2)

- The catch:
  - It’s based on mathematics of finite fields, in which “numbers” represent polynomials
  - e.g, 10011010 is $x^7 + x^4 + x^3 + x^1$
- What this means:
  - We work with binary values and operate using modulo 2 arithmetic



CRCs (3)

- Send Procedure:

1. Extend the n data bits with k zeros
2. Divide by the generator value C
3. Keep remainder, ignore quotient
4. Adjust k check bits by remainder

- Receive Procedure:

1. Divide and check for zero remainder

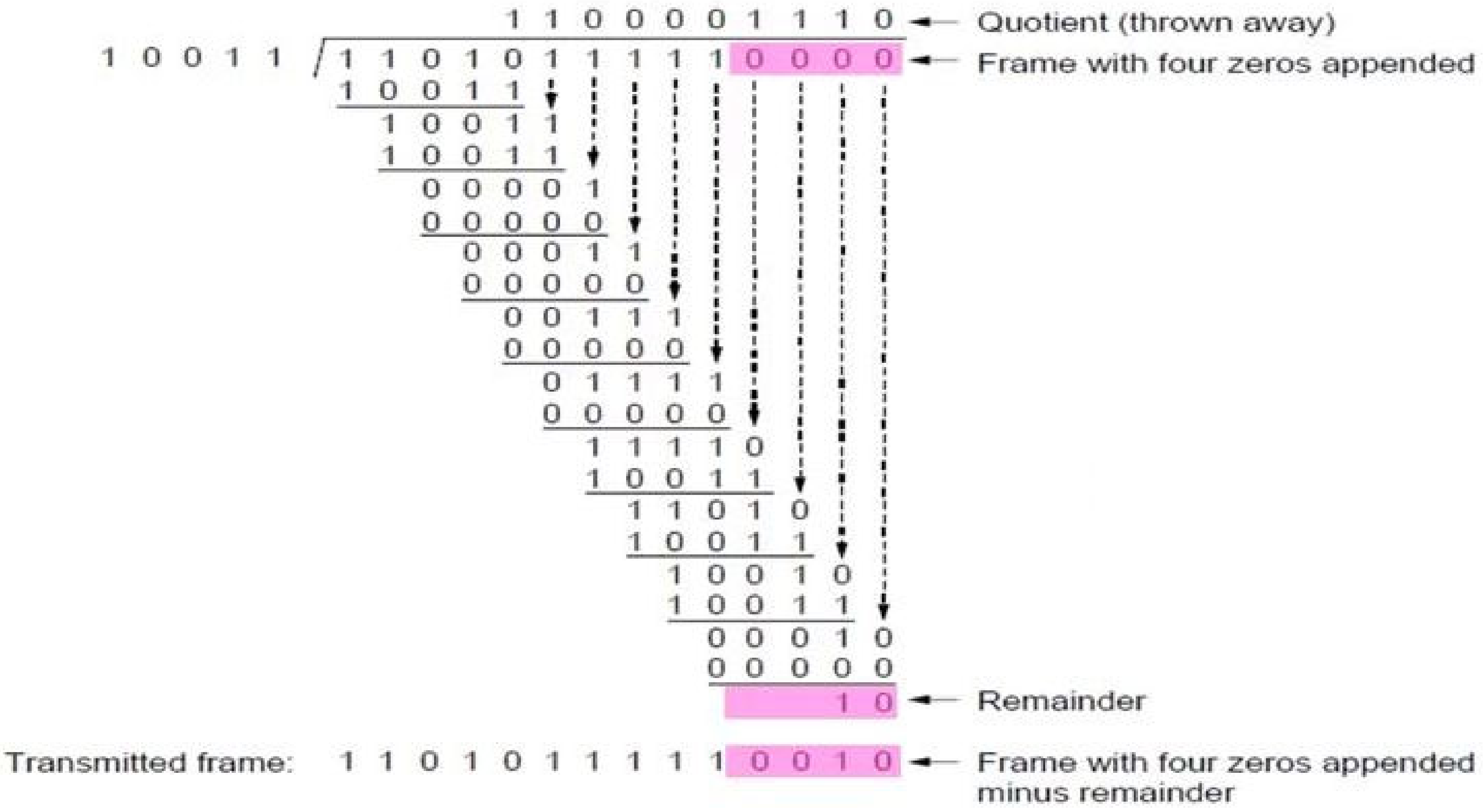
CRCs (4)

Data bits:
1101011111

Check bits:
$C(x)=x^4+x^1+1$
C = 10011
k = 4

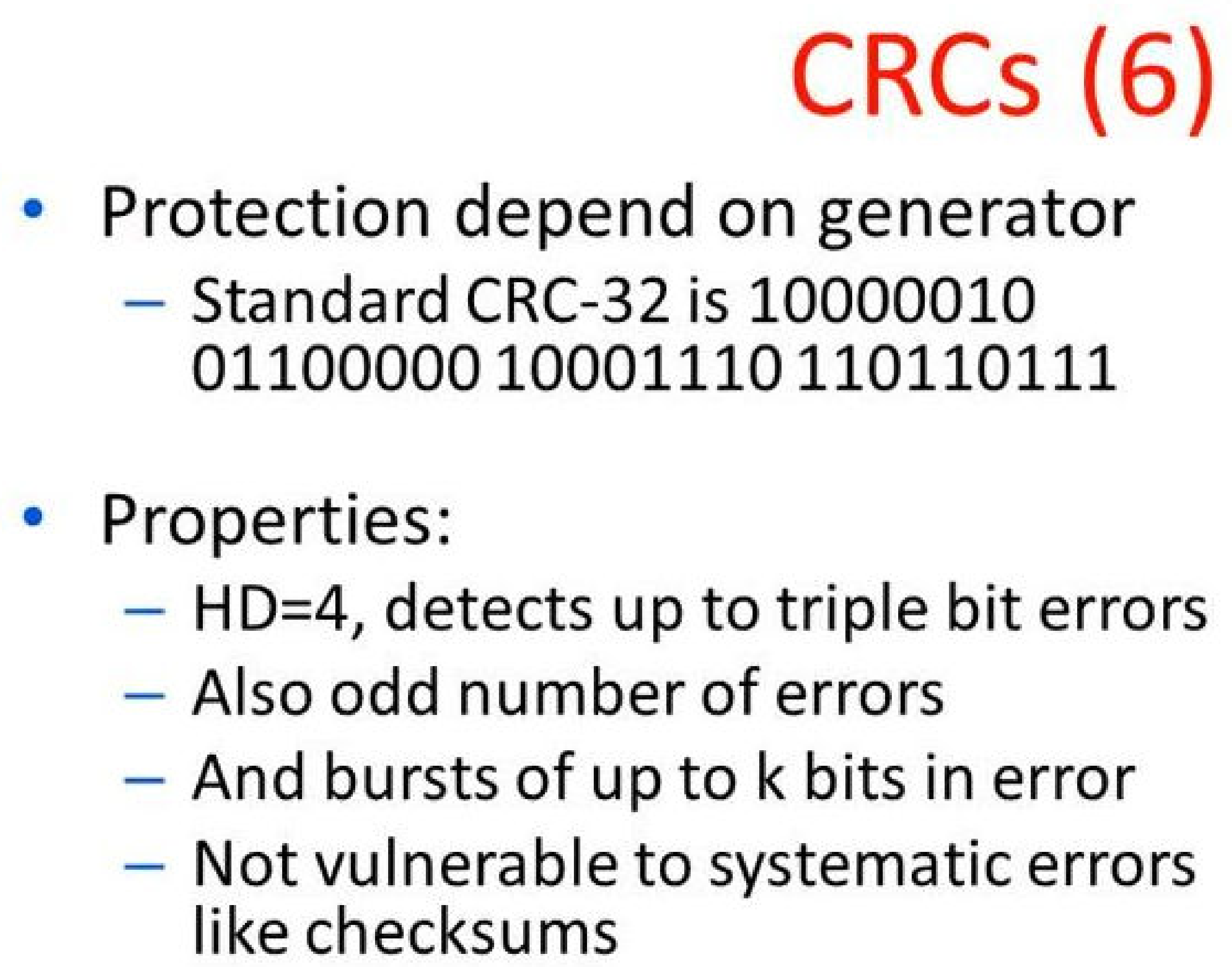
$$1\,0\,0\,1\,1\,\overline{\big)\,1\;1\;0\;1\;0\;1\;1\;1\;1\;1\qquad\qquad}$$

CRCs (5)



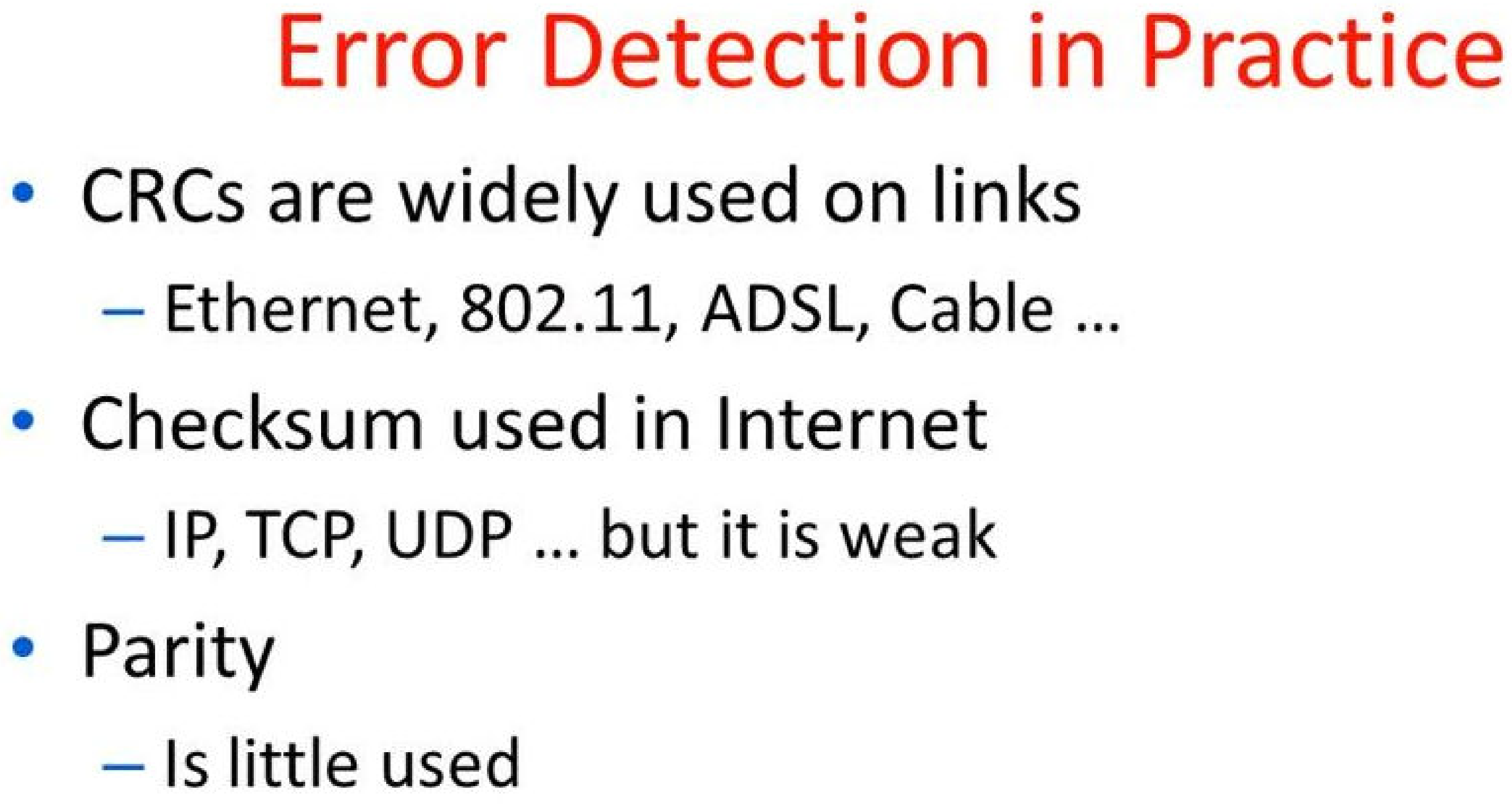
```
                    1 1 0 0 0 0 1 1 1 0     ← Quotient (thrown away)
            ____________________________
1 0 0 1 1 ) 1 1 0 1 0 1 1 1 1 1 0 0 0 0     ← Frame with four zeros appended
            1 0 0 1 1
              1 0 0 1 1
              1 0 0 1 1
                0 0 0 0 1
                0 0 0 0 0
                  0 0 0 1 1
                  0 0 0 0 0
                    0 0 1 1 1
                    0 0 0 0 0
                      0 1 1 1 1
                      0 0 0 0 0
                        1 1 1 1 0
                        1 0 0 1 1
                          1 1 0 1 0
                          1 0 0 1 1
                            1 0 0 1 0
                            1 0 0 1 1
                              0 0 0 1 0
                              0 0 0 0 0
                                    1 0     ← Remainder

Transmitted frame:  1 1 0 1 0 1 1 1 1 1 0 0 1 0     ← Frame with four zeros appended
                                                       minus remainder
```



CRCs (6)

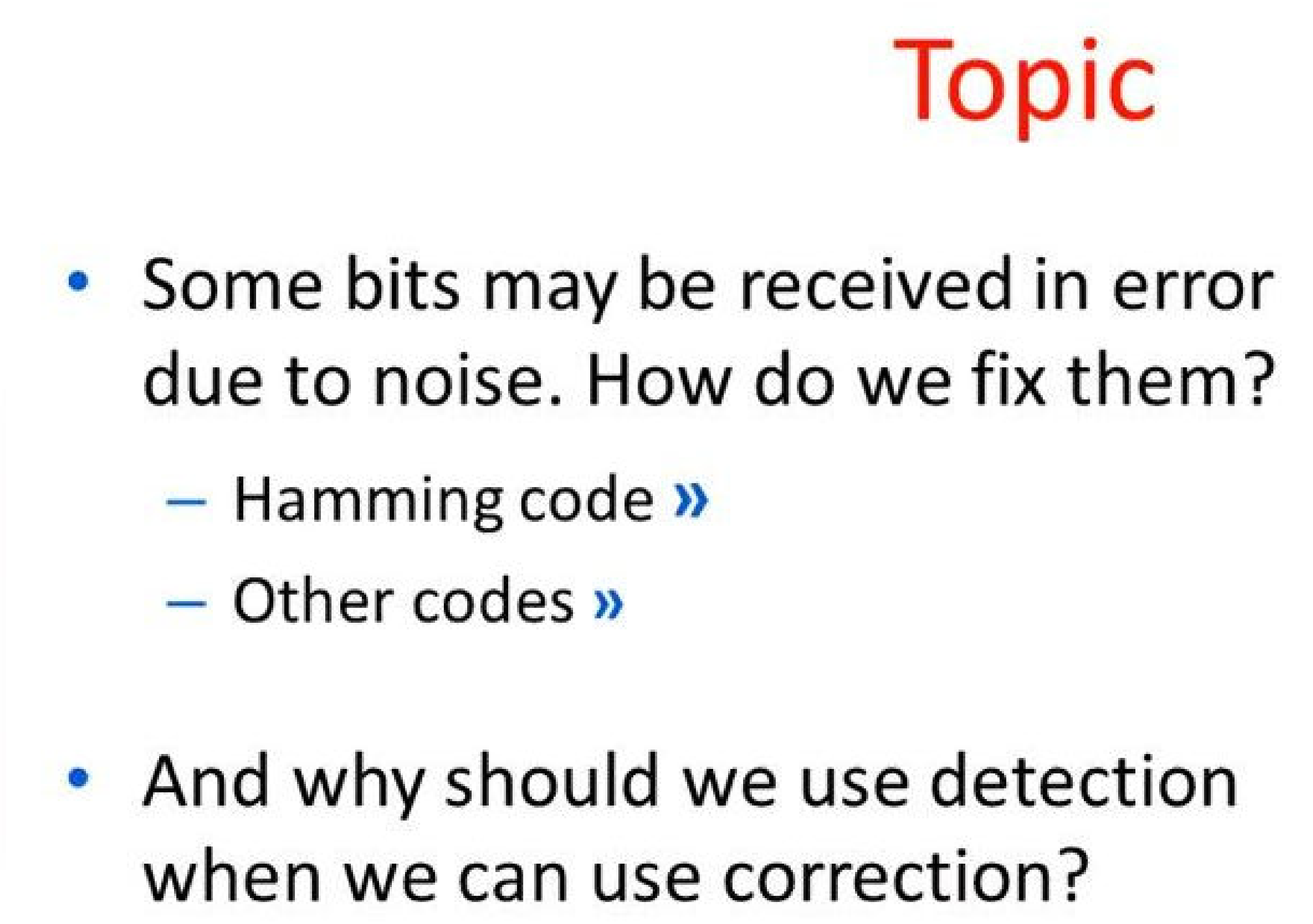
- Protection depend on generator
  - Standard CRC-32 is 10000010 01100000 10001110 110110111
- Properties:
  - HD=4, detects up to triple bit errors
  - Also odd number of errors
  - And bursts of up to k bits in error
  - Not vulnerable to systematic errors like checksums



Error Detection in Practice

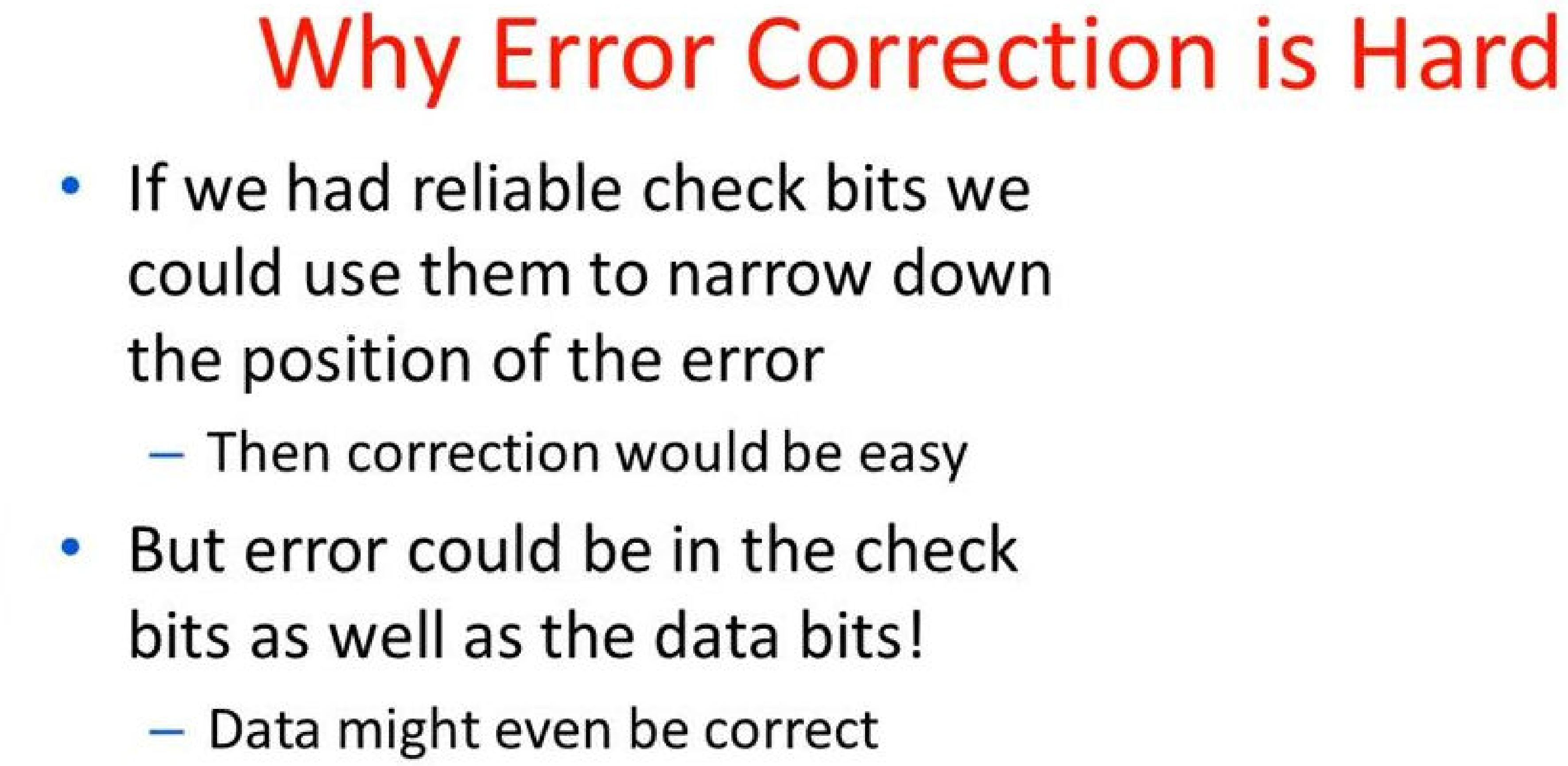
- CRCs are widely used on links
  - Ethernet, 802.11, ADSL, Cable …
- Checksum used in Internet
  - IP, TCP, UDP … but it is weak
- Parity
  - Is little used

Error Correction



Topic

- Some bits may be received in error due to noise. How do we fix them?
  - Hamming code »
  - Other codes »
- And why should we use detection when we can use correction?



Why Error Correction is Hard

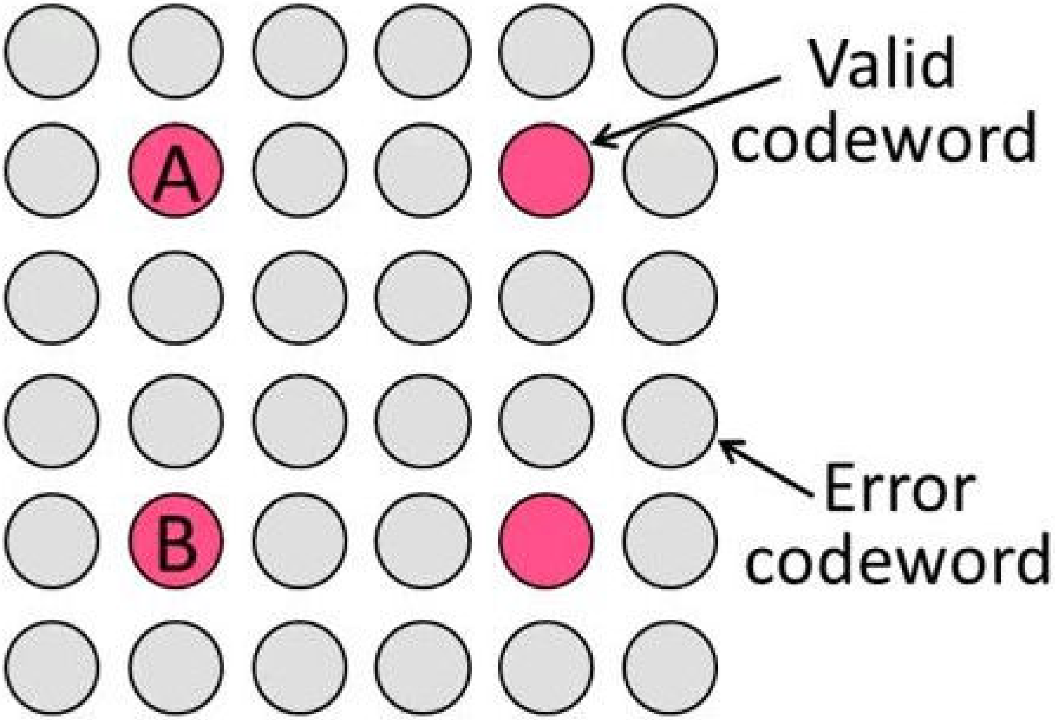
- If we had reliable check bits we could use them to narrow down the position of the error
  - Then correction would be easy
- But error could be in the check bits as well as the data bits!
  - Data might even be correct

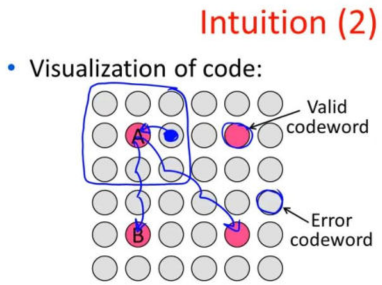
Intuition for Error Correcting Code

- Suppose we construct a code with a Hamming distance of at least 3
  - Need ≥3 bit errors to change one valid codeword into another
  - Single bit errors will be closest to a unique valid codeword
- If we assume errors are only 1 bit, we can correct them by mapping an error to the closest valid codeword
  - Works for d errors if HD ≥ 2d + 1

Intuition (2)

- Visualization of code:

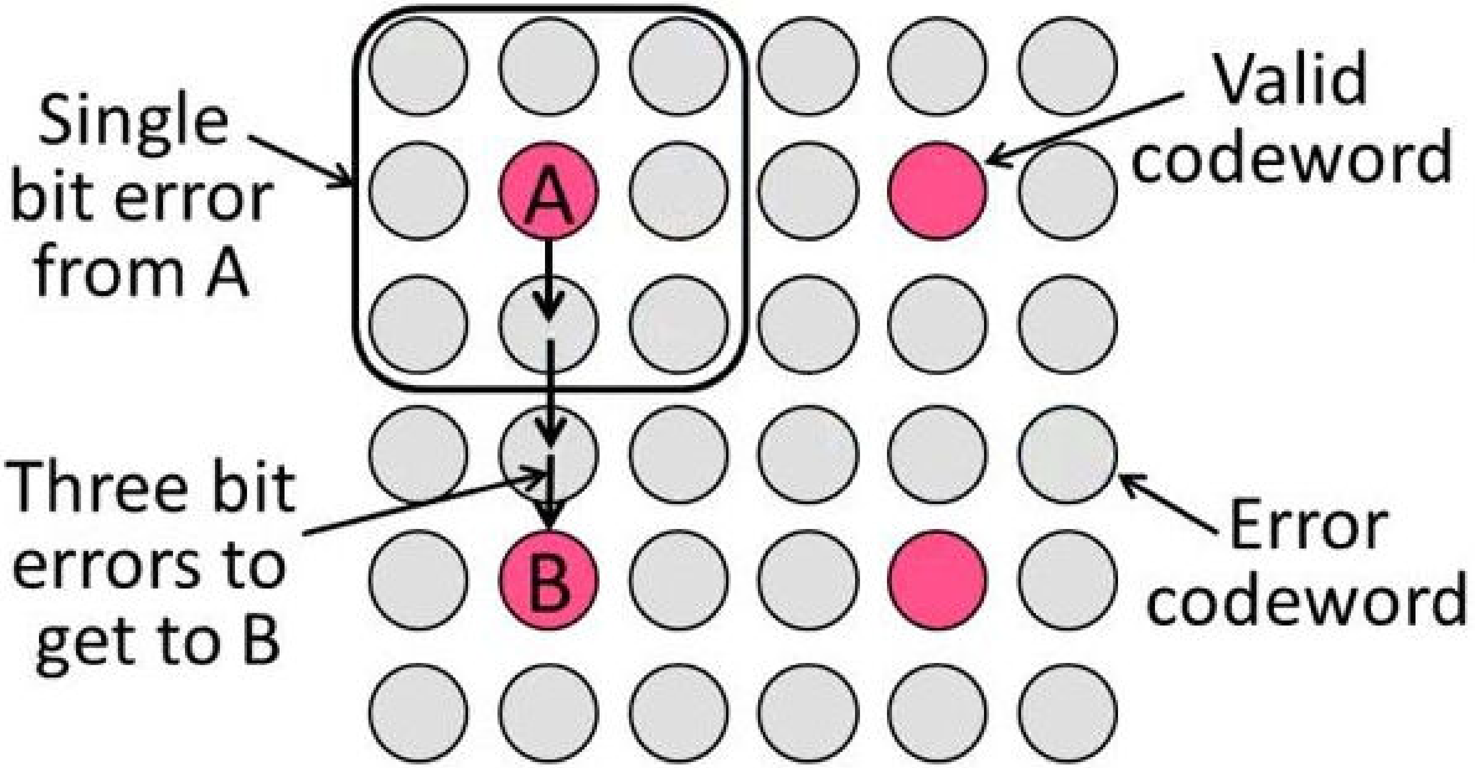



Intuition (2)

- Visualization of code:

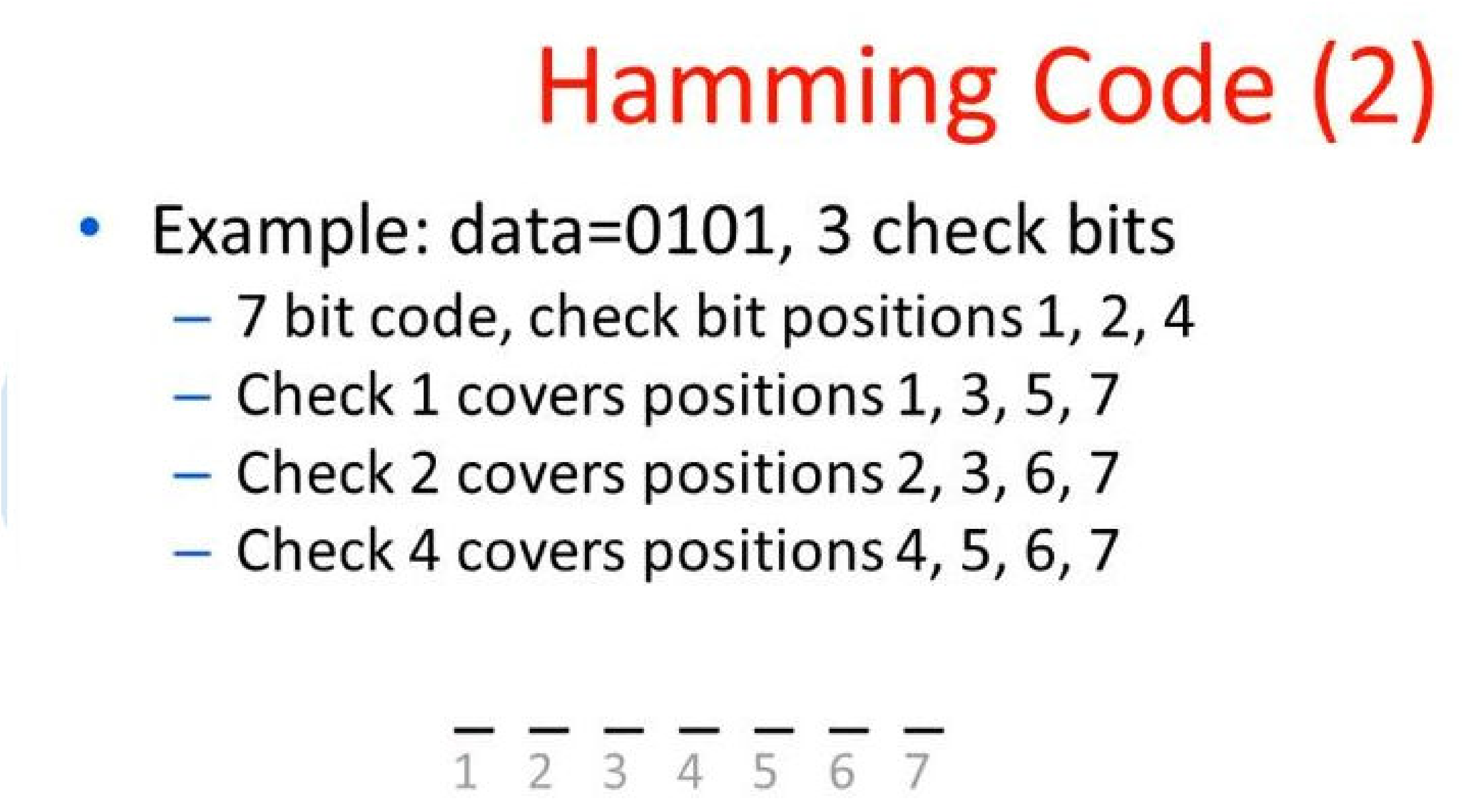
Intuition (3)

- Visualization of code:



Hamming Code

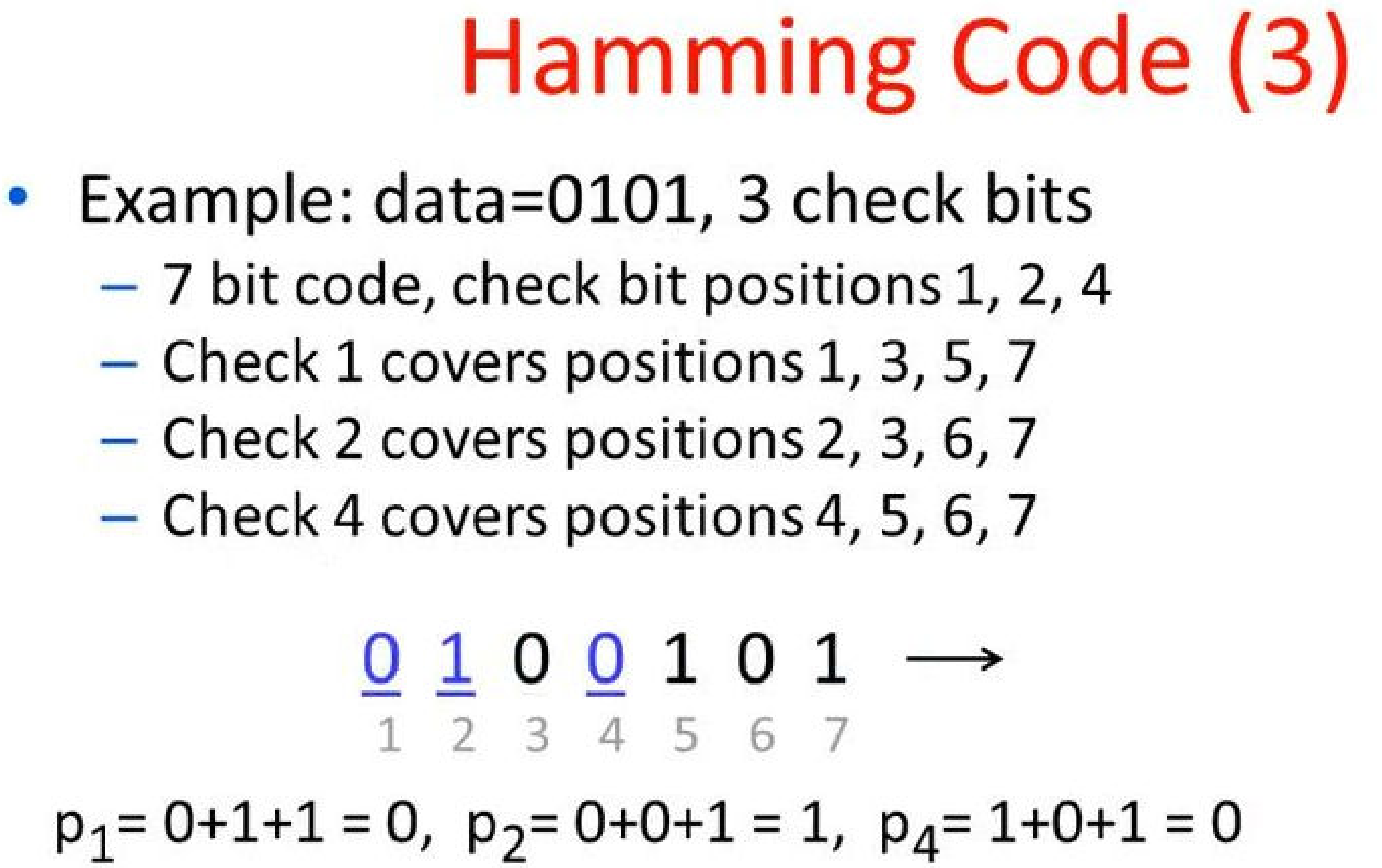
- Gives a method for constructing a code with a distance of 3
  - → Uses $n = 2^k - k - 1$, e.g., $n=4$, $k=3$
  - Put check bits in positions p that are powers of 2, starting with position 1
  - Check bit in position p is parity of positions with a p term in their values
- Plus an easy way to correct [soon]



Hamming Code (2)

- Example: data=0101, 3 check bits
  - 7 bit code, check bit positions 1, 2, 4
  - Check 1 covers positions 1, 3, 5, 7
  - Check 2 covers positions 2, 3, 6, 7
  - Check 4 covers positions 4, 5, 6, 7

_ _ _ _ _ _ _
1 2 3 4 5 6 7

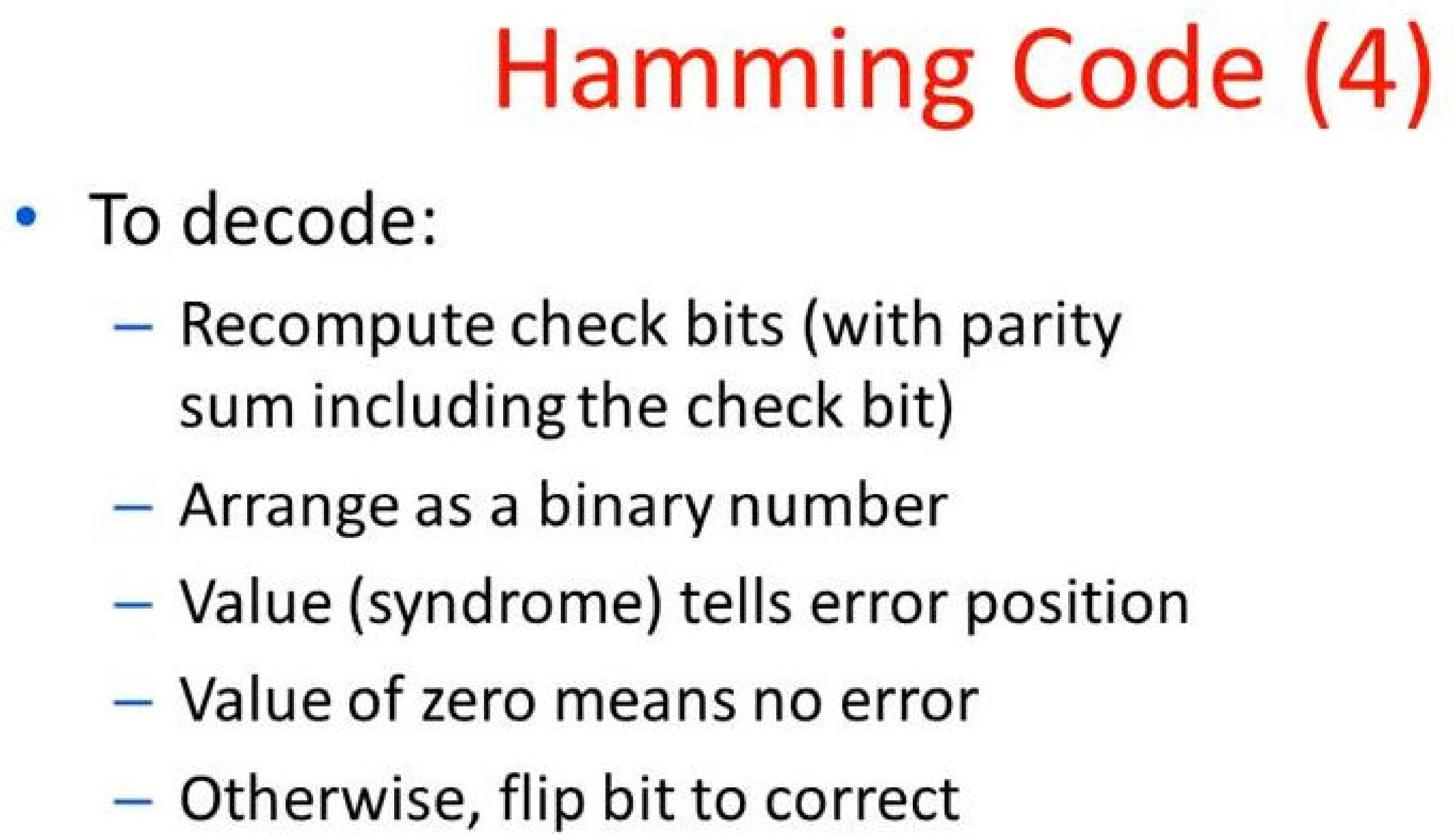


Hamming Code (3)

- Example: data=0101, 3 check bits
  - 7 bit code, check bit positions 1, 2, 4
  - Check 1 covers positions 1, 3, 5, 7
  - Check 2 covers positions 2, 3, 6, 7
  - Check 4 covers positions 4, 5, 6, 7

$$\begin{array}{ccccccccc} \underline{0} & \underline{1} & 0 & \underline{0} & 1 & 0 & 1 & \longrightarrow \\ 1 & 2 & 3 & 4 & 5 & 6 & 7 & \end{array}$$

p_1= 0+1+1 = 0, p_2= 0+0+1 = 1, p_4= 1+0+1 = 0



Hamming Code (4)

- To decode:
  - Recompute check bits (with parity sum including the check bit)
  - Arrange as a binary number
  - Value (syndrome) tells error position
  - Value of zero means no error
  - Otherwise, flip bit to correct

Hamming Code (5)

- Example, continued

$$\begin{array}{cccccccc} \longrightarrow & \underline{0} & \underline{1} & 0 & \underline{0} & 1 & 0 & 1 \\ & 1 & 2 & 3 & 4 & 5 & 6 & 7 \end{array}$$

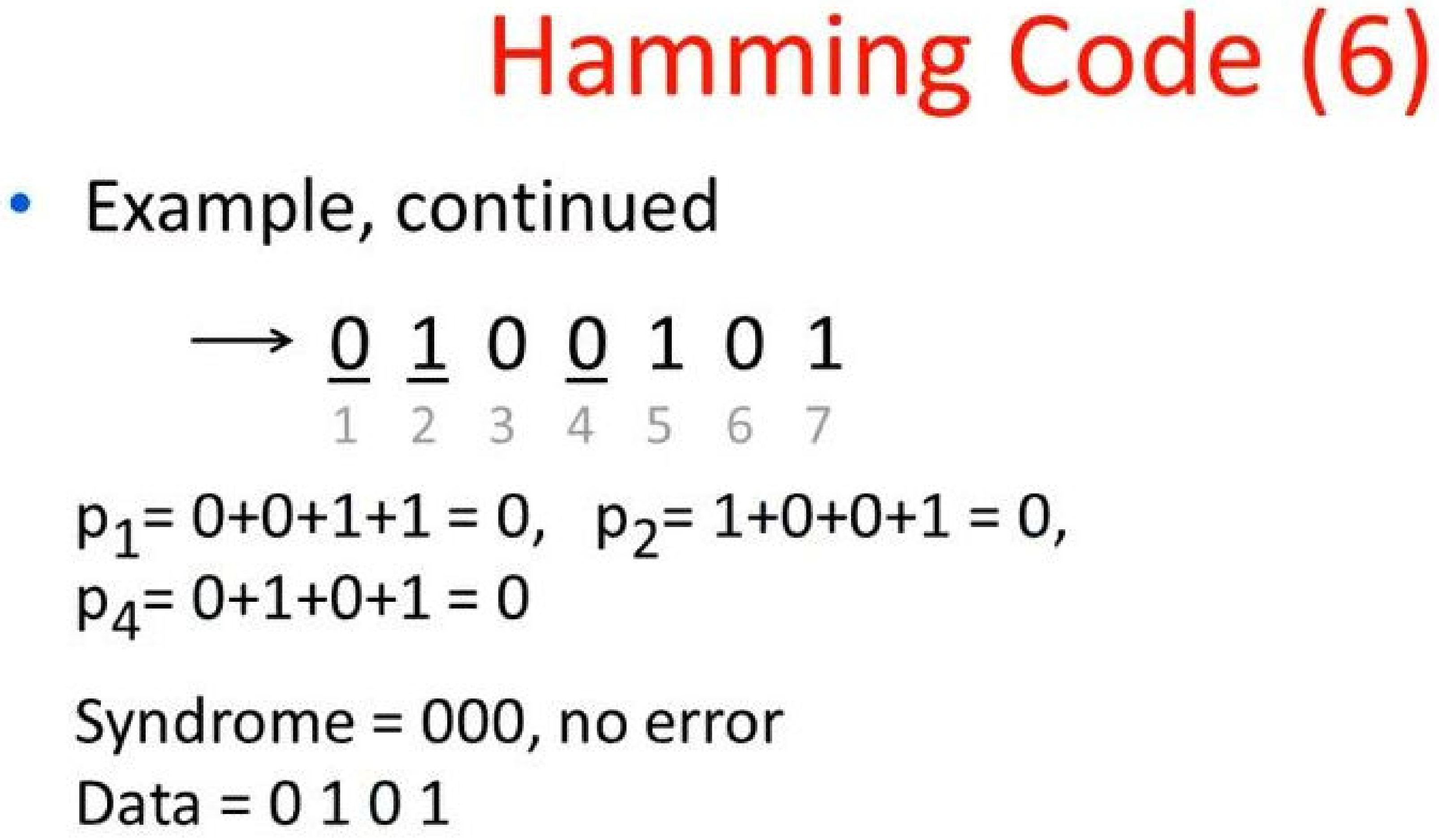
p_1=

p_2=

p_4=

Syndrome =

Data =



Hamming Code (6)

- Example, continued

$$\longrightarrow \begin{array}{ccccccc} \underline{0} & \underline{1} & 0 & \underline{0} & 1 & 0 & 1 \\ 1 & 2 & 3 & 4 & 5 & 6 & 7 \end{array}$$

p_1= 0+0+1+1 = 0, p_2= 1+0+0+1 = 0,
p_4= 0+1+0+1 = 0

Syndrome = 000, no error
Data = 0 1 0 1

Hamming Code (7)

- Example, continued

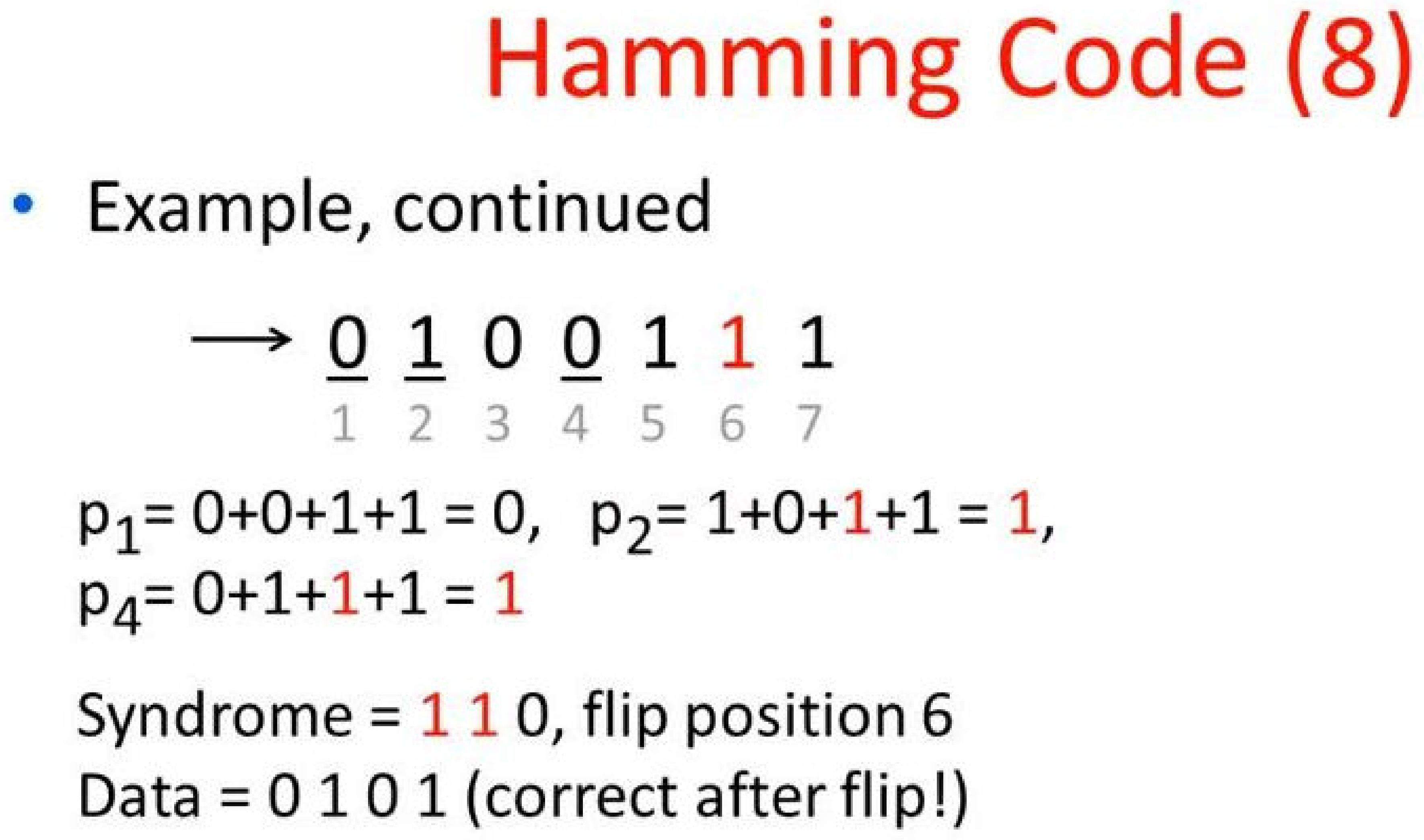
$$\longrightarrow \begin{array}{ccccccc} \underline{0} & \underline{1} & 0 & \underline{0} & 1 & 1 & 1 \\ 1 & 2 & 3 & 4 & 5 & 6 & 7 \end{array}$$

p_1= p_2=

p_4=

Syndrome =

Data =



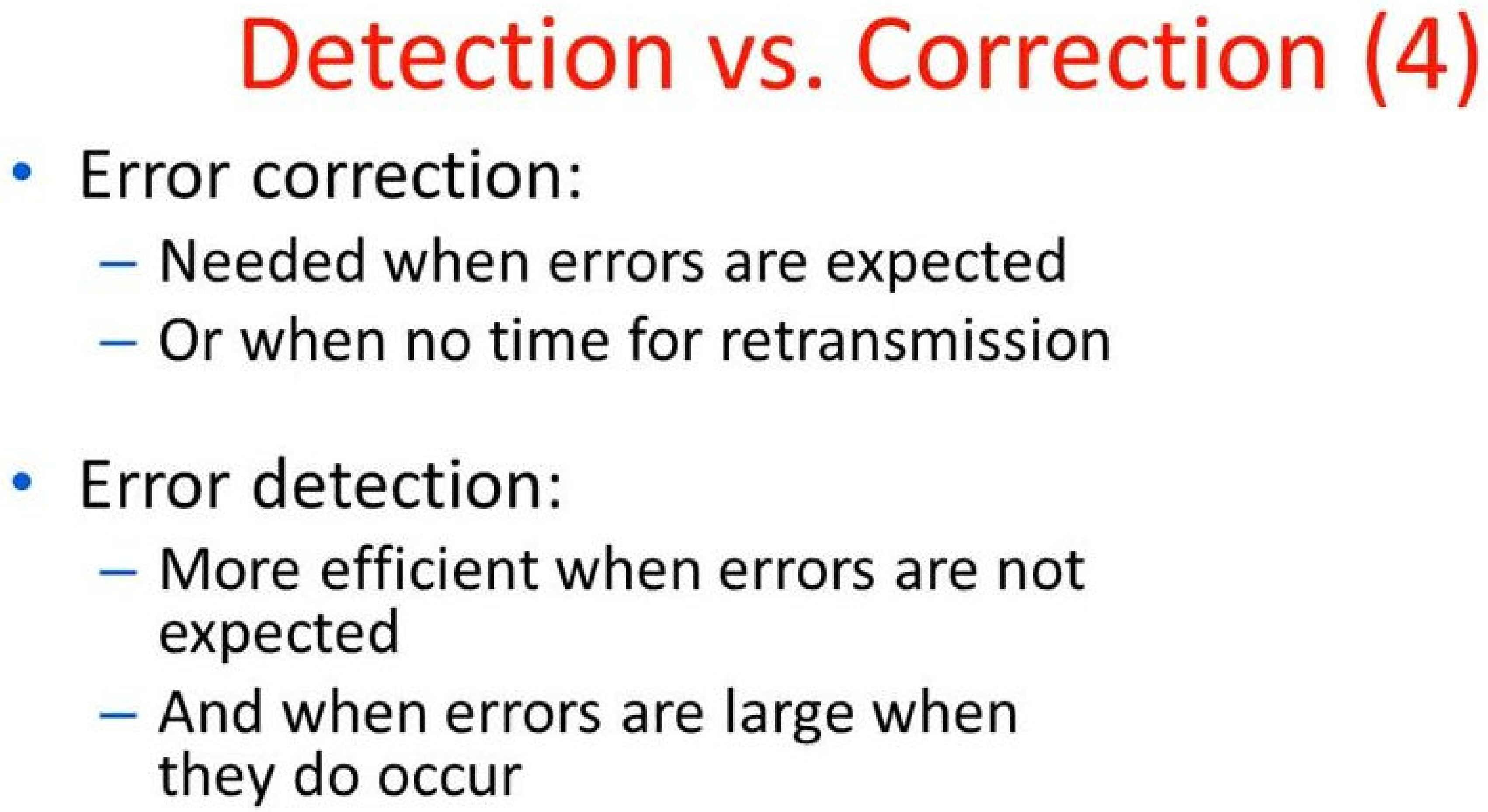
Hamming Code (8)

- Example, continued

$$\longrightarrow \begin{array}{ccccccc} \underline{0} & \underline{1} & 0 & \underline{0} & 1 & 1 & 1 \\ 1 & 2 & 3 & 4 & 5 & 6 & 7 \end{array}$$

$p_1 = 0+0+1+1 = 0$, $p_2 = 1+0+1+1 = 1$,
$p_4 = 0+1+1+1 = 1$

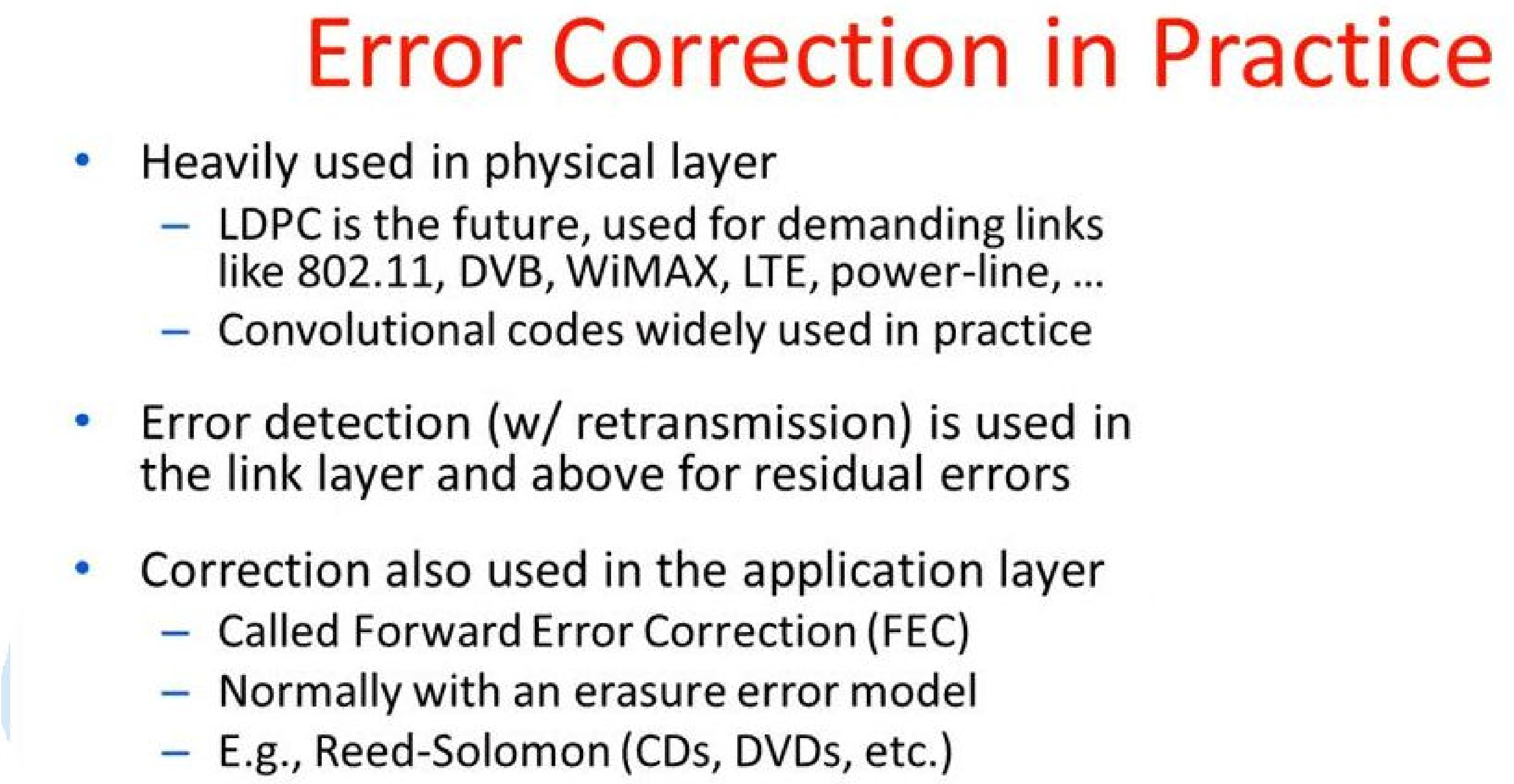
Syndrome = 1 1 0, flip position 6
Data = 0 1 0 1 (correct after flip!)



Detection vs. Correction (4)

- Error correction:
  - Needed when errors are expected
  - Or when no time for retransmission

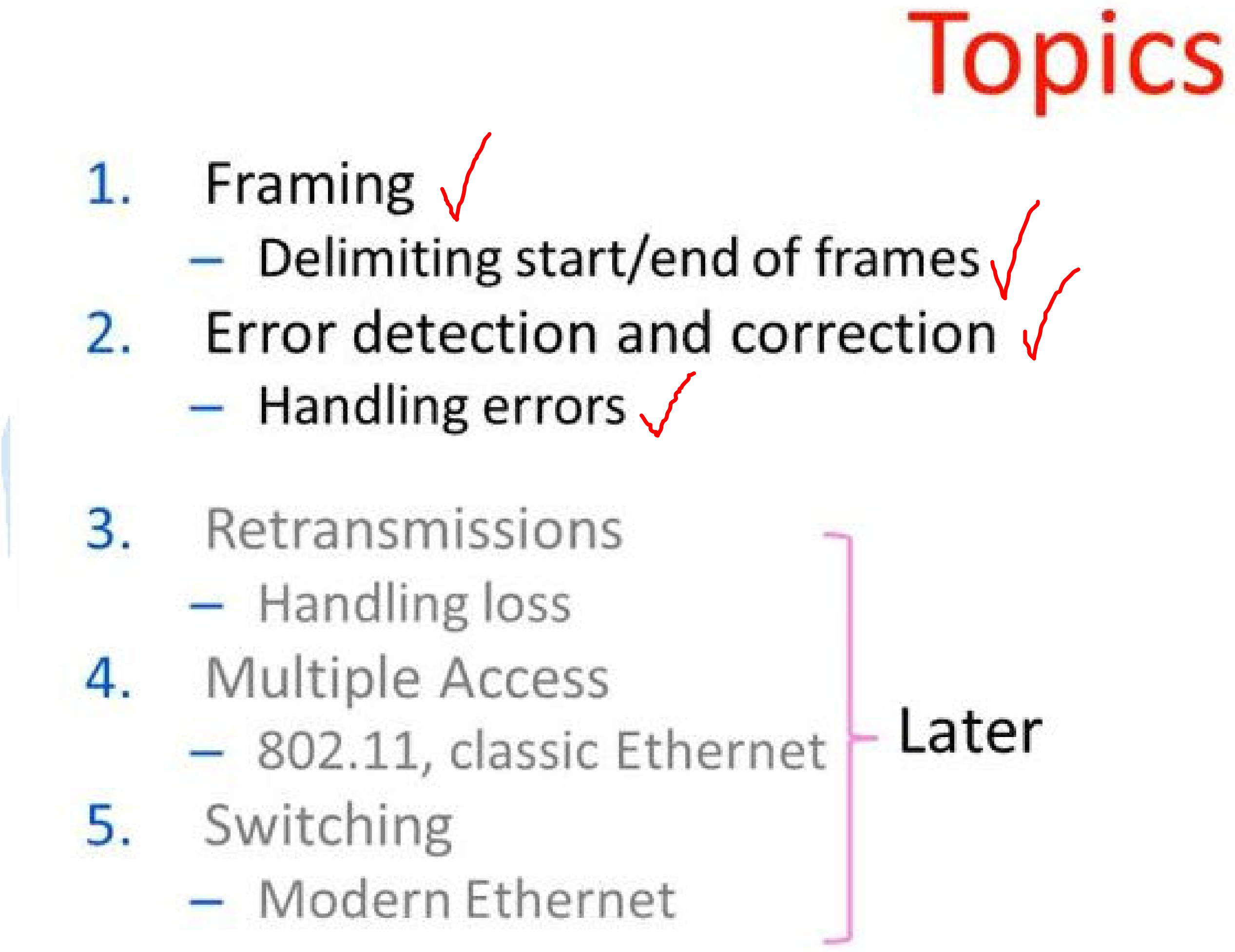
- Error detection:
  - More efficient when errors are not expected
  - And when errors are large when they do occur



Error Correction in Practice

- Heavily used in physical layer
  - LDPC is the future, used for demanding links like 802.11, DVB, WiMAX, LTE, power-line, …
  - Convolutional codes widely used in practice
- Error detection (w/ retransmission) is used in the link layer and above for residual errors
- Correction also used in the application layer
  - Called Forward Error Correction (FEC)
  - Normally with an erasure error model
  - E.g., Reed-Solomon (CDs, DVDs, etc.)

Main topics for Lecture 1:



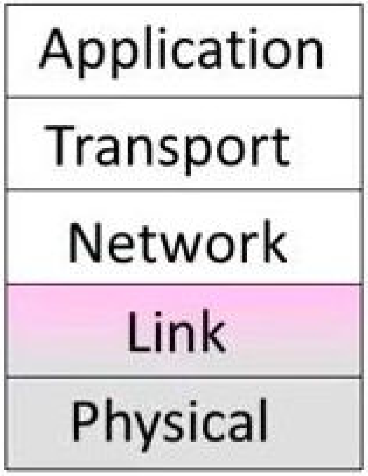
Topics

1. Framing ✓
   - Delimiting start/end of frames ✓
2. Error detection and correction ✓
   - Handling errors ✓
3. Retransmissions
   - Handling loss
4. Multiple Access
   - 802.11, classic Ethernet
5. Switching
   - Modern Ethernet

Later

Where we are in the Course

- Finishing off the Link Layer!
  - Builds on the physical layer to transfer frames over connected links



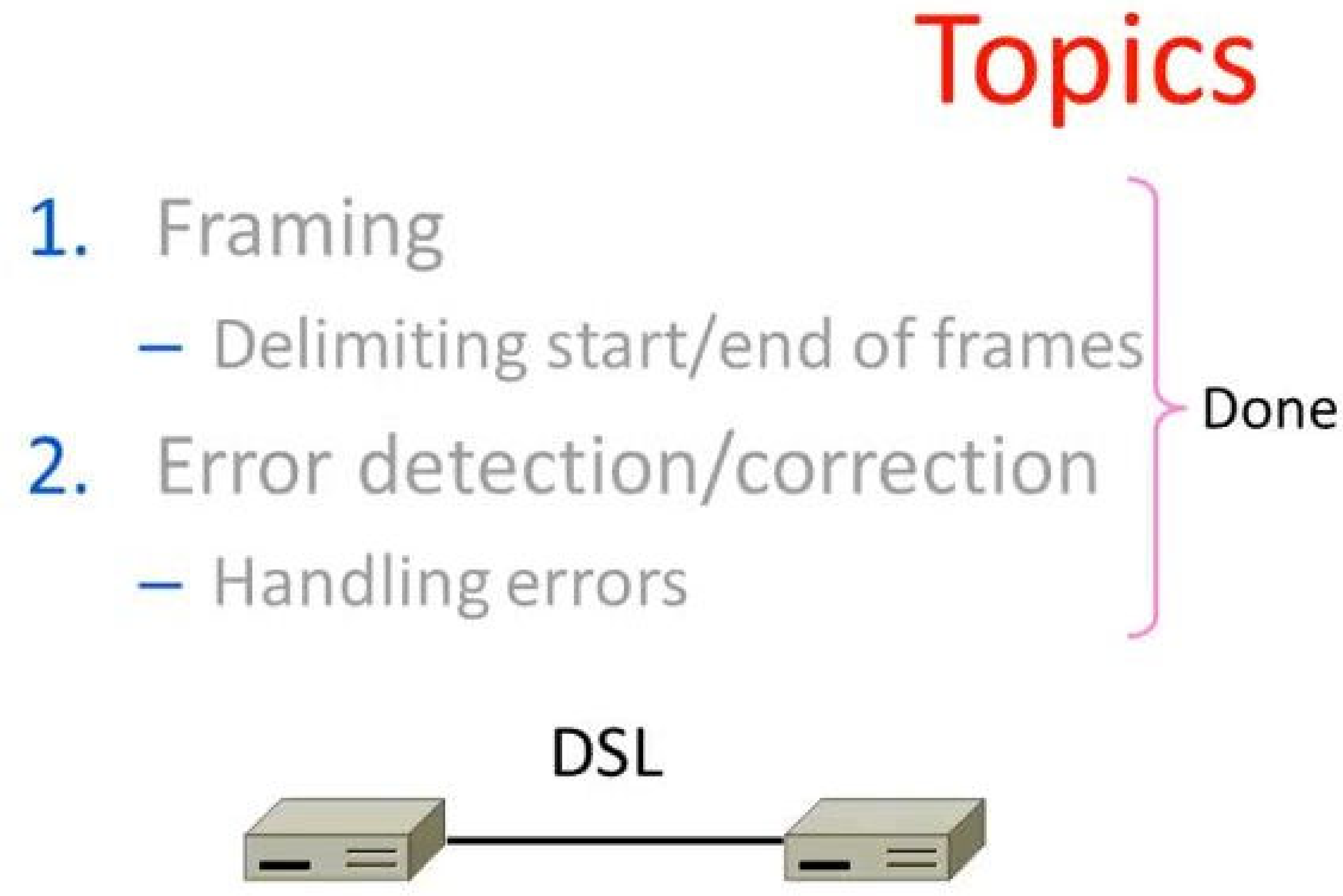
Application

Transport

Network

Link

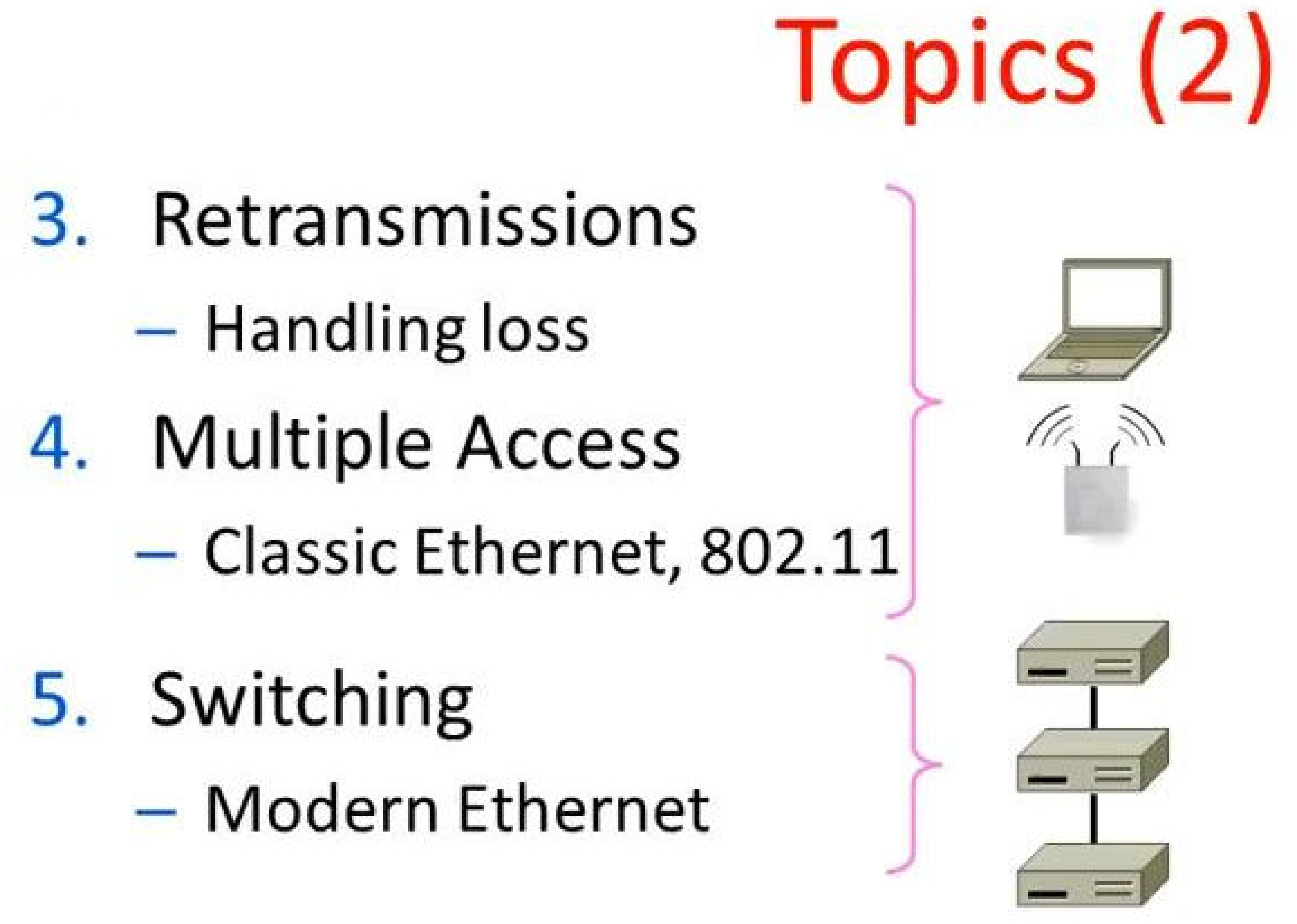
Physical



Topics

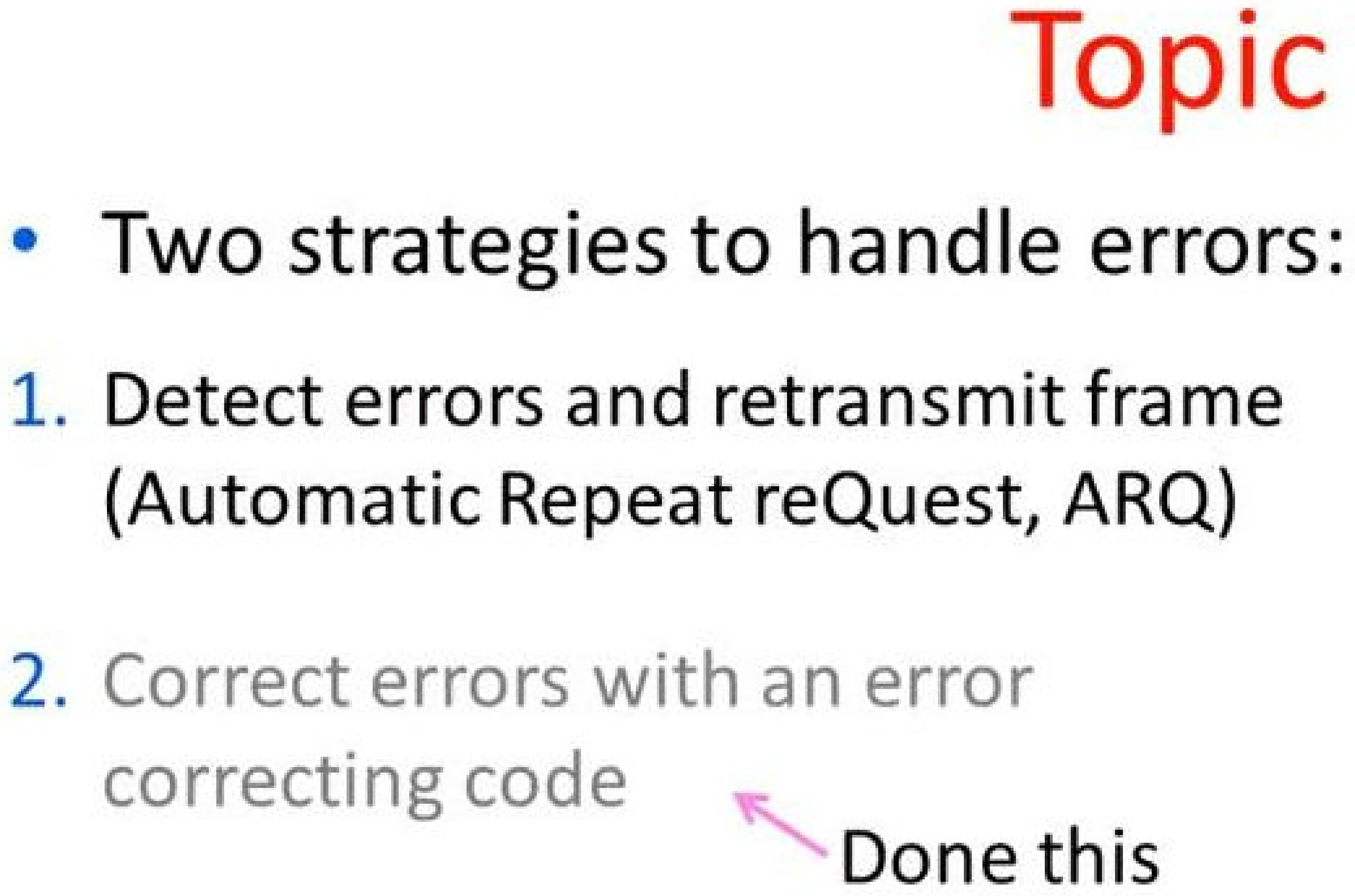
1. Framing
   - Delimiting start/end of frames
2. Error detection/correction
   - Handling errors

Done



Topics (2)

3. Retransmissions
   - Handling loss
4. Multiple Access
   - Classic Ethernet, 802.11
5. Switching
   - Modern Ethernet



Topic

- Two strategies to handle errors:

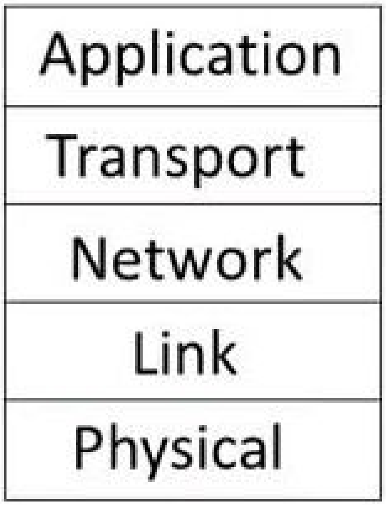
1. Detect errors and retransmit frame (Automatic Repeat reQuest, ARQ)

2. Correct errors with an error correcting code

Done this

Context on Reliability

- Where in the stack should we place reliability functions?



Application

Transport

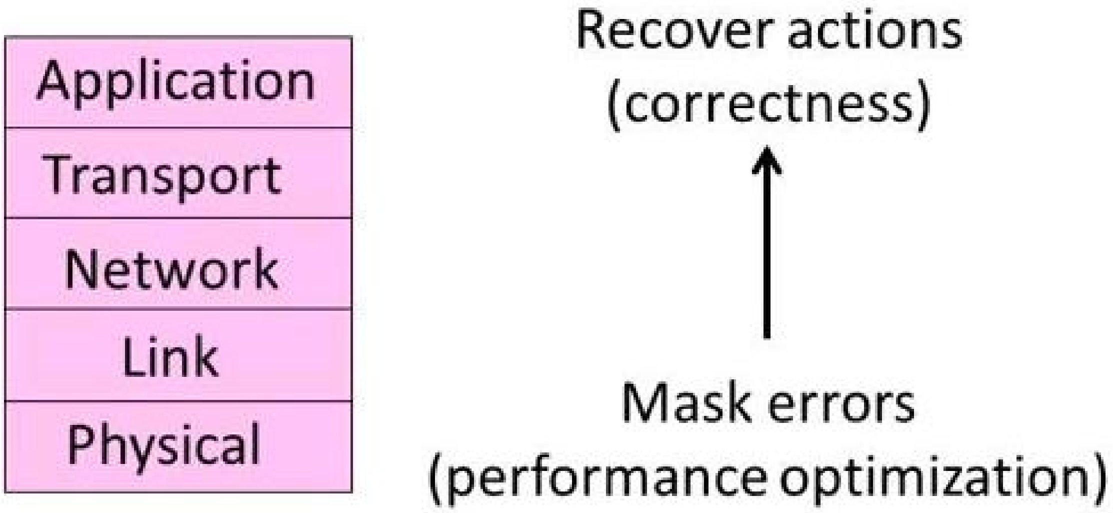
Network

Link

Physical

Context on Reliability (2)

- Everywhere! It is a key issue
  - Different layers contribute differently



Application

Transport

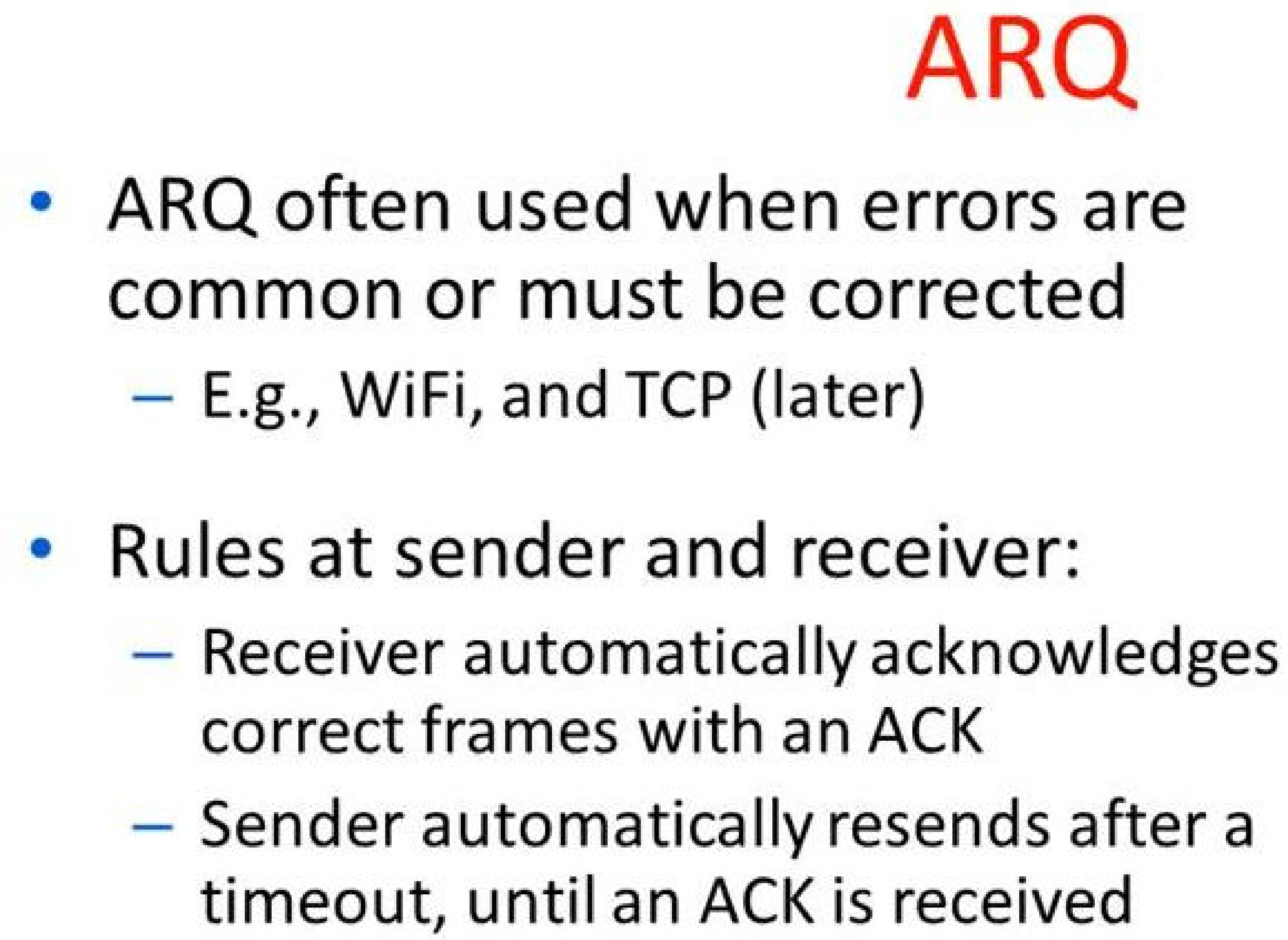
Network

Link

Physical

Recover actions (correctness)

Mask errors (performance optimization)

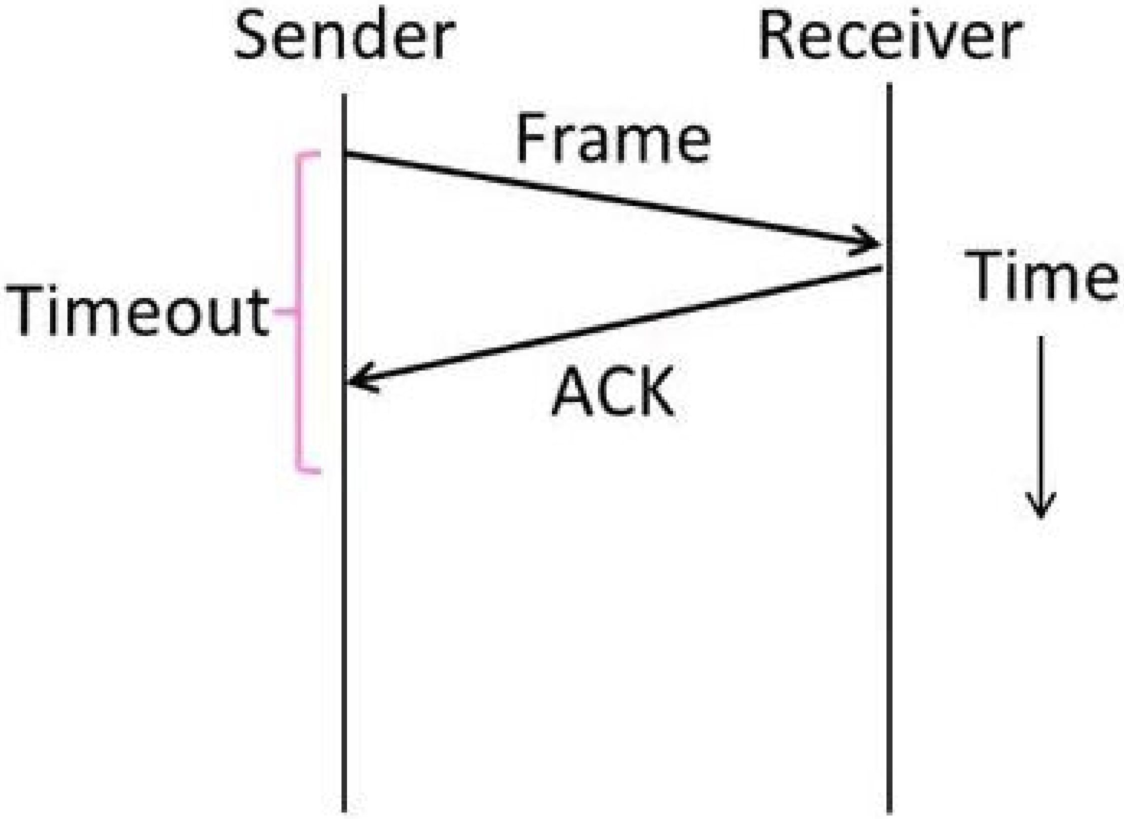


ARQ

- ARQ often used when errors are common or must be corrected
  - E.g., WiFi, and TCP (later)
- Rules at sender and receiver:
  - Receiver automatically acknowledges correct frames with an ACK
  - Sender automatically resends after a timeout, until an ACK is received

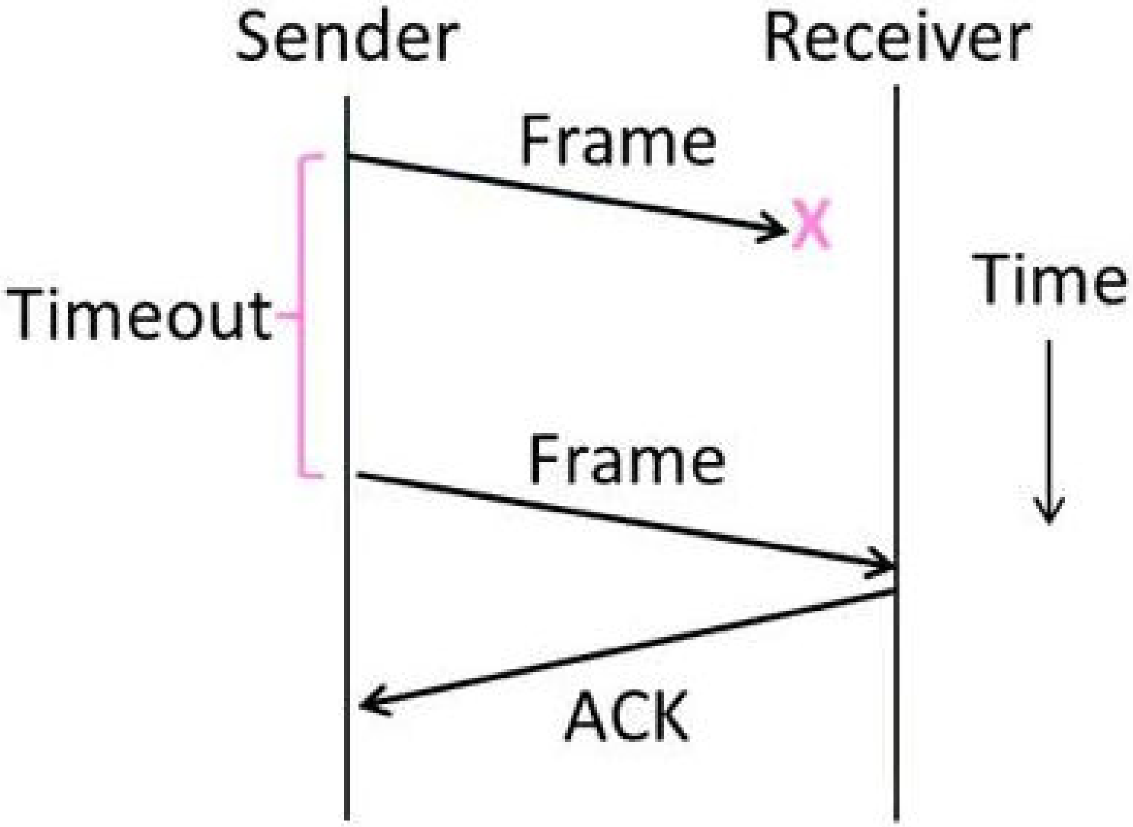
ARQ (2)

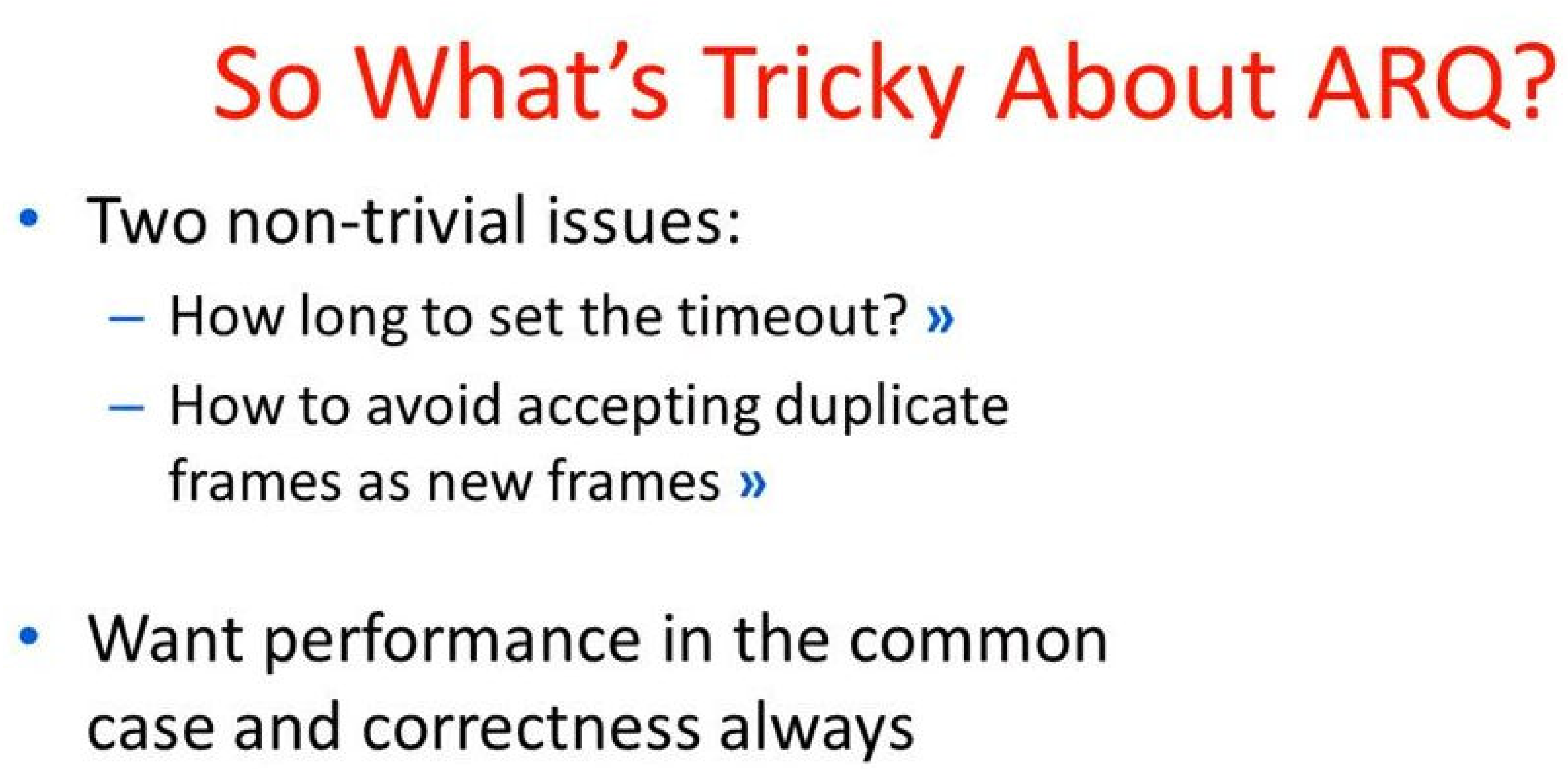
- Normal operation (no loss)



ARQ (3)

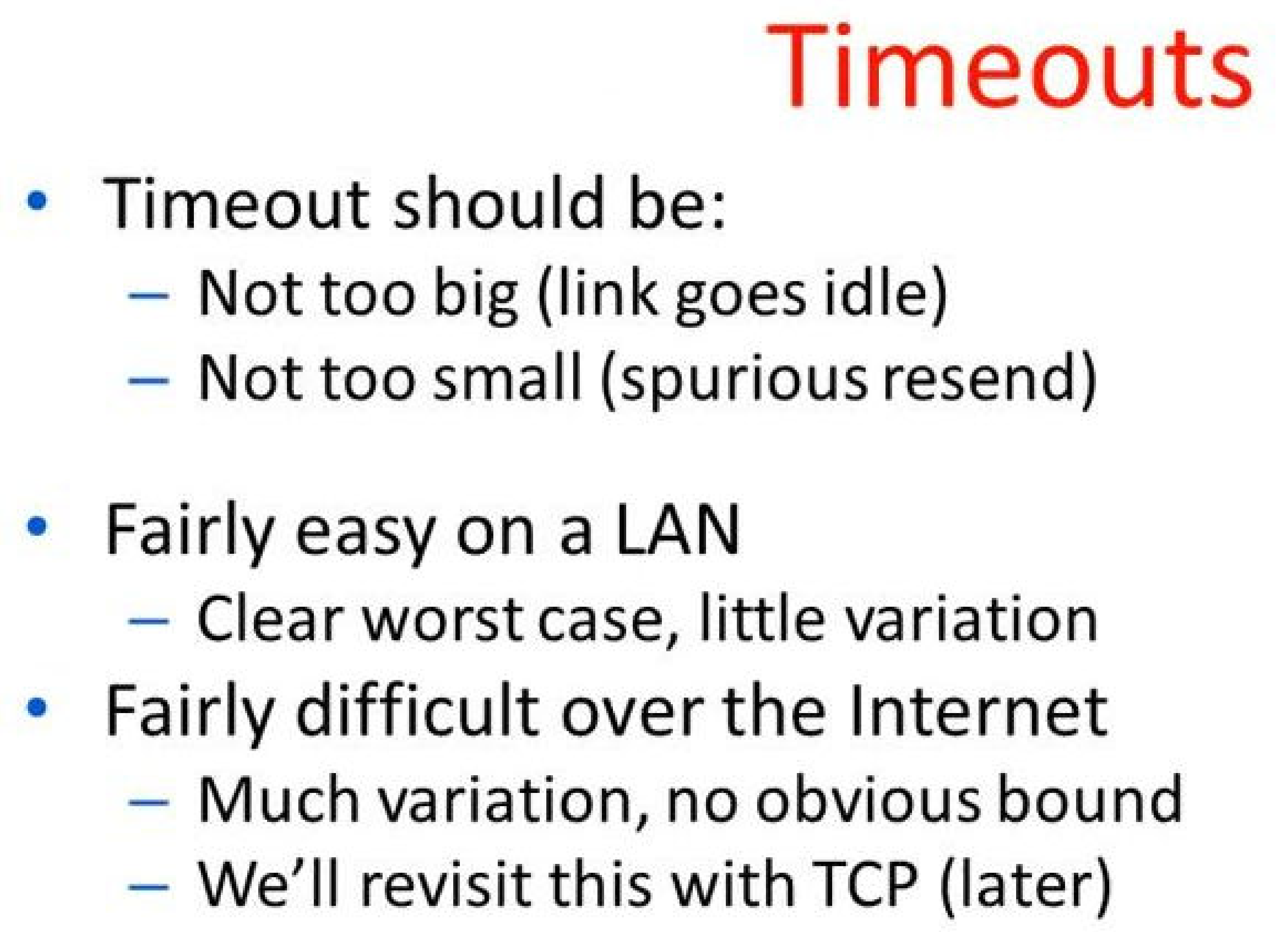
- Loss and retransmission





So What’s Tricky About ARQ?

- Two non-trivial issues:
  - How long to set the timeout? »
  - How to avoid accepting duplicate frames as new frames »
- Want performance in the common case and correctness always



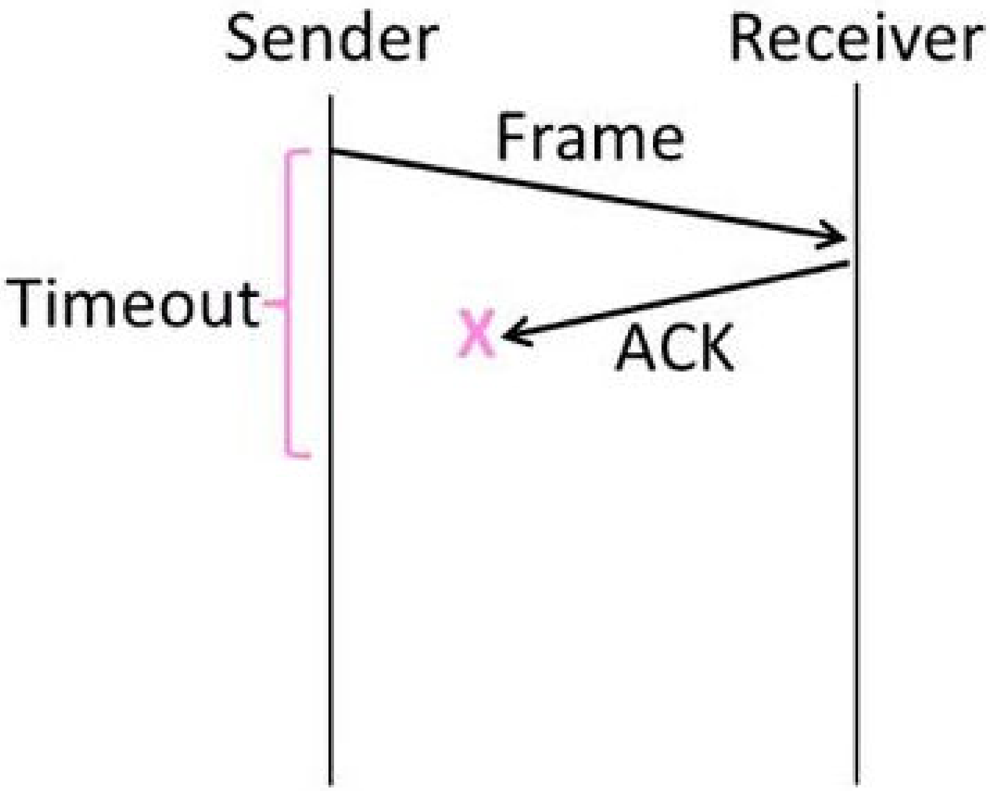
Timeouts

- Timeout should be:
  - Not too big (link goes idle)
  - Not too small (spurious resend)

- Fairly easy on a LAN
  - Clear worst case, little variation
- Fairly difficult over the Internet
  - Much variation, no obvious bound
  - We'll revisit this with TCP (later)

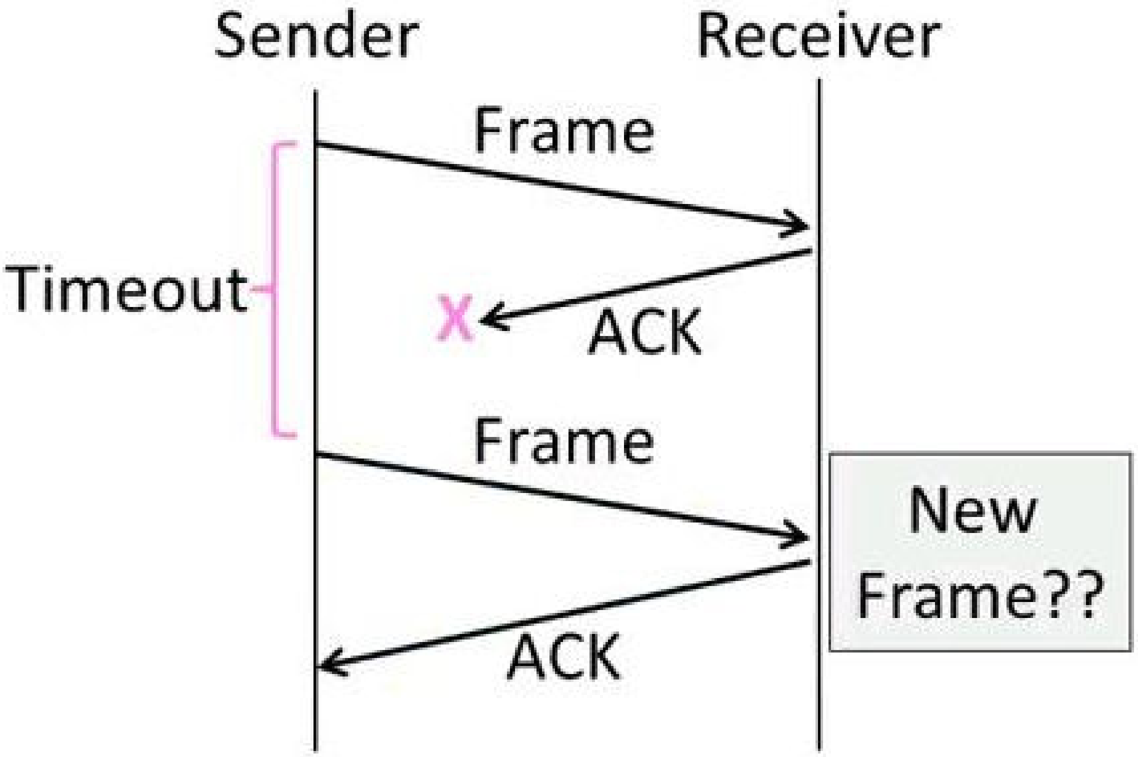
Duplicates

- What happens if an ACK is lost?



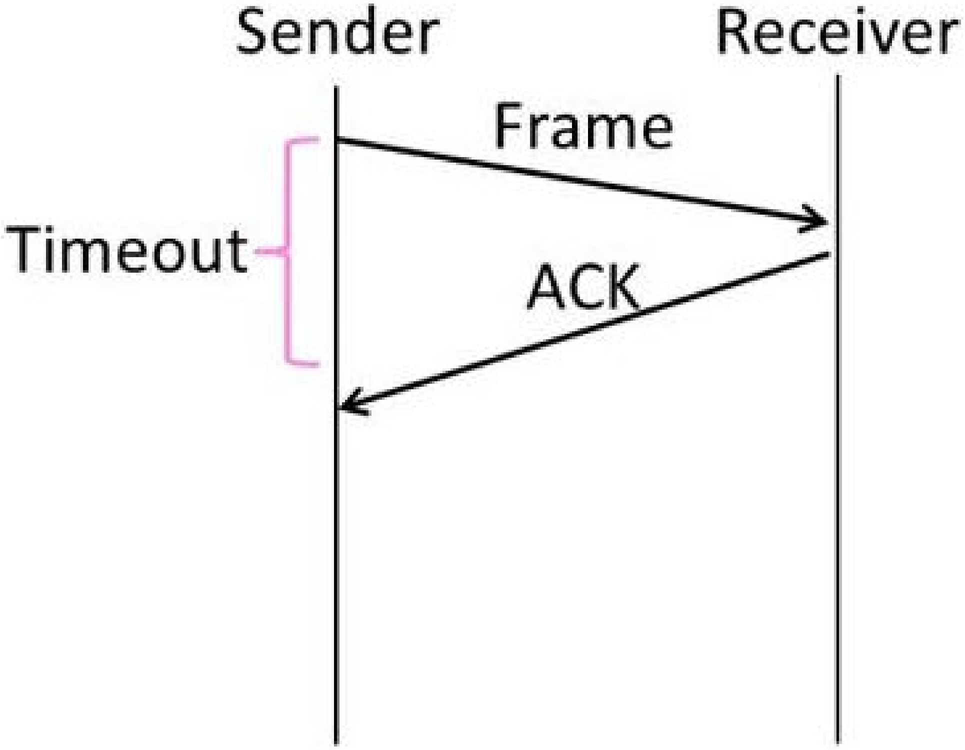
Duplicates (2)

- What happens if an ACK is lost?



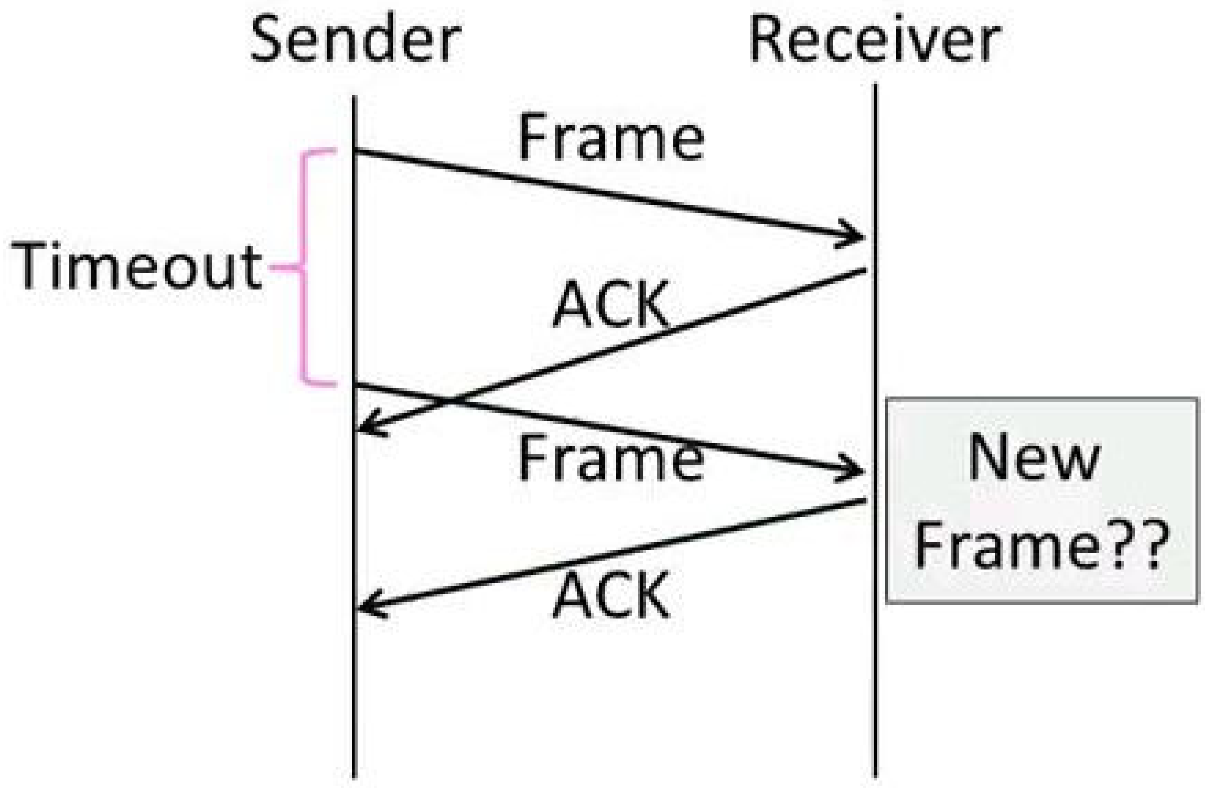
Duplicates (3)

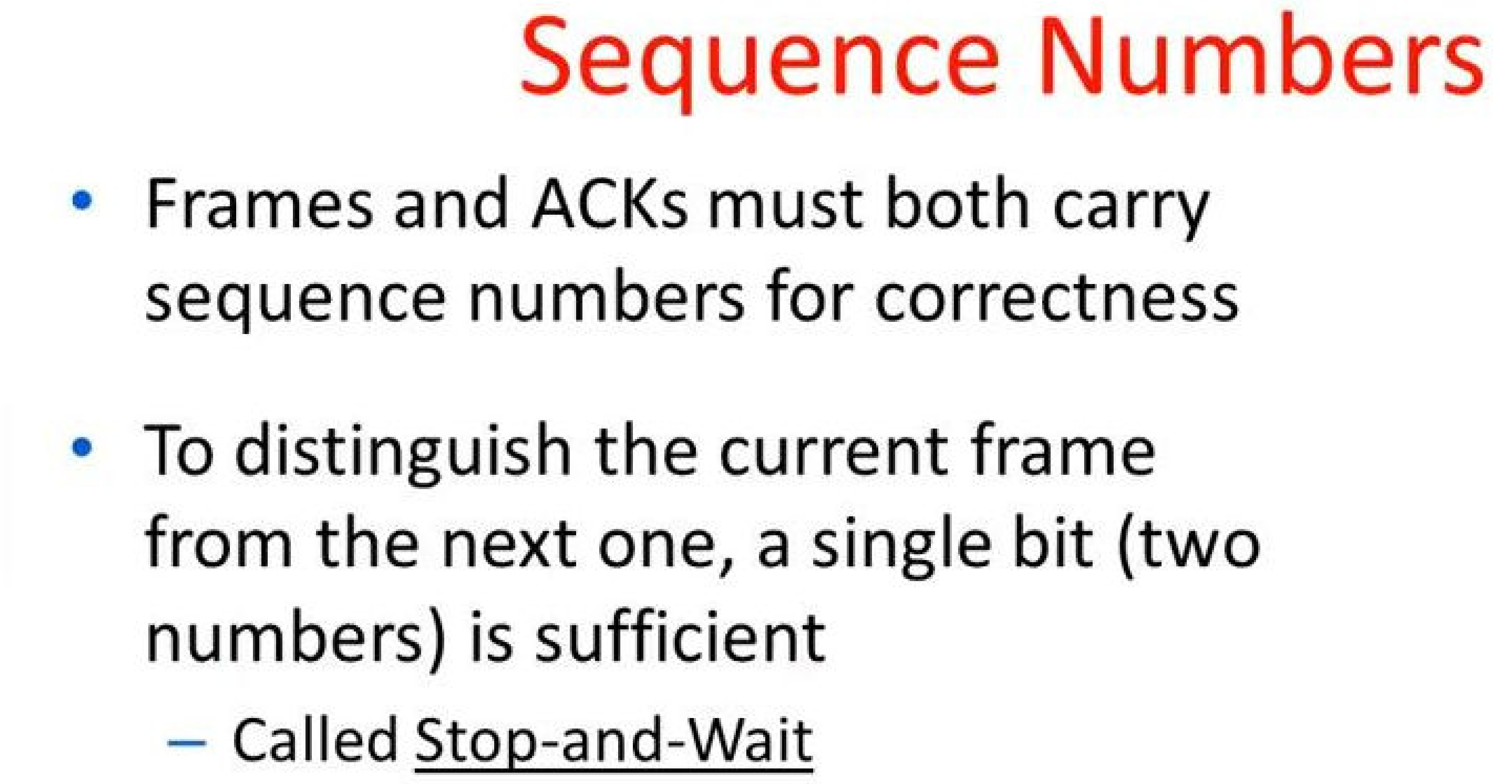
- Or the timeout is early?



Duplicates (4)

- Or the timeout is early?



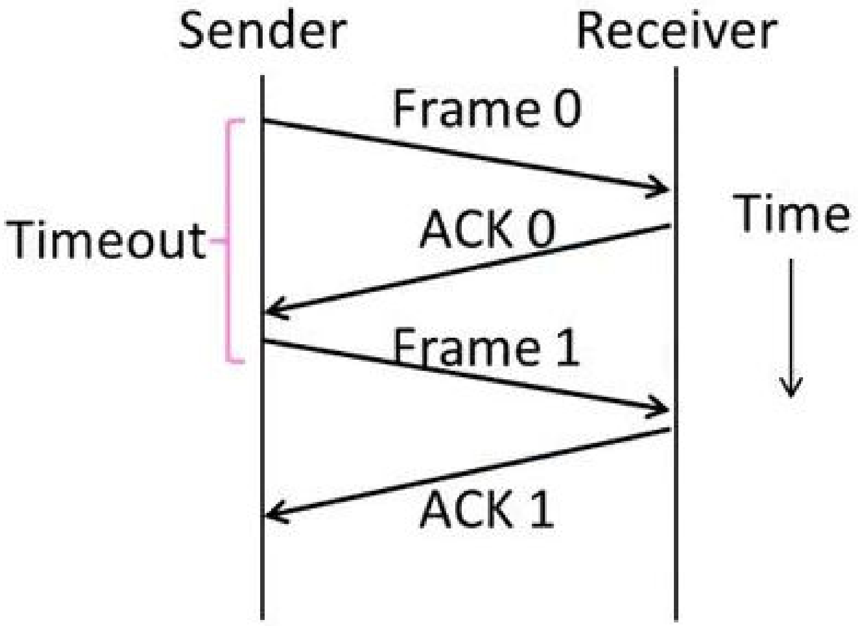


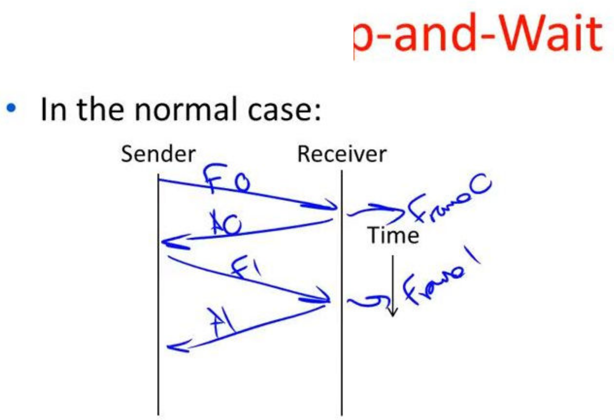
Sequence Numbers

- Frames and ACKs must both carry sequence numbers for correctness
- To distinguish the current frame from the next one, a single bit (two numbers) is sufficient
  - Called Stop-and-Wait

Stop-and-Wait (2)

- In the normal case:



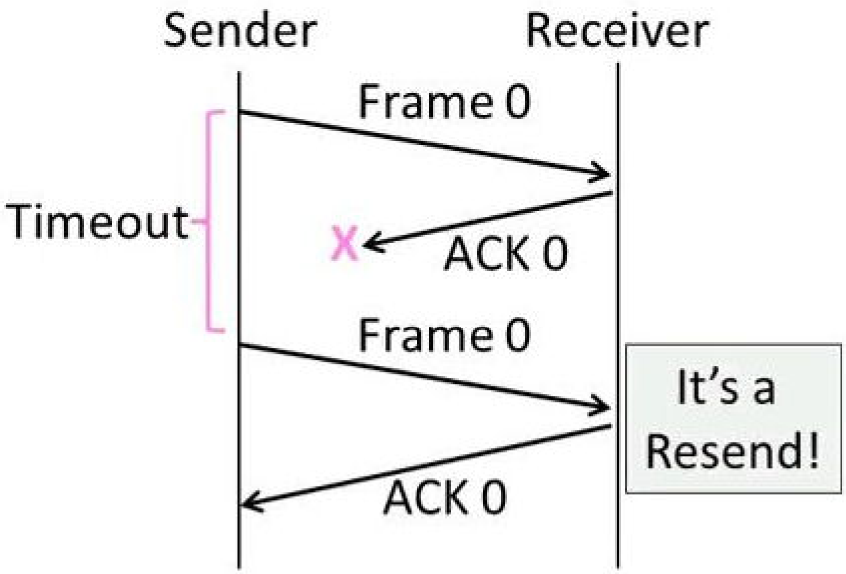


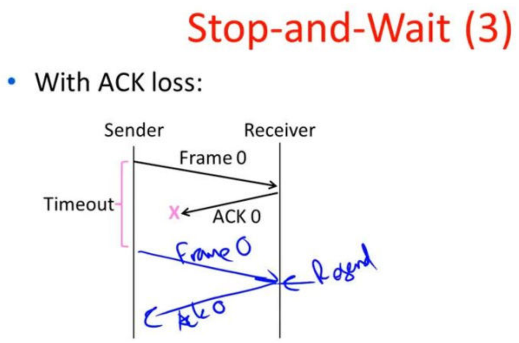
p-and-Wait

- In the normal case:

Stop-and-Wait (4)

- With ACK loss:

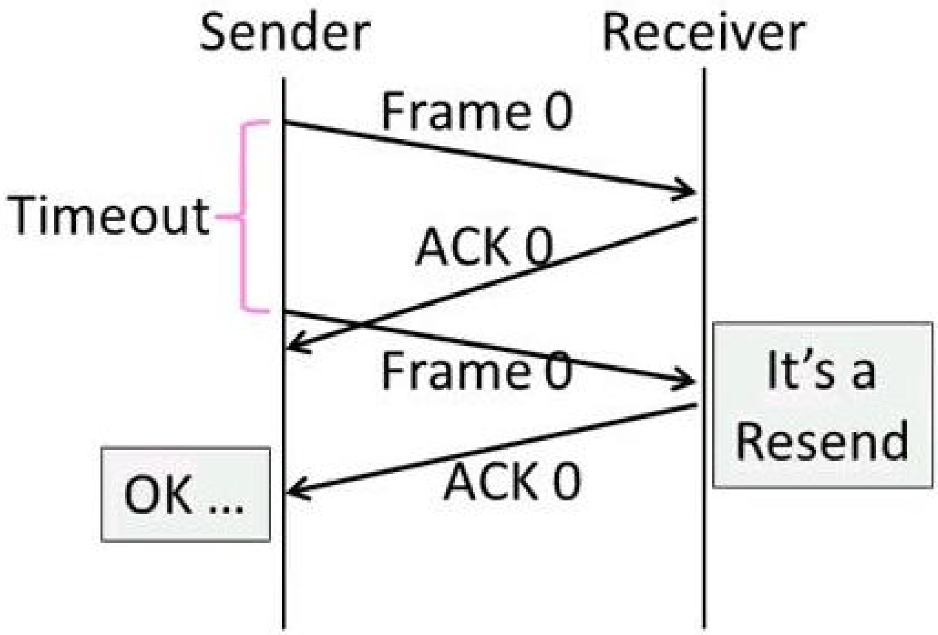


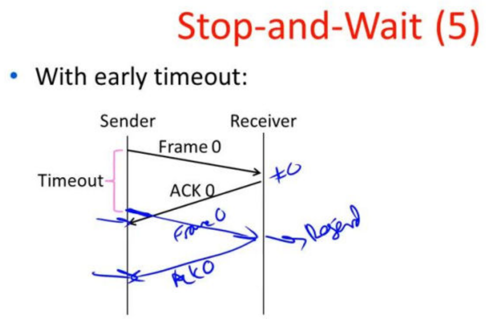

Stop-and-Wait (3)

- With ACK loss:

Stop-and-Wait (6)

- With early timeout:



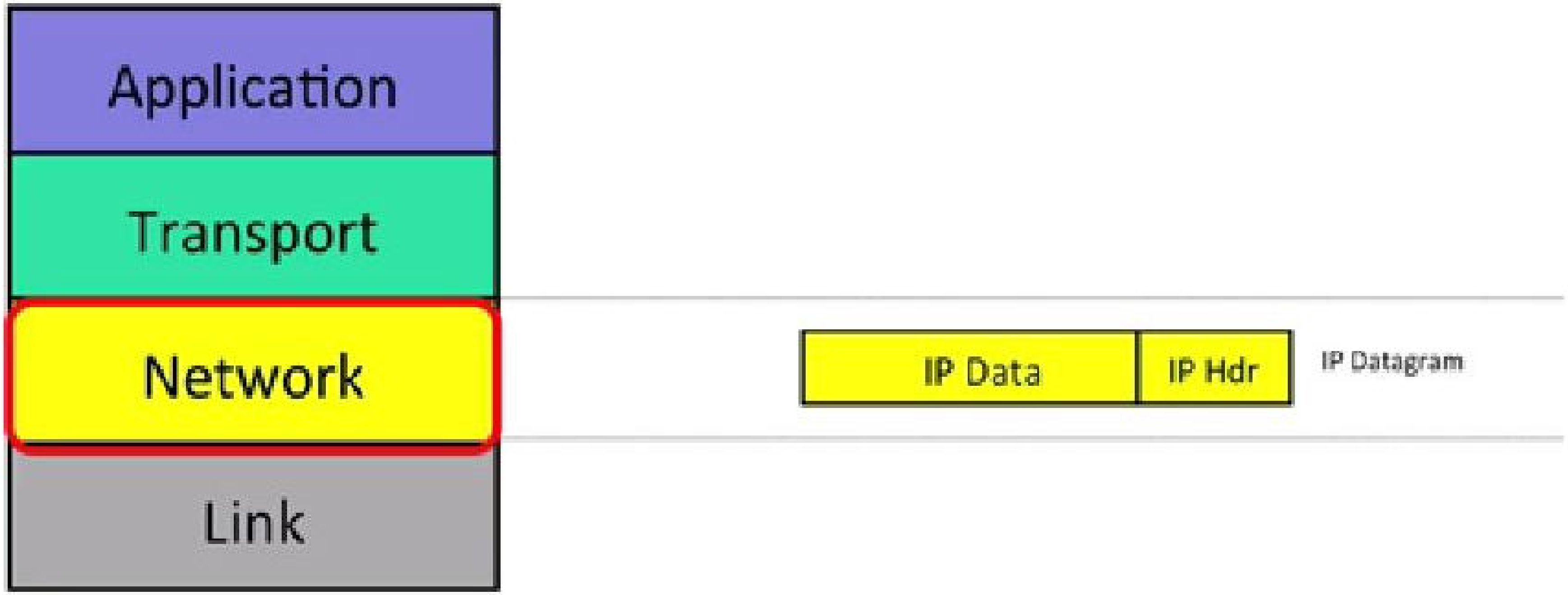


Stop-and-Wait (5)

- With early timeout:

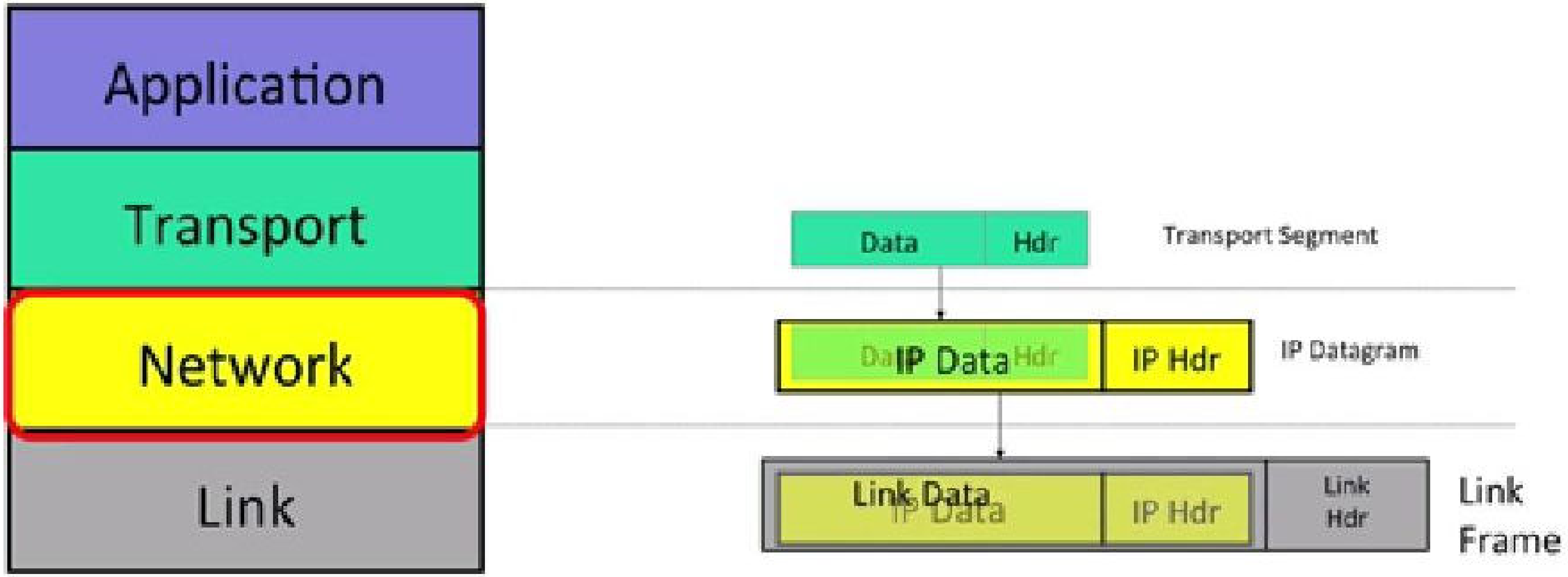
The IP service model

The Internet Protocol (IP)

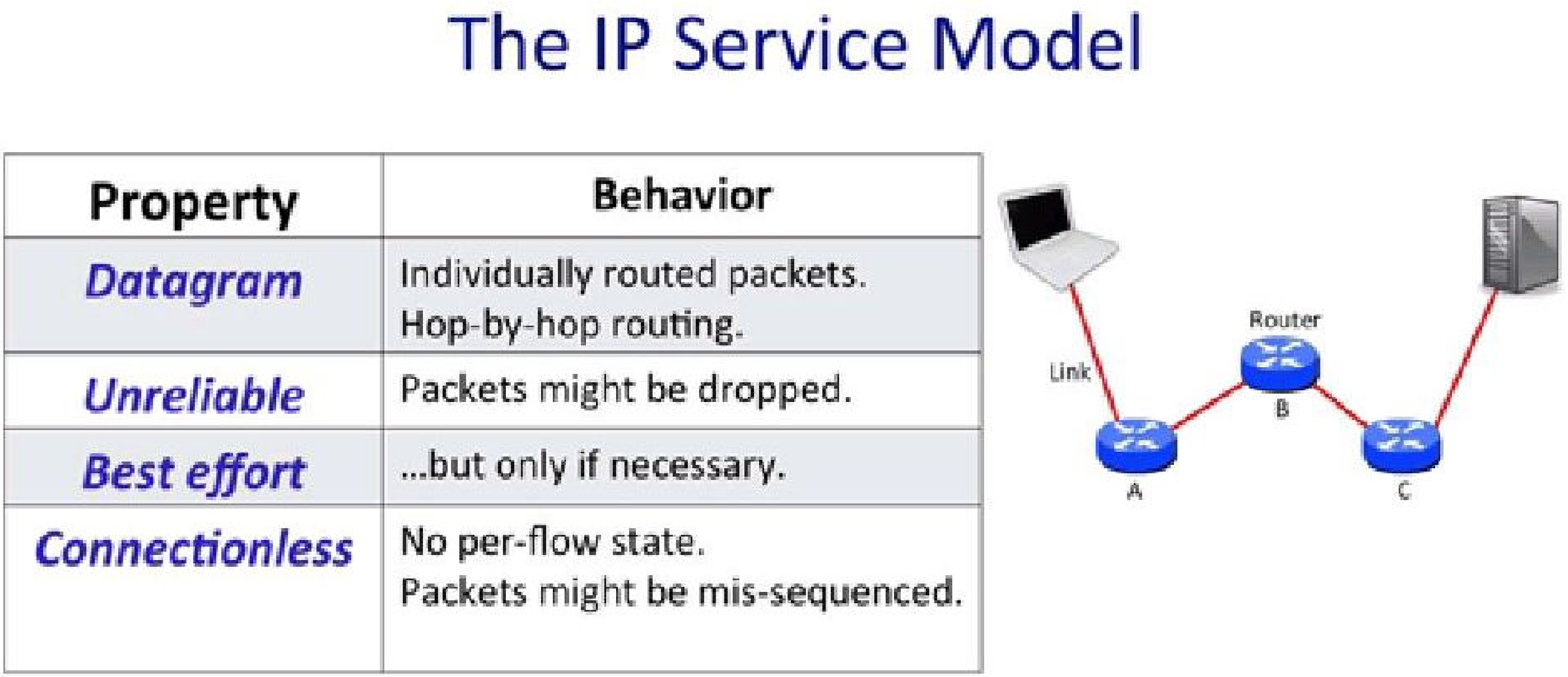


The IP service model

The Internet Protocol (IP)



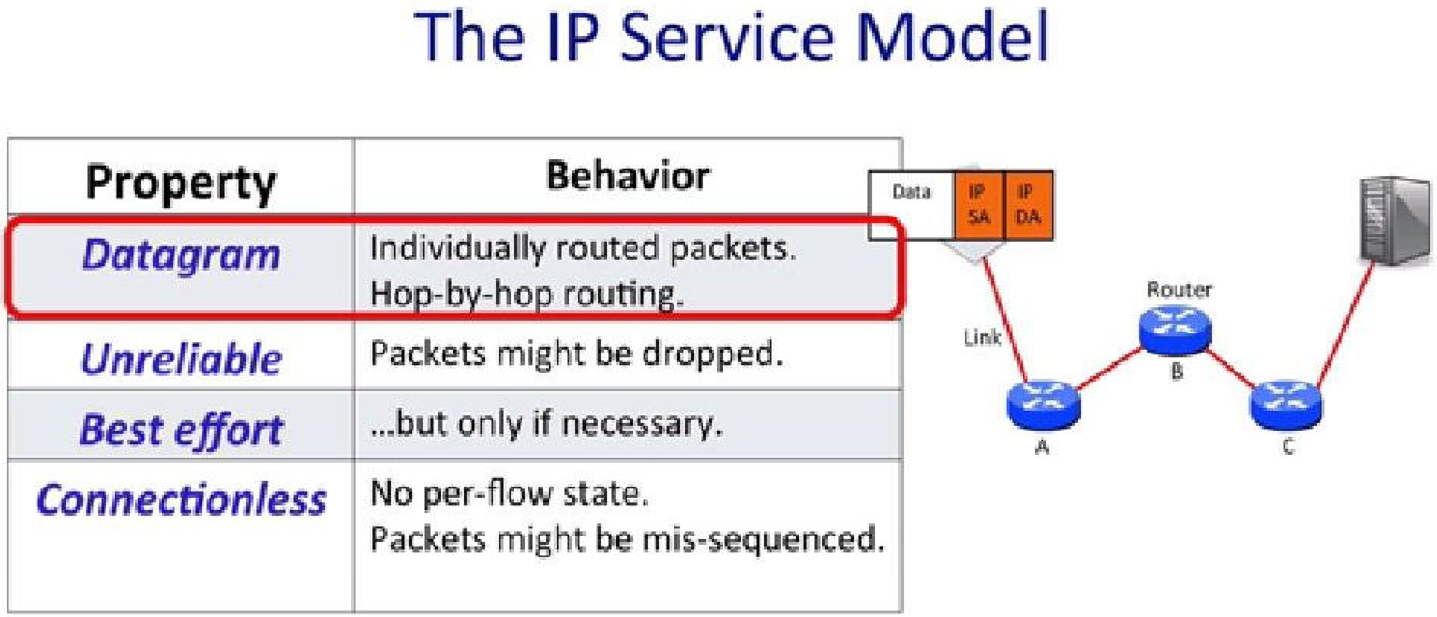
The IP service model



The IP Service Model

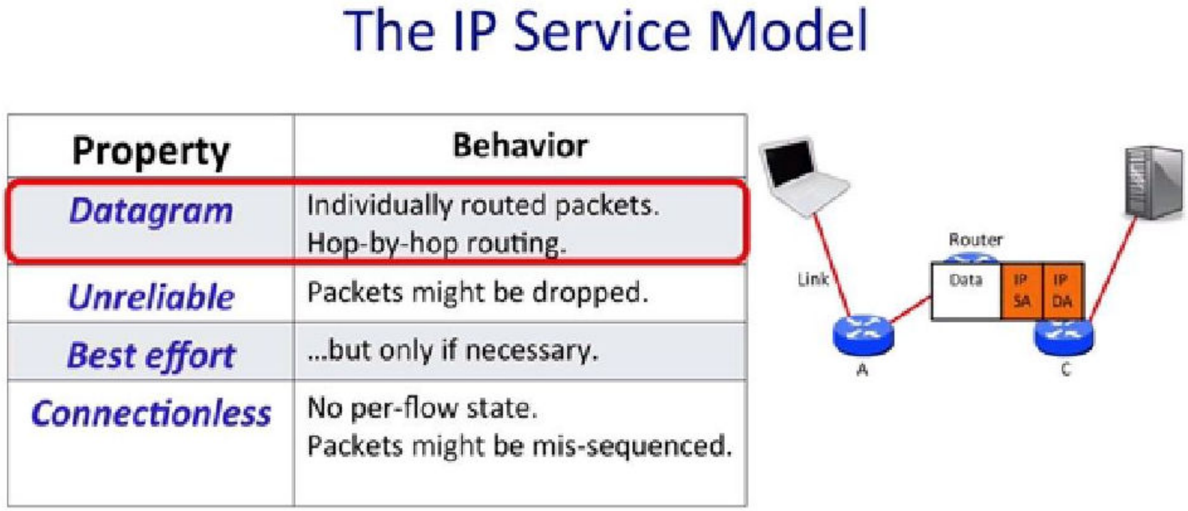
Property	Behavior
Datagram	Individually routed packets. Hop-by-hop routing.
Unreliable	Packets might be dropped.
Best effort	...but only if necessary.
Connectionless	No per-flow state. Packets might be mis-sequenced.

The IP service model



The IP Service Model

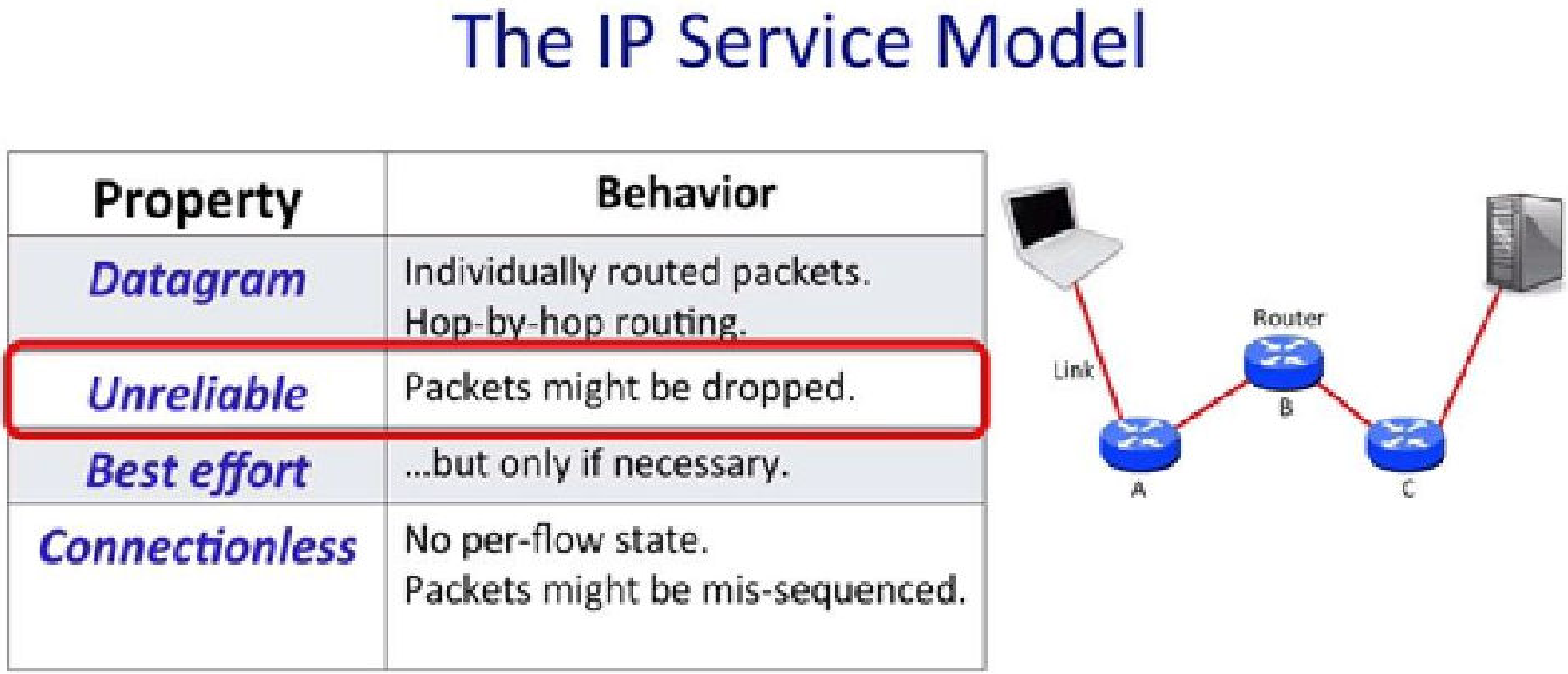
Property	Behavior
Datagram	Individually routed packets. Hop-by-hop routing.
Unreliable	Packets might be dropped.
Best effort	...but only if necessary.
Connectionless	No per-flow state. Packets might be mis-sequenced.



The IP Service Model

Property	Behavior
Datagram	Individually routed packets. Hop-by-hop routing.
Unreliable	Packets might be dropped.
Best effort	...but only if necessary.
Connectionless	No per-flow state. Packets might be mis-sequenced.

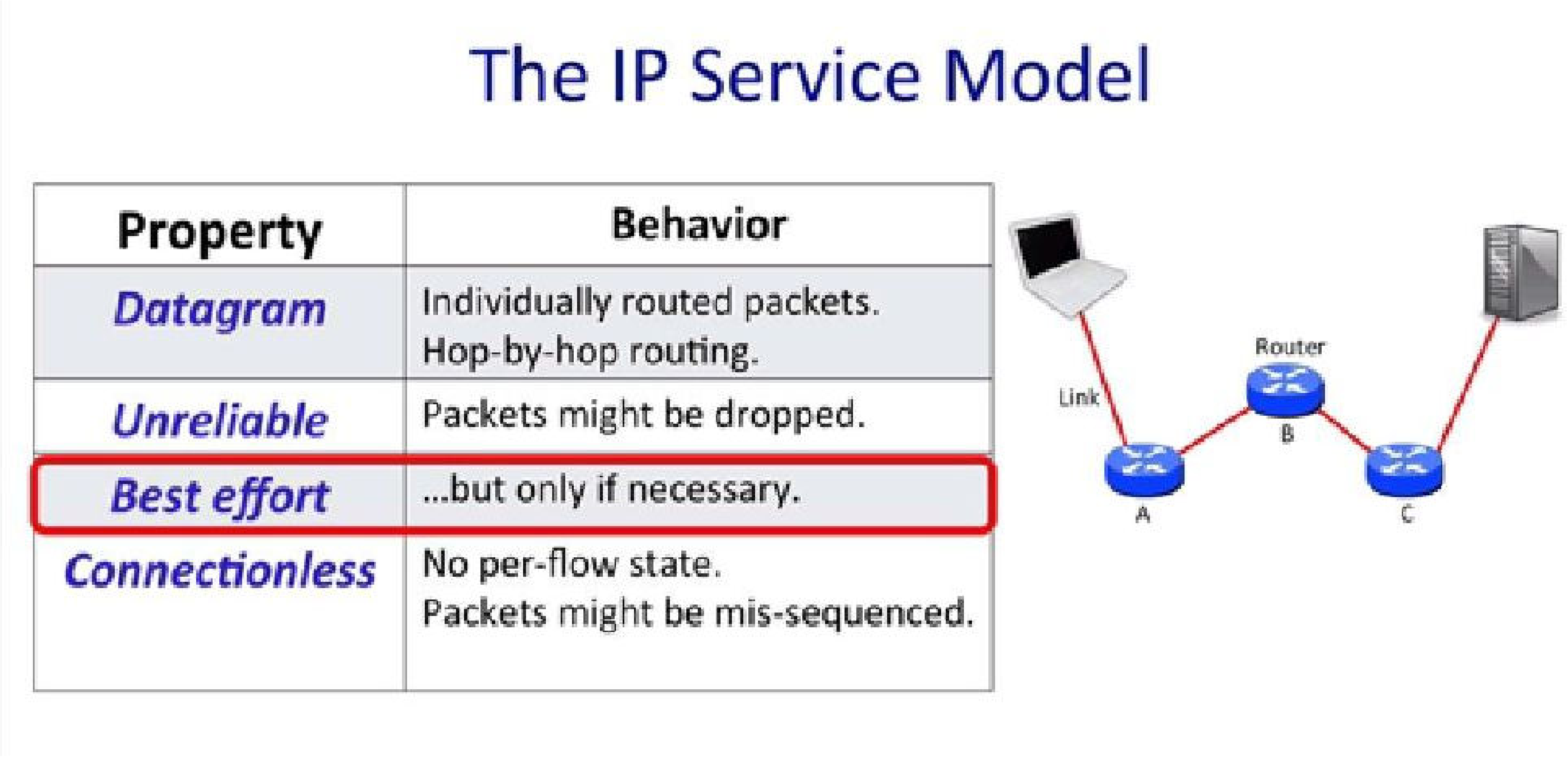
The IP service model



The IP Service Model

Property	Behavior
Datagram	Individually routed packets. Hop-by-hop routing.
Unreliable	Packets might be dropped.
Best effort	...but only if necessary.
Connectionless	No per-flow state. Packets might be mis-sequenced.

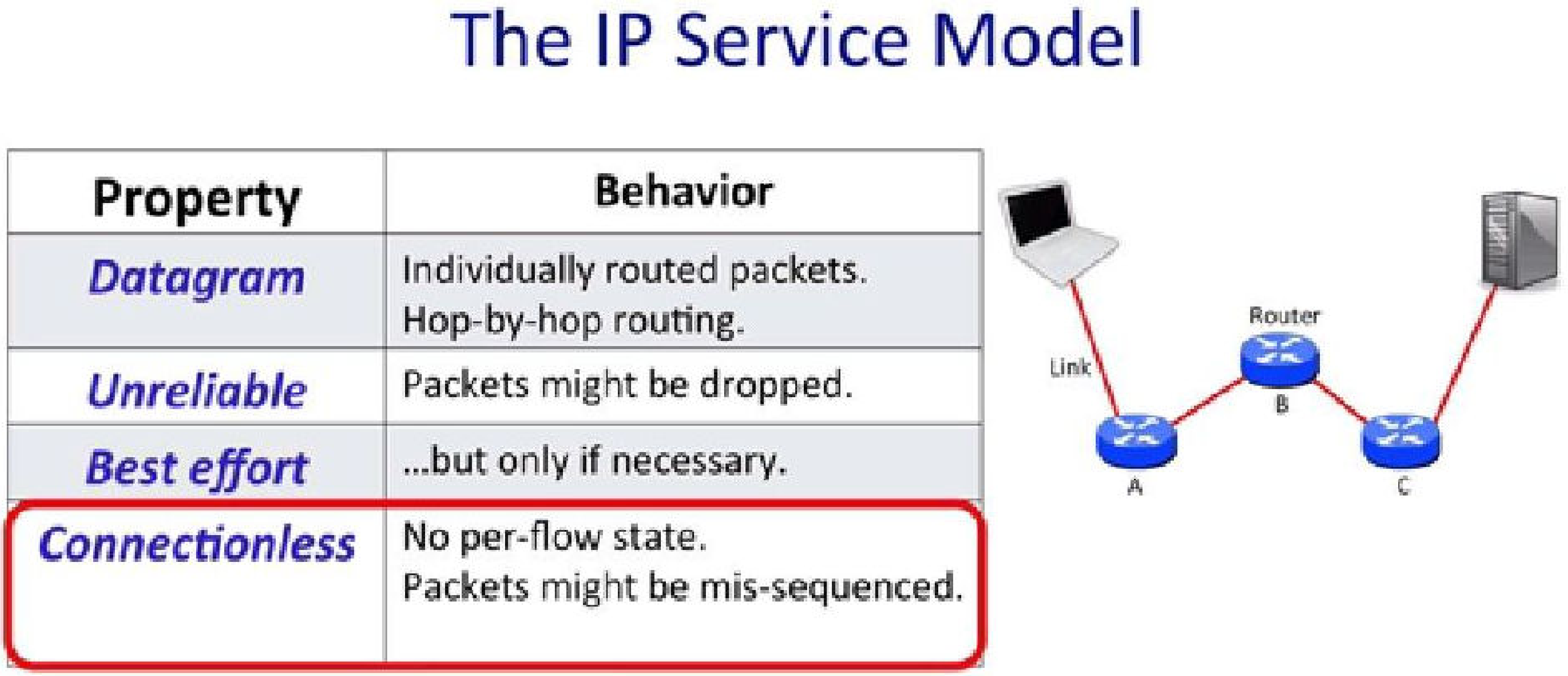
The IP service model



The IP Service Model

Property	Behavior
Datagram	Individually routed packets. Hop-by-hop routing.
Unreliable	Packets might be dropped.
Best effort	...but only if necessary.
Connectionless	No per-flow state. Packets might be mis-sequenced.

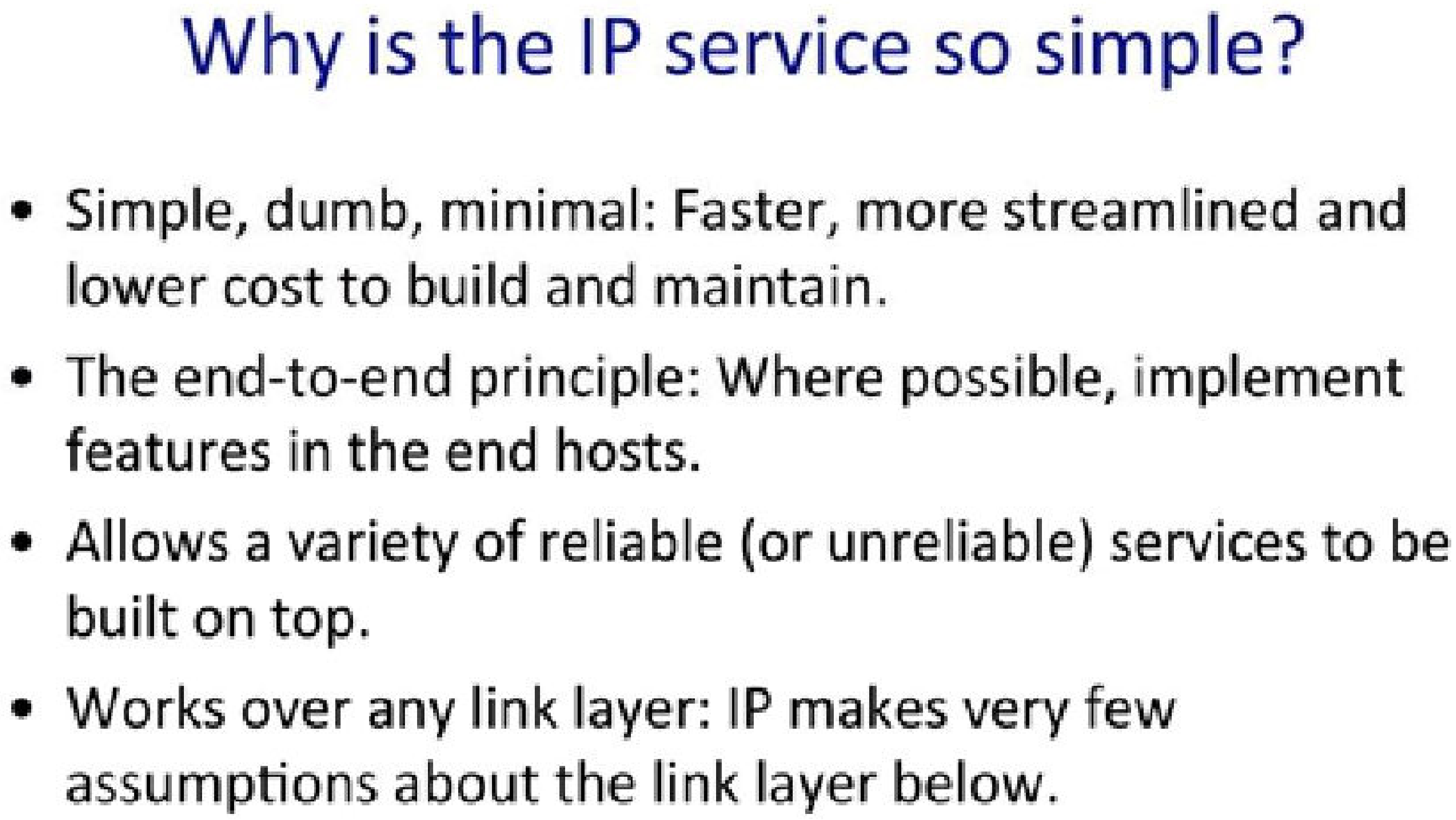
The IP service model



The IP Service Model

Property	Behavior
Datagram	Individually routed packets. Hop-by-hop routing.
Unreliable	Packets might be dropped.
Best effort	...but only if necessary.
Connectionless	No per-flow state. Packets might be mis-sequenced.

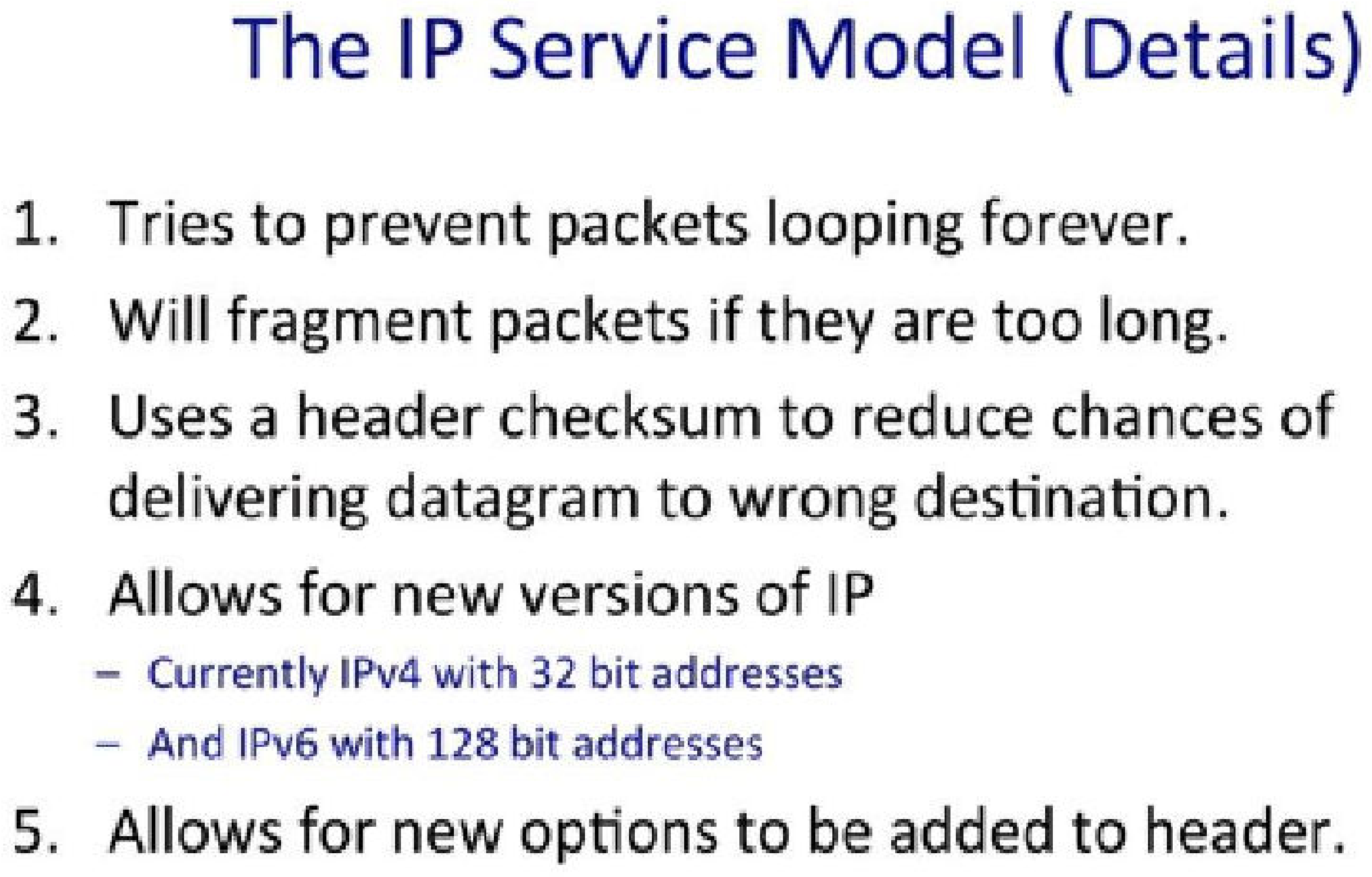
The IP service model



Why is the IP service so simple?

- Simple, dumb, minimal: Faster, more streamlined and lower cost to build and maintain.
- The end-to-end principle: Where possible, implement features in the end hosts.
- Allows a variety of reliable (or unreliable) services to be built on top.
- Works over any link layer: IP makes very few assumptions about the link layer below.

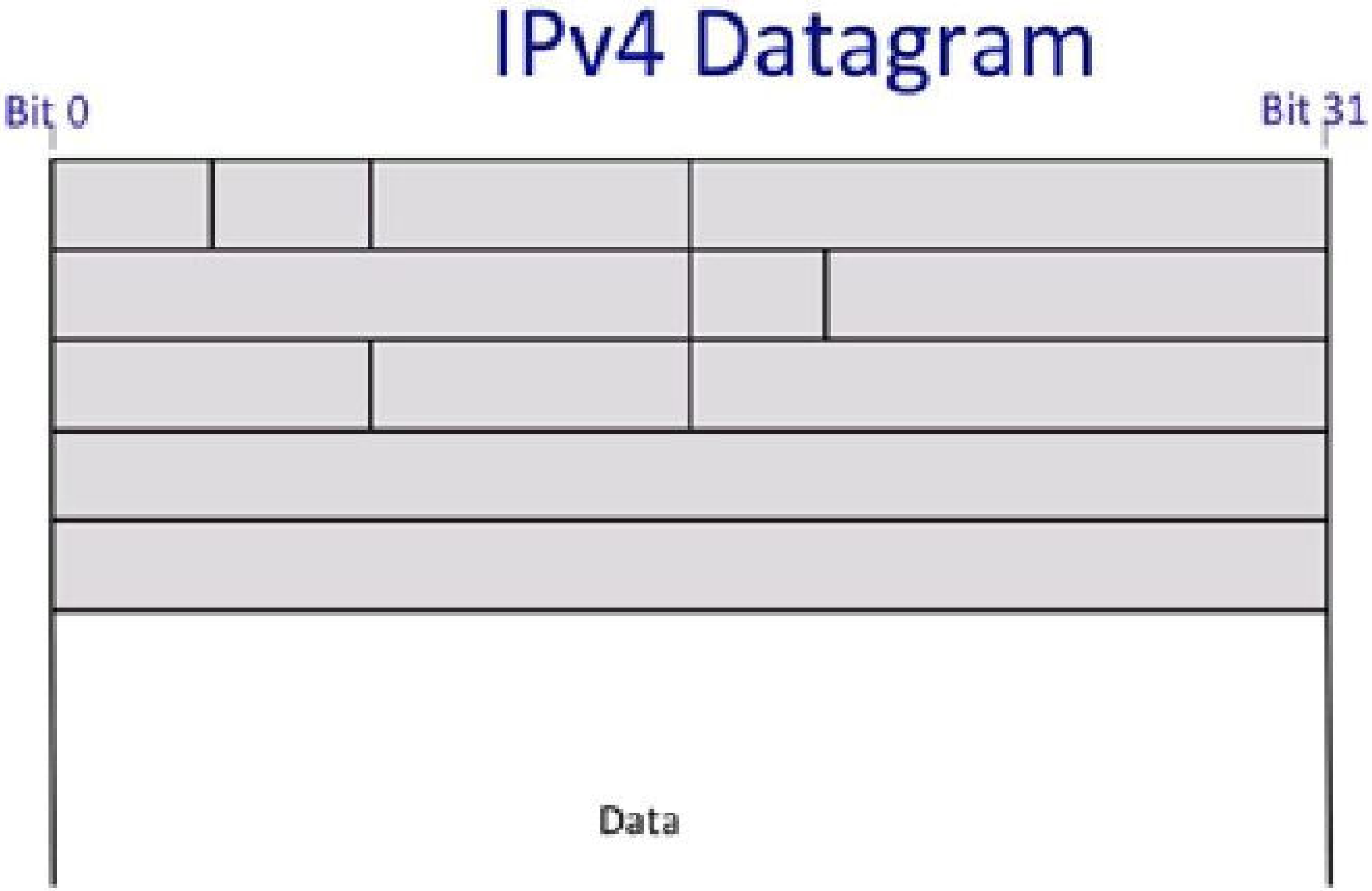
The IP service model



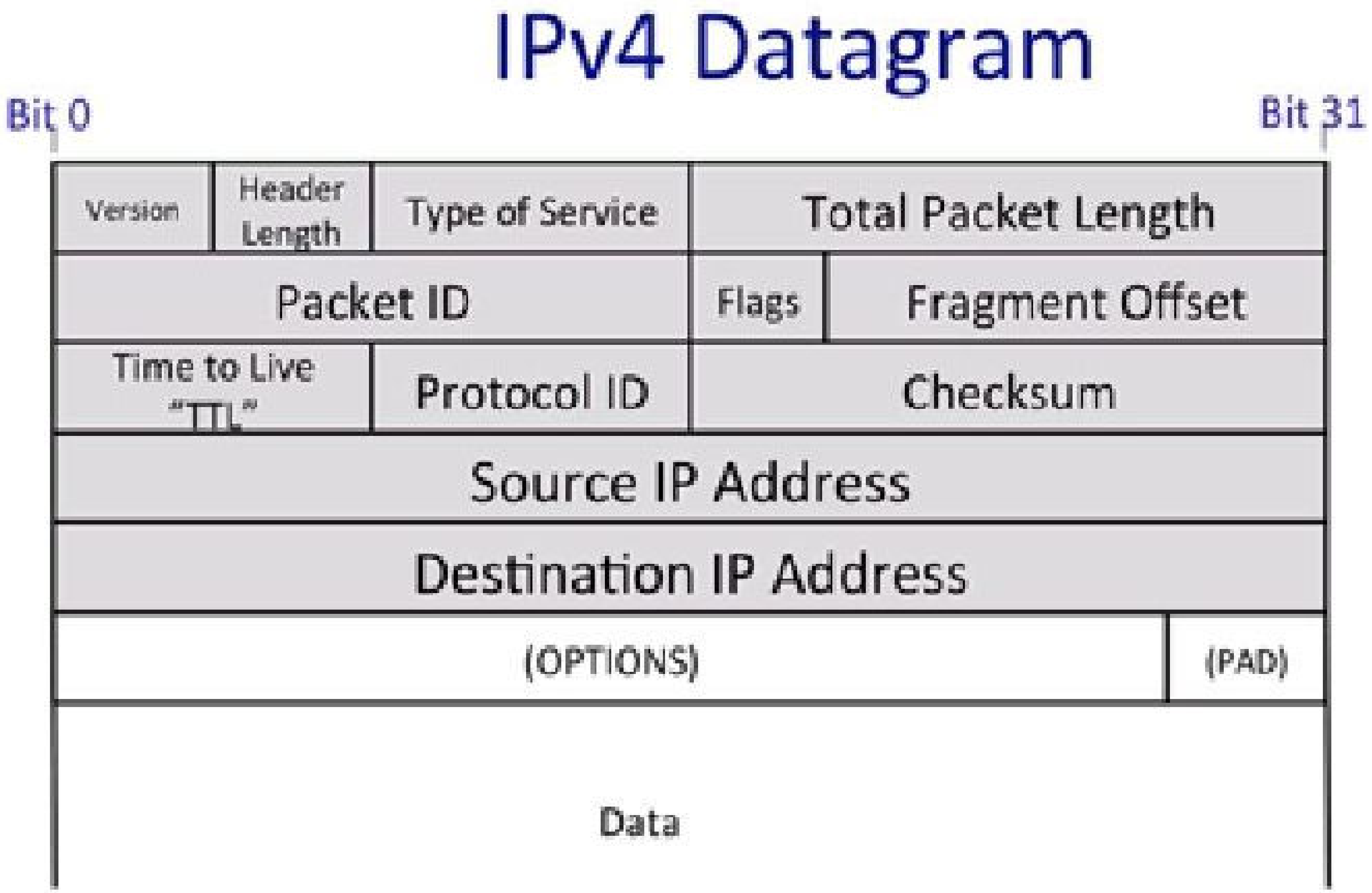
The IP Service Model (Details)

1. Tries to prevent packets looping forever.
2. Will fragment packets if they are too long.
3. Uses a header checksum to reduce chances of delivering datagram to wrong destination.
4. Allows for new versions of IP
   - Currently IPv4 with 32 bit addresses
   - And IPv6 with 128 bit addresses
5. Allows for new options to be added to header.

The IP service model



The IP service model



IPv4 Datagram

Bit 0

Bit 31

Version | Header Length | Type of Service | Total Packet Length

Packet ID | Flags | Fragment Offset

Time to Live "TTL" | Protocol ID | Checksum

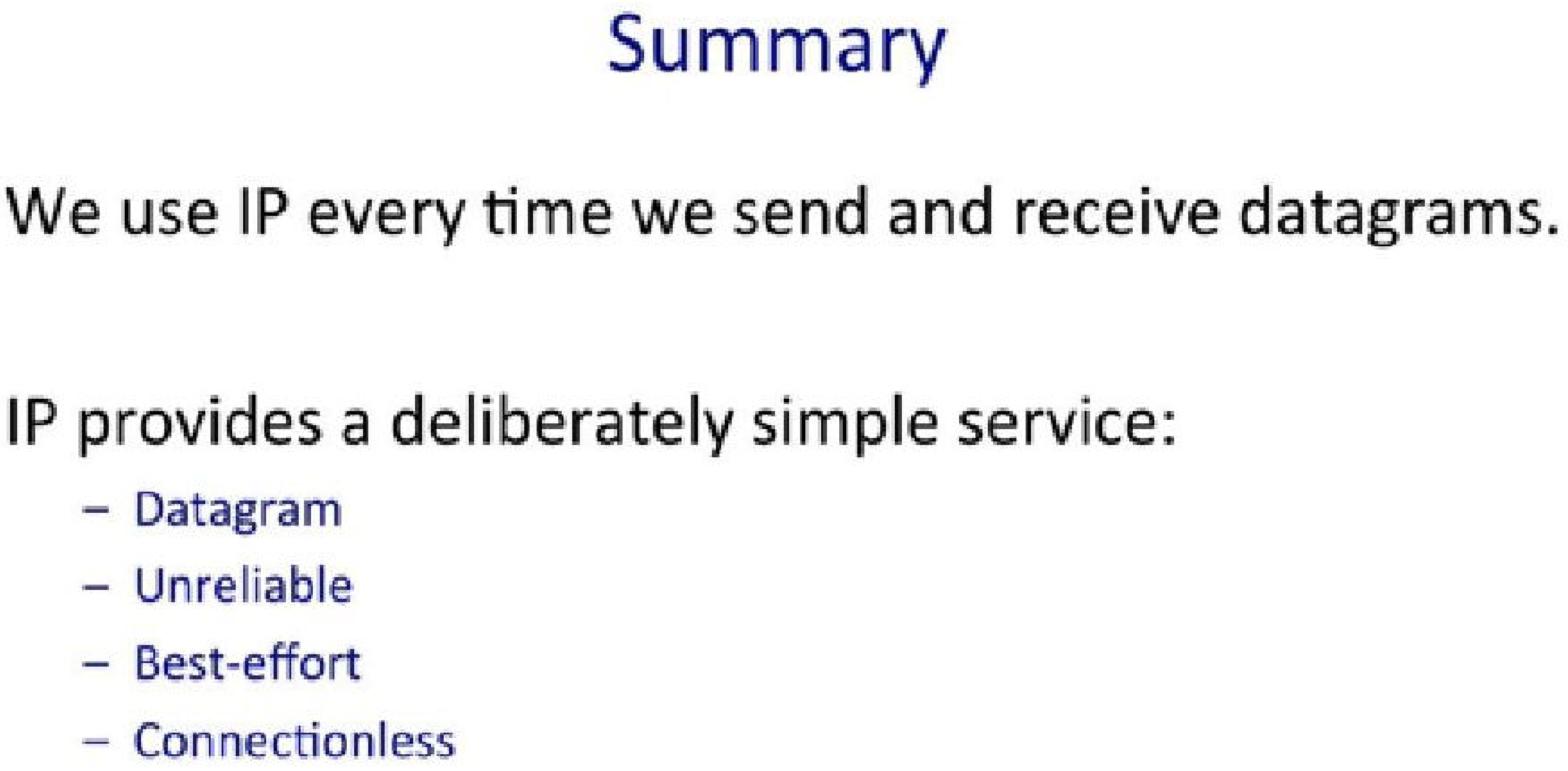
Source IP Address

Destination IP Address

(OPTIONS) | (PAD)

Data

The IP service model



Summary

We use IP every time we send and receive datagrams.

IP provides a deliberately simple service:

- Datagram
- Unreliable
- Best-effort
- Connectionless

End of Lecture 2