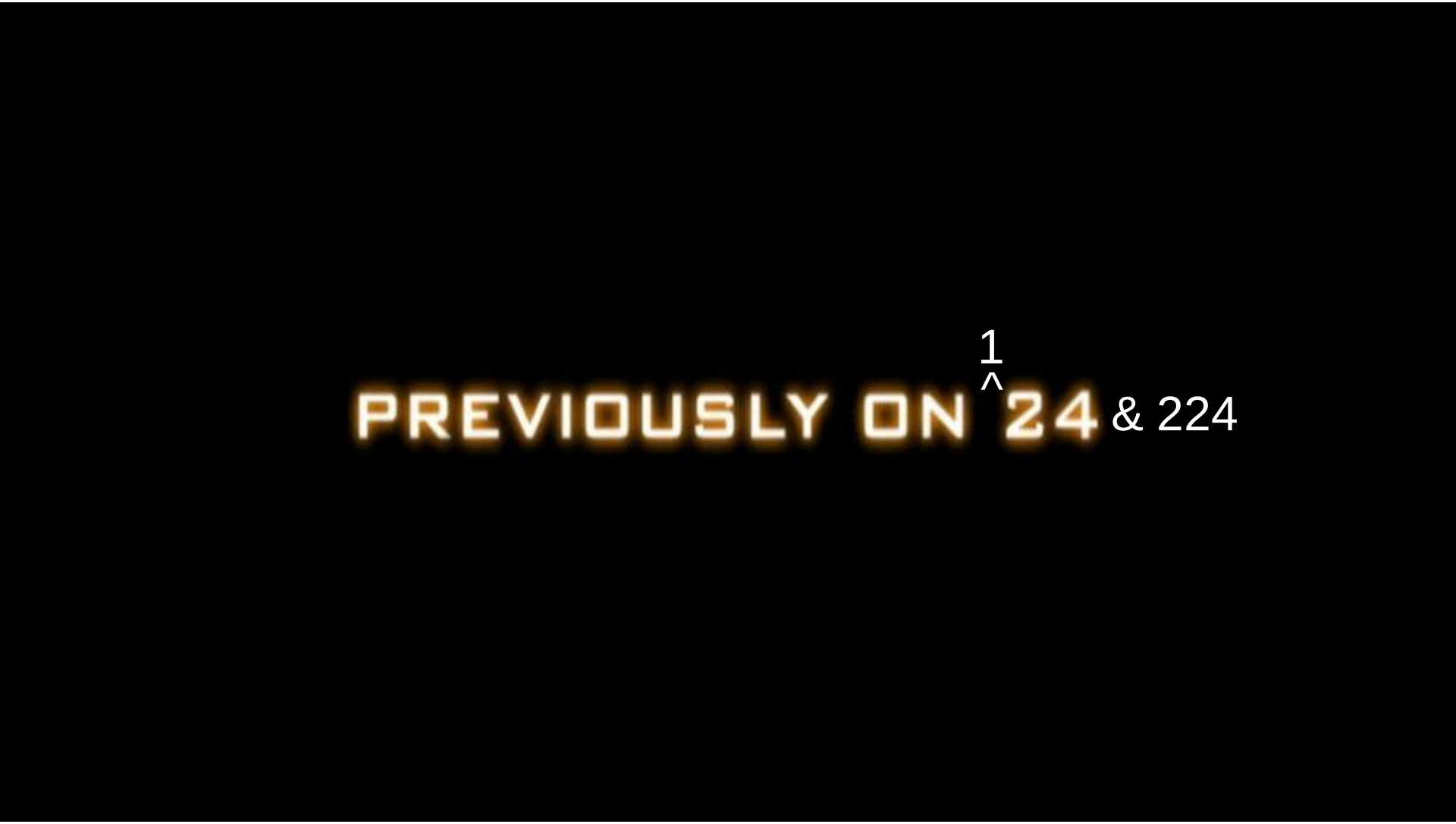
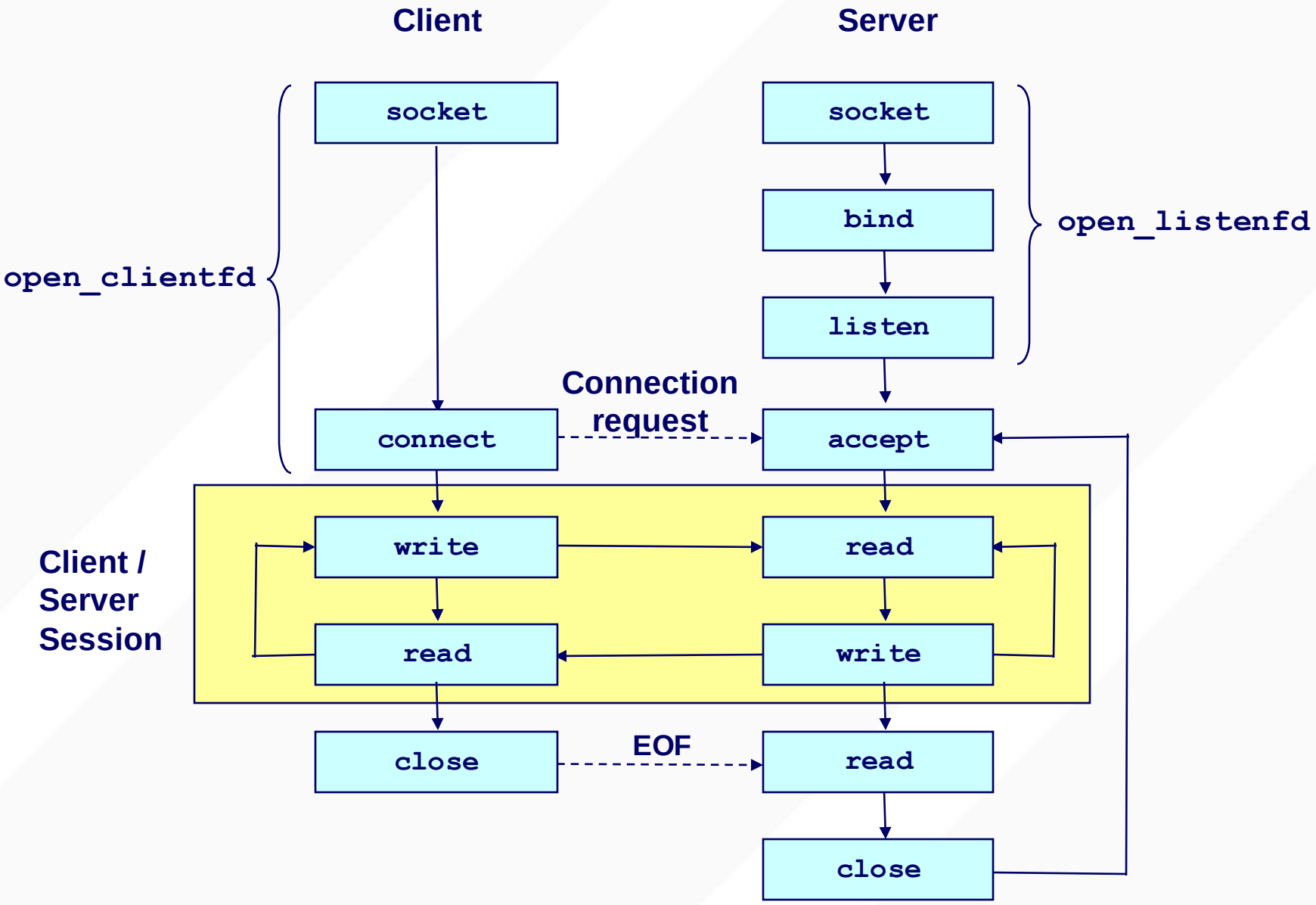


1^ & 224

CLIENT AND SERVER SOCKETS (SYSTEM CALLS)



SENDING DATA (BOTH CLIENT AND SERVER)

```
char *data_addr = "hello, world";
int data_len = 12;

int sent_bytes = send(sock, data_addr, data_len, 0);
if (sent_bytes < 0) {
   perror("send failed");
}
```

- **send()**: sends data, returns the number of sent bytes
  - Also OK with `write()`, `writev()`
- **data_addr**: address of data to send
- **data_len**: size of the data

- With blocking sockets (default), send() blocks until it sends all the data.
- With non-blocking sockets, **sent_bytes** may not equal to **data_len**
  - If kernel does not have enough space, it accepts only partial data
  - You must retry for the unsent data

RECEIVING DATA (BOTH CLIENT AND SERVER)

```
char buffer[4096];
int expected_data_len = sizeof(buffer);

int read_bytes = recv(sock, buffer, expected_data_len, 0);
if (read_bytes == 0) {  // connection is closed
    …
} else if (read_bytes < 0) { // error
   perror("recv failed");
} else {    // OK. But no guarantee read_bytes == expected_data_len
   …
}
```

- **recv()**: reads bytes from the socket and returns the number of read bytes.

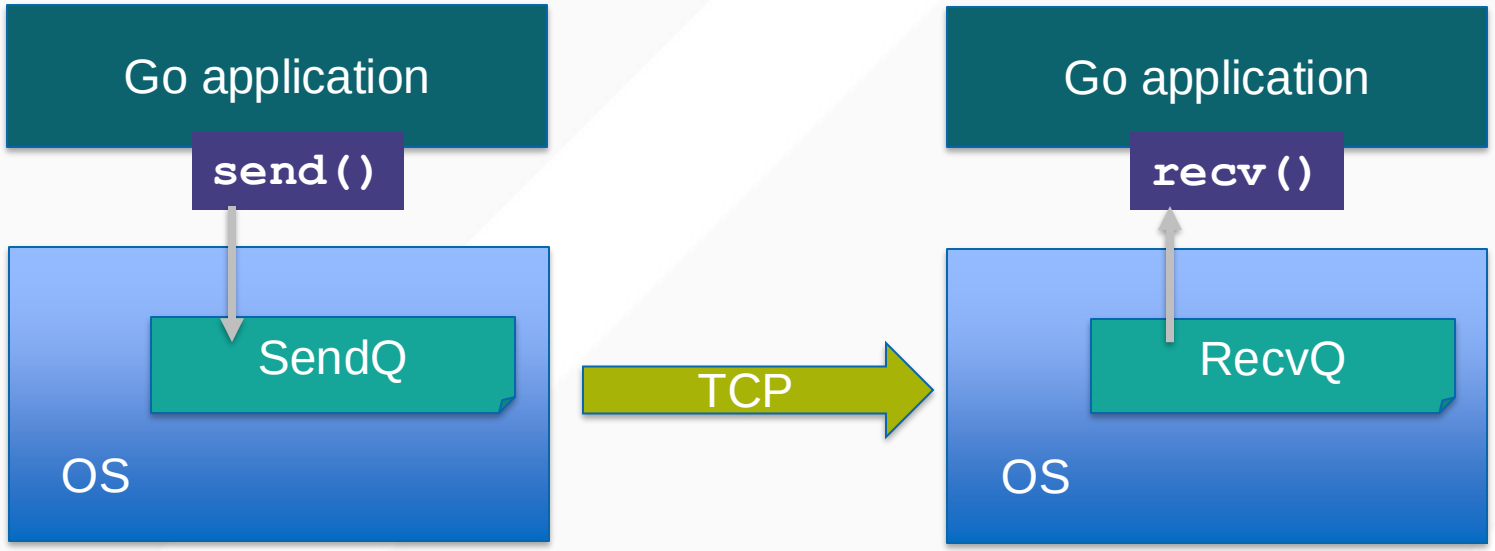
- **read_bytes** may not equal to **expected_data_len**
  - If no data is available, it blocks
  - If only partial data is available, read_bytes < expected_data_len
  - On socket close, expected_data_len equals to 0 (not error!)
  - If you get only partial data, you should retry for the remaining portion.

- ***TIP: You almost always want to call recv()/Read() inside a loop…***

Today's agenda

- Let's write a couple Go programs that send/receive data over the network using the TCP protocol
- Then we'll go back and take a deep dive on why they work and what is going on "under the engine hood"
- TCP and "blocking" in the OS

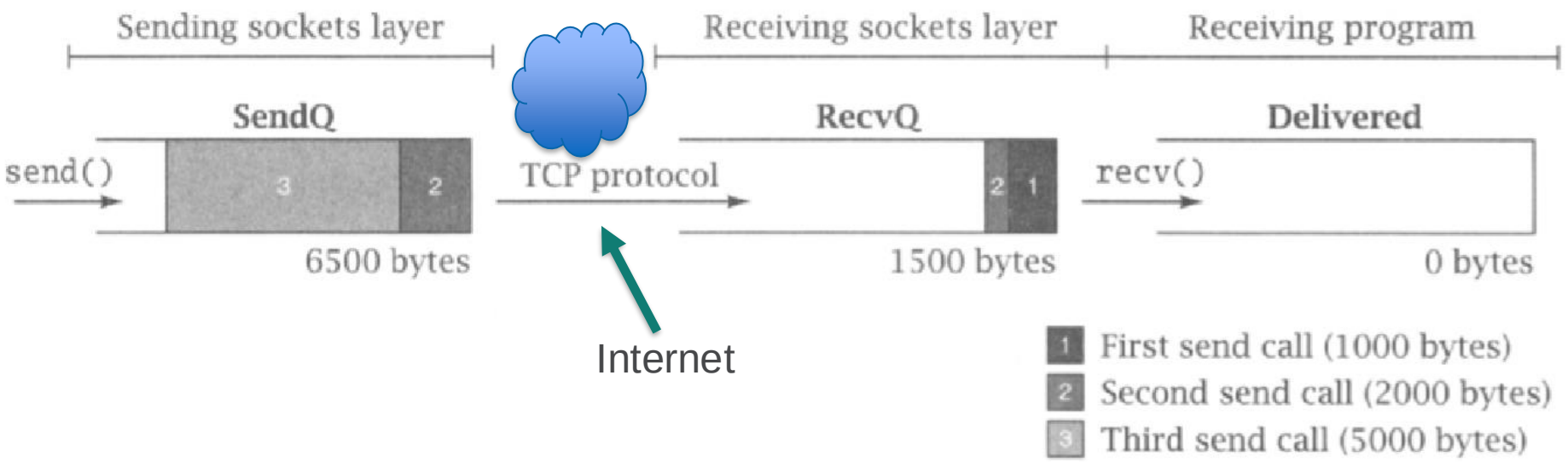
CONCEPTUAL SETUP



DIGGING INTO SEND() A BIT MORE

```
rv = connect(s,...);
.
.
.
rv = send(s,buffer0,1000,0);
.
.
.
rv = send(s,buffer1,2000,0);
.
.
.
rv = send(s,buffer2,5000,0);
.
.
.
close(s);
```

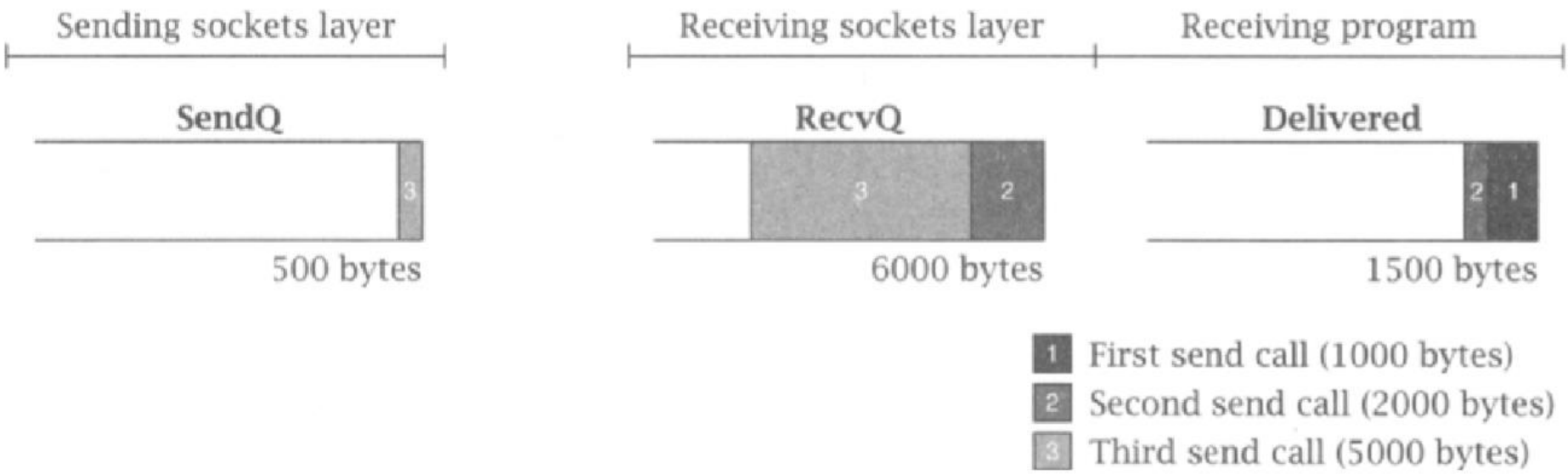
AFTER 3 SEND() CALLS



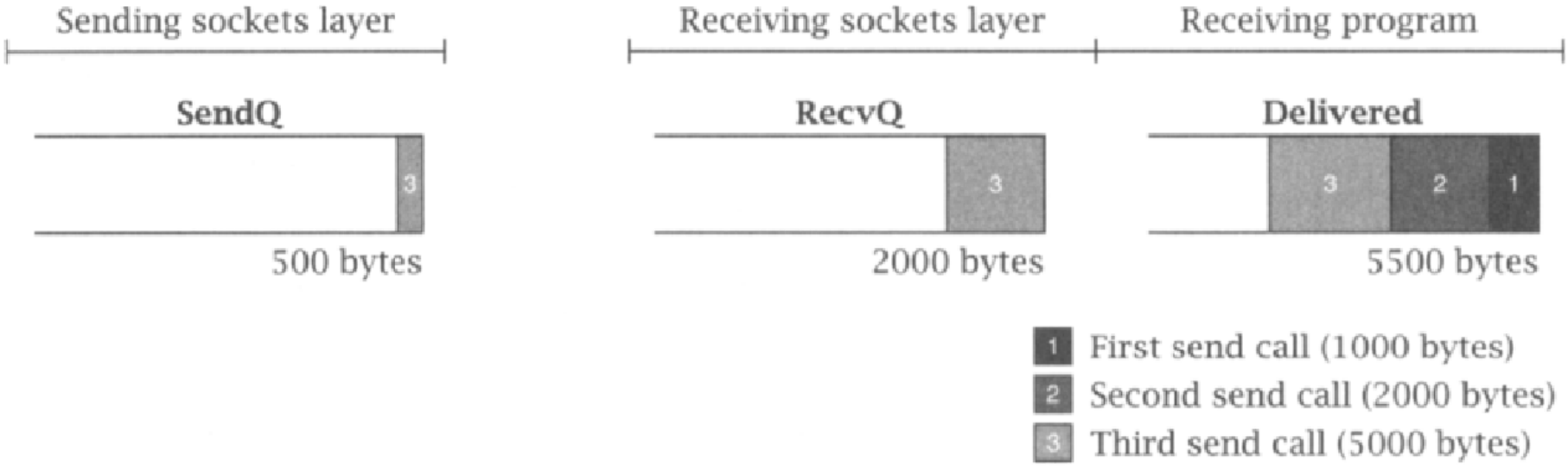
Client machine

Server machine

AFTER FIRST RECV()



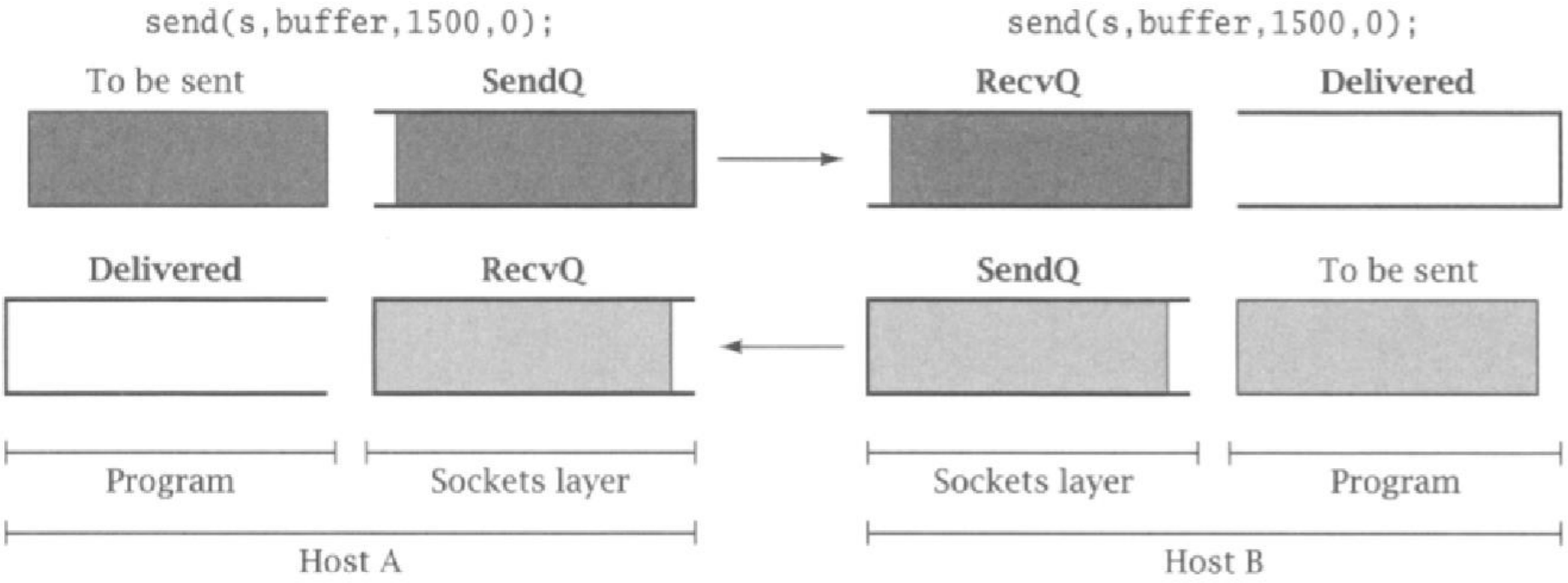
AFTER ANOTHER RECV()



WHEN DOES BLOCKING OCCUR?

- SendQ size: SQS
- RecvQ size: RQS
- send(s, buffer, **n**, 0);
  - n>SQS: blocks until (n – SQS) bytes xfered to RecvQ
  - If n>(SQS+RQS), blocks until receiver calls recv() enough to read in n-(SQS+RQS) bytes
- How does this lead to deadlock?
  - Trivial cause: both sides call recv() w/o sending data

MORE SUBTLE REASON FOR DEADLOCK



- SendQ size = 500; RecvQ size = 500

CSE 124 AND CSE 224:

SENDING AND RECEIVING DATA, ERROR HANDLING, AND PRACTICAL CONSIDERATIONS

George Porter
April 15, 2025

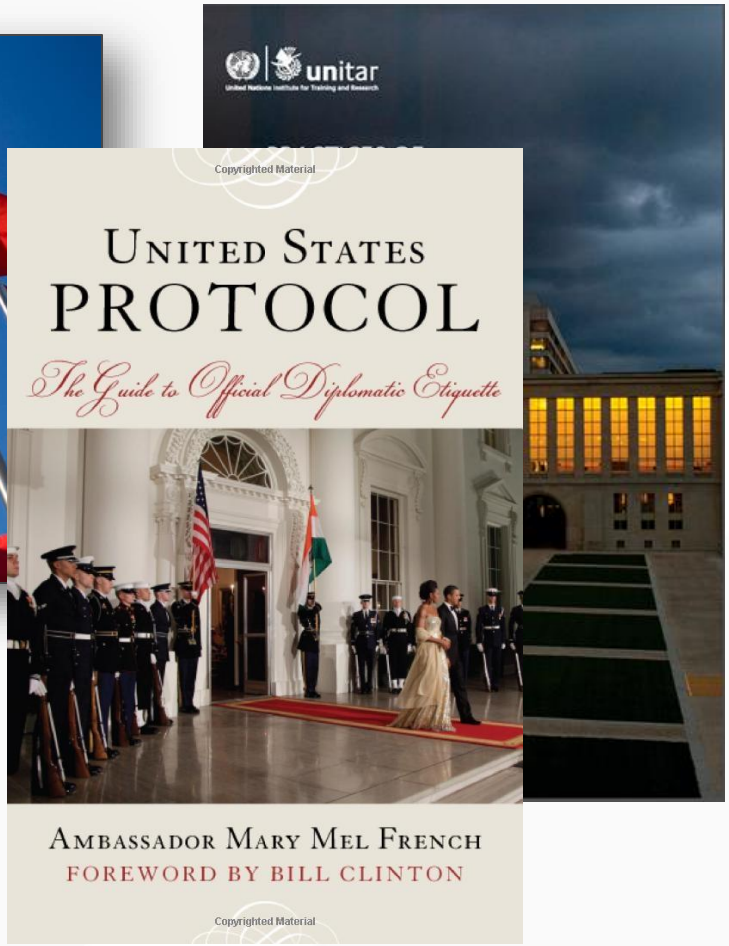
UC San Diego

ATTRIBUTION

- These slides incorporate material from:
  - Computer Networks: A Systems Approach, 5e, by Peterson and Davie
  - Michael Freedman and Kyle Jamieson, Princeton University (also under a CC BY-NC-SA 3.0 Creative Commons license)

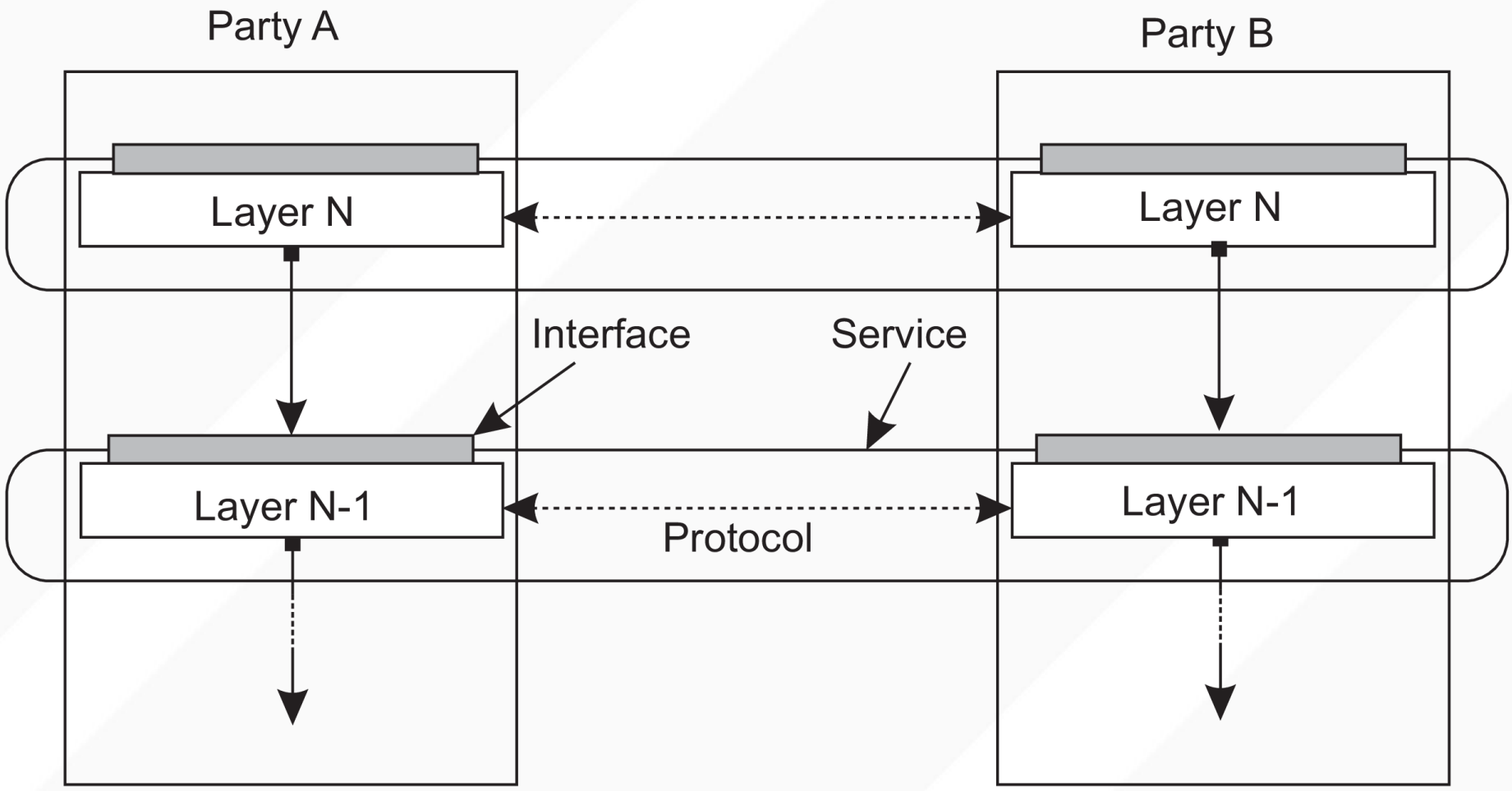
WHAT ARE PROTOCOLS?

Image from public domain



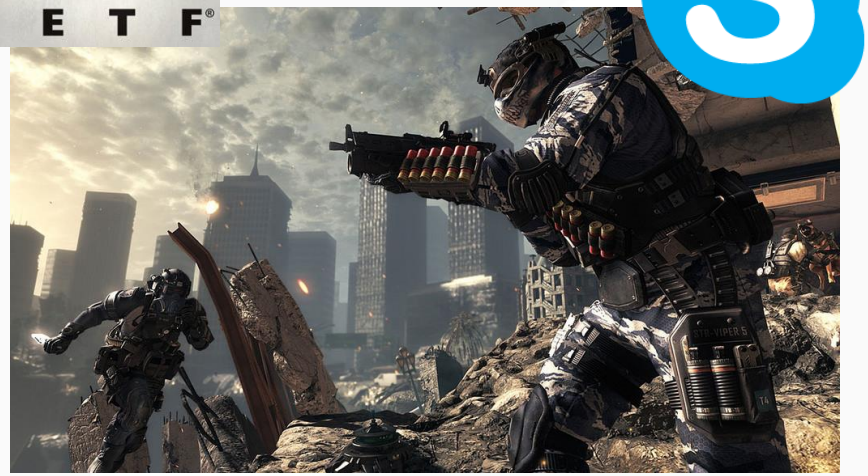
- Explicit and implicit conventions for how to communicate
  - Not for what is communicated
- Enables heterogeneous architectures, languages, OSes, byte ordering, …

SERVICE AND PROTOCOL INTERFACES



WHERE DO PROTOCOLS COME FROM?

- Standards bodies
  - IETF: Internet Engineering Task Force
  - ISO: International Standards Organization
- Community efforts
  - “Request for comments”
  - Bitcoin
- Corporations/industry
  - RealAudio™, Call of Duty multiplayer, Skype

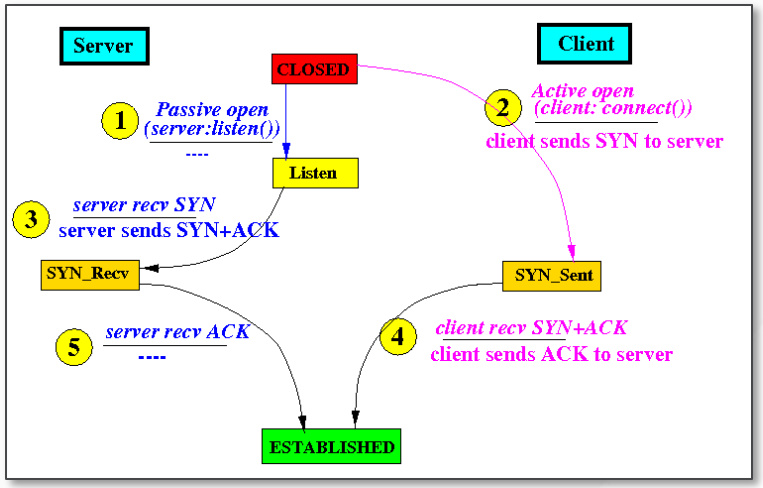


HOW ARE PROTOCOLS SPECIFIED?

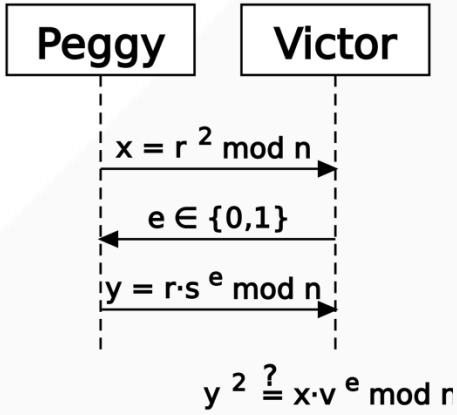
Prose/BNF

```
3.2.  HEADER FIELD DEFINITIONS

     These rules show a field meta-syntax, without regard for the
particular  type  or internal syntax.  Their purpose is to permit
detection of fields; also, they present to  higher-level  parsers
an image of each field as fitting on one line.

field       =  field-name ":" [ field-body ] CRLF

field-name  =  1*<any CHAR, excluding CTLs, SPACE, and ":">

field-body  =  field-body-contents
               [CRLF LWSP-char field-body]

field-body-contents =
              <the ASCII characters making up the field-body, as
               defined in the following sections, and consisting
               of combinations of atom, quoted-string, and
               specials tokens, or else consisting of texts>
```

State transition diagrams

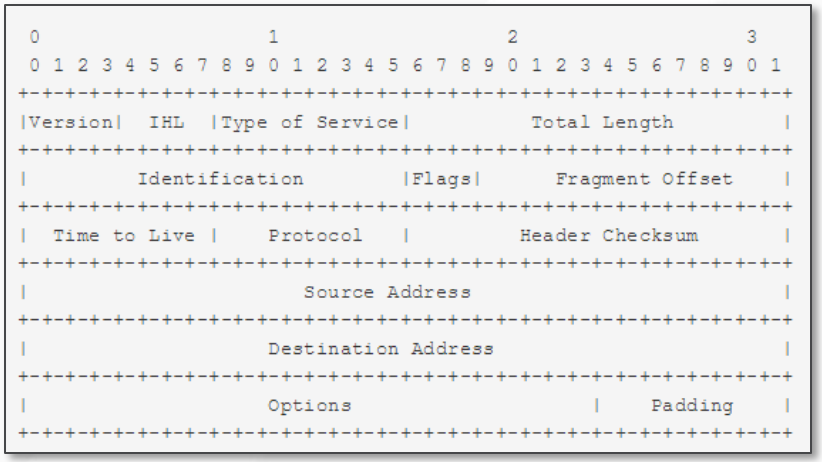


Message Sequence Diagram



By Stefan Birkner, cc-by-sa-2.5,2.0,1.0

Packet formats



EXAMPLE: A SIMPLE VOTE COUNTING SYSTEM

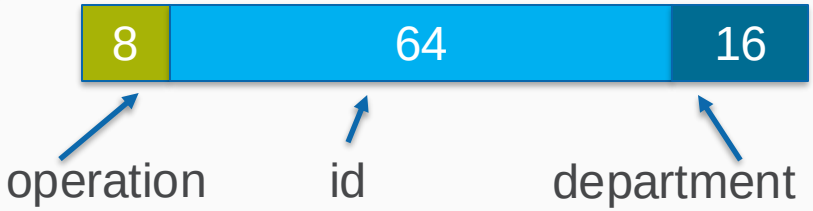
DEFINITIONS

- Operation (e.g., in a voting system)
  - An action you can perform within a protocol's service interface
  - E.g., "Submit vote", "get current vote count", "reset vote count to zero"
- Message
  - An encoding of an operation or data according to a protocol's *wire format.* Common formats include XML, binary, JSON, …
- Framing
  - Writing out (and reading in) messages from a stream such that messages can be separated and interpreted correctly
- Parsing/encoding/decoding
  - Converting a message to/from an application-level data structure

PARSING: CONVERTING IN-MEMORY REPRESENTATION WITH A "WIRE" REPRESENTATION

```
type Employee struct {
        operation uint8
        id uint64
        department uint16
}
```

- Binary



- Text (ad-hoc)

"OP=1, id=428, d=80"

(1,428,80)

- Text (XML)

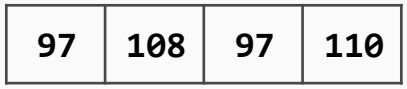
```
<employee>
  <operation>1</operation>
  <id>428</id>
  <department>80</department>
</employee>
```

- Many others…

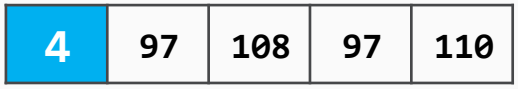
FRAMING: LENGTH SPECIFICATION VS DELIMITERS

```
package main

import "fmt"

type Employee struct {
        operation uint8
        name string
        id uint64
        department uint16
}
```

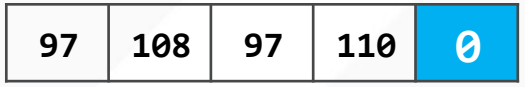
- Binary representation of name?
  - Handling variable length
- Consider “Alan” as a name



- Option 1: Explicit length

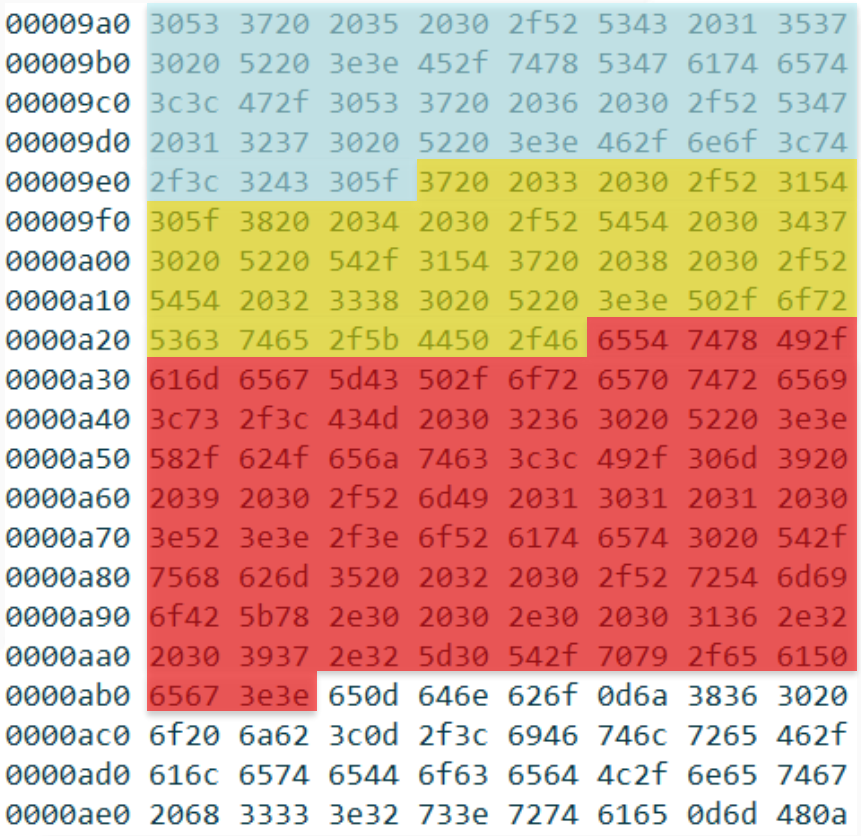


  - But how big should length be?
- Option 2: Delimiter



  - But what if delimiter is in the message?

FRAMING: DETERMINING MESSAGE BOUNDARIES



```
00009a0 3053 3720 2035 2030 2f52 5343 2031 3537
00009b0 3020 5220 3e3e 452f 7478 5347 6174 6574
00009c0 3c3c 472f 3053 3720 2036 2030 2f52 5347
00009d0 2031 3237 3020 5220 3e3e 462f 6e6f 3c74
00009e0 2f3c 3243 305f 3720 2033 2030 2f52 3154
00009f0 305f 3820 2034 2030 2f52 5454 2030 3437
0000a00 3020 5220 542f 3154 3720 2038 2030 2f52
0000a10 5454 2032 3338 3020 5220 3e3e 502f 6f72
0000a20 5363 7465 2f5b 4450 2f46 6554 7478 492f
0000a30 616d 6567 5d43 502f 6f72 6570 7472 6569
0000a40 3c73 2f3c 434d 2030 3236 3020 5220 3e3e
0000a50 582f 624f 656a 7463 3c3c 492f 306d 3920
0000a60 2039 2030 2f52 6d49 2031 3031 2031 2030
0000a70 3e52 3e3e 2f3e 6f52 6174 6574 3020 542f
0000a80 7568 626d 3520 2032 2030 2f52 7254 6d69
0000a90 6f42 5b78 2e30 2030 2e30 2030 3136 2e32
0000aa0 2030 3937 2e32 5d30 542f 7079 2f65 6150
0000ab0 6567 3e3e 650d 646e 626f 0d6a 3836 3020
0000ac0 6f20 6a62 3c0d 2f3c 6946 746c 7265 462f
0000ad0 616c 6574 6544 6f63 6564 4c2f 6e65 7467
0000ae0 2068 3333 3e32 733e 7274 6165 0d6d 480a
```

- Framing
  - Finds and returns bytes corresponding to single message
  - Even if messages are variable length
  - Writes out bytes corresponding to a message with enough *context* for the other side to determine the message boundaries

FRAMING SCENARIO

- Consider a voting scenario
- Each message is variable length
  - “Voting v 134” → [Vote for candidate 134]
  - “Voting i 19381”
    - → [Query candidate 19381’s vote count]
  - First is 12 characters, second is 14 characters
- Given a stream of vote operations, how to separate them?

FRAMING CHOICES

- Delimiter (in this case ‘$’)

```
Voting v 134$Voting v 2817$Voting i
9172651$Voting v 2$Voting i 1900$Voting v
32$Voting i 8
```

- Length + message

```
12Voting v 13413Voting v 281716Voting i
917265110Voting v 213Voting i 190011Voting v
3210Voting i 8
```

THE MAIN LOOP OF YOUR SERVER

```
Remaining := “”
buf := make([]byte, 1024)
for {
   for “Does remaining contain a full request?” {
            If yes,(1) parse it, then (2) remove from remaining
    }
  size, err := c.Read(buf)
  data := buf[:size]
  remaining = remaining + string(data)
}
```

How do you know when a request is completed?

HOW TO TELL IF BUFFER CONTAINS A COMPLETE REQUEST?

- This is the framing problem
- For length-based framing:

  - Keep reading until we have 12 bytes of request data
- For delimiter-based framing:
  - OK to simply scan for delimiters using e.g., a for loop

FRAMING: SUMMARY

- Writing data
  - Given an array of bytes representing an application-level operation, writes to stream
  1. Explicit length
     - Writes out the length of the message, then message
  2. Delimiter
     - Ensures delimiter doesn't appear in message
     - Writes out message
     - Then writes out delimiter
- Reading data
  - Reads from stream until entire message is read, returns to higher layer
  1. Explicit length
     - Reads the length, then reads that many bytes (security?)
  2. Delimiter
     - Reads continuously into a buffer until delimiter is encountered
     - Message then returned to higher layer

PRIMARY FRAMING/PARSING LOOP

- [see turing-printer.go and turing-sender.go demo]
- Concepts introduced in this code:
  - flags package
  - defer command
  - pointers
  - ‘go’ command (for concurrency)
  - Anonymous functions
  - bufio package and Scanner


UC San Diego