Transport Layer
Part 4

Mark Allman
mallman@case.edu

EECS 325/425
Fall 2018

"Got my 45 on …
…so I …
… can rock on"

These slides are more-or-less directly from the slide set developed by Jim Kurose and Keith Ross for their book "Computer Networking: A Top Down Approach, 5th edition".

The slides have been lightly adapted for Mark Allman's EECS 325/425 Computer Networks class at Case Western Reserve University.

Traffic By Protocol

CCZ, Sept. 10 2018

Traffic By Protocol

CCZ, Sept. 10 2018

Traffic By Protocol

CCZ, Sept. 10 2018

Traffic By Protocol

	Bytes
TCP	856.2 GB ~89%
UDP	102.8 GB ~11%

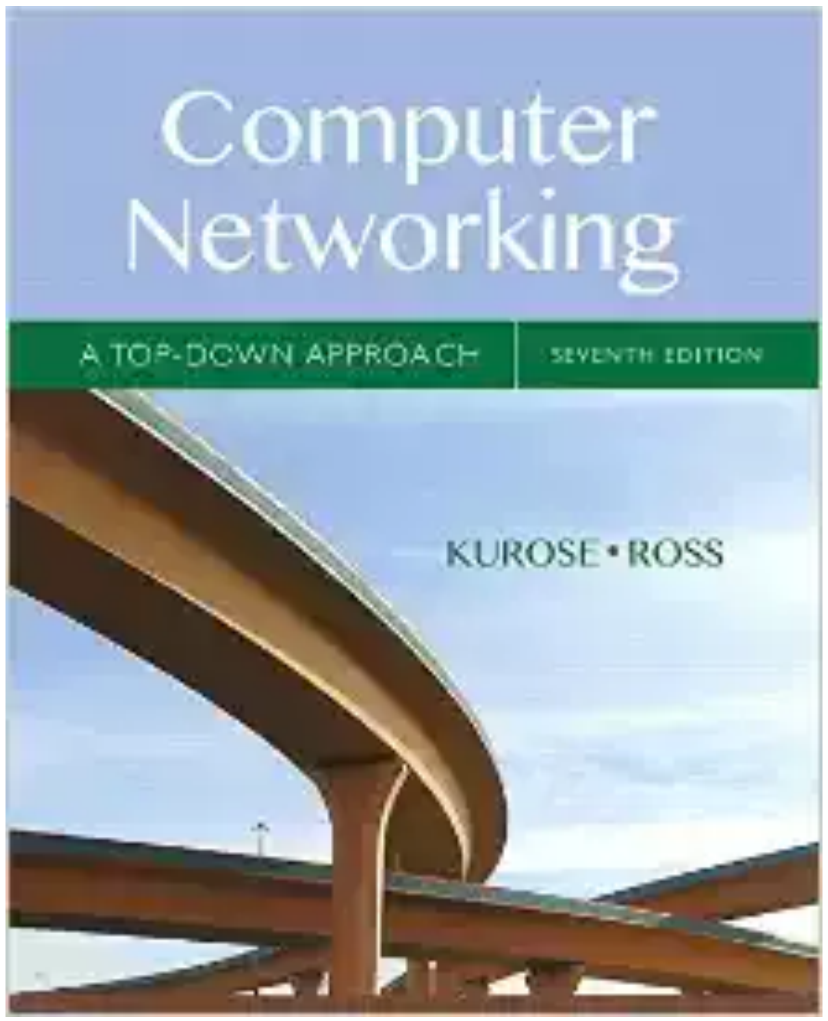
CCZ, Sept. 10 2018

Traffic By Protocol

	Bytes	Connections
TCP	856.2 GB ~89%	1.7 M ~53%
UDP	102.8 GB ~11%	1.5 M ~47%

CCZ, Sept. 10 2018

Reading Along ...



- 3.5: Connection-oriented transport: TCP

TCP: Overview

TCP: Overview

- point-to-point:
  - one sender, one receiver

TCP: Overview

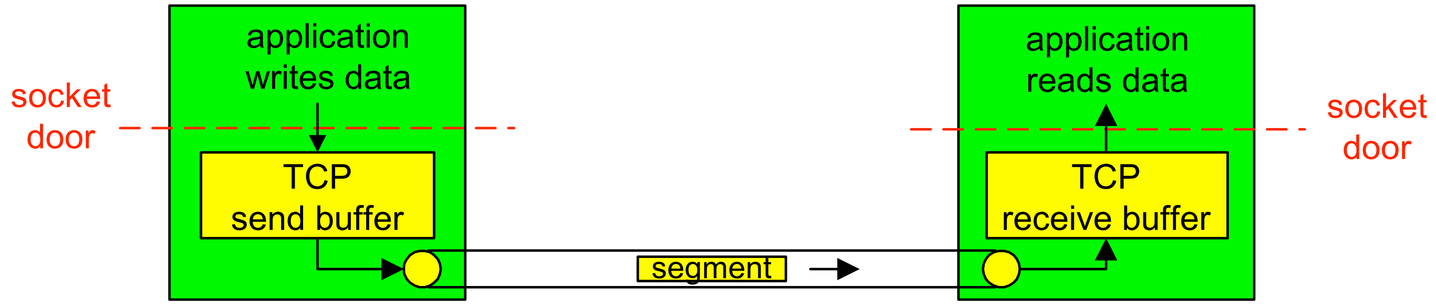
- point-to-point:
  - one sender, one receiver
- reliable, in-order byte stream:
  - no “message boundaries”

TCP: Overview

- point-to-point:
  - one sender, one receiver
- reliable, in-order byte stream:
  - no "message boundaries"
- pipelined:
  - TCP congestion and flow control set window size

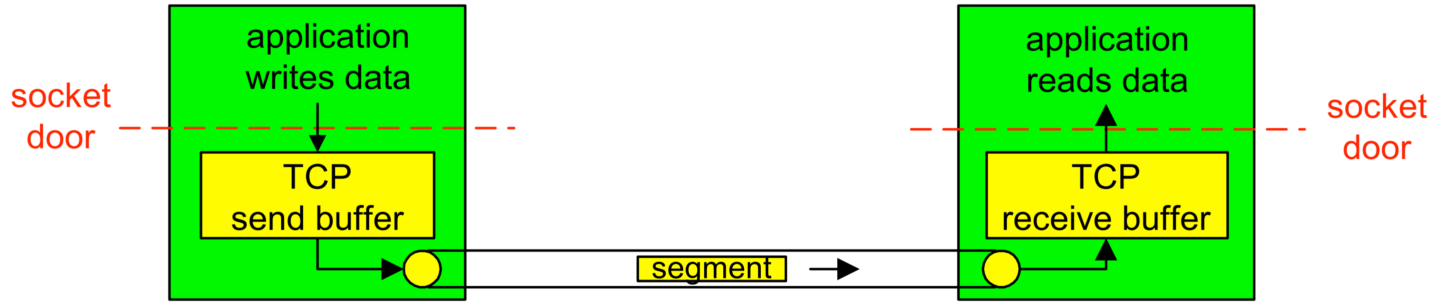
TCP: Overview

- point-to-point:
  - one sender, one receiver
- reliable, in-order byte stream:
  - no "message boundaries"
- pipelined:
  - TCP congestion and flow control set window size
- send & receive buffers



TCP: Overview

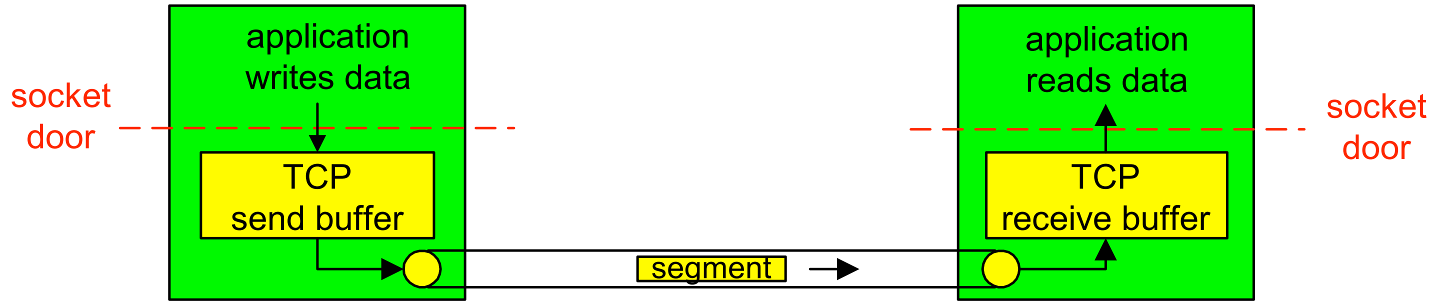
- point-to-point:
  - one sender, one receiver
- reliable, in-order byte stream:
  - no "message boundaries"
- pipelined:
  - TCP congestion and flow control set window size
- send & receive buffers



- full duplex data:
  - bi-directional data flow in same connection

TCP: Overview

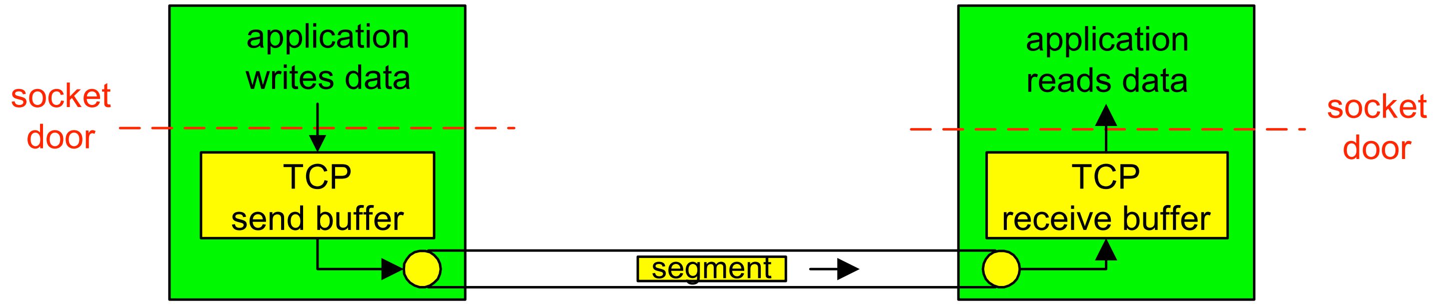
- **point-to-point:**
  - one sender, one receiver
- **reliable, in-order byte stream:**
  - no "message boundaries"
- **pipelined:**
  - TCP congestion and flow control set window size
- **send & receive buffers**



- **full duplex data:**
  - bi-directional data flow in same connection
- **connection-oriented:**
  - handshaking inits sender, receiver state before data exchange

TCP: Overview

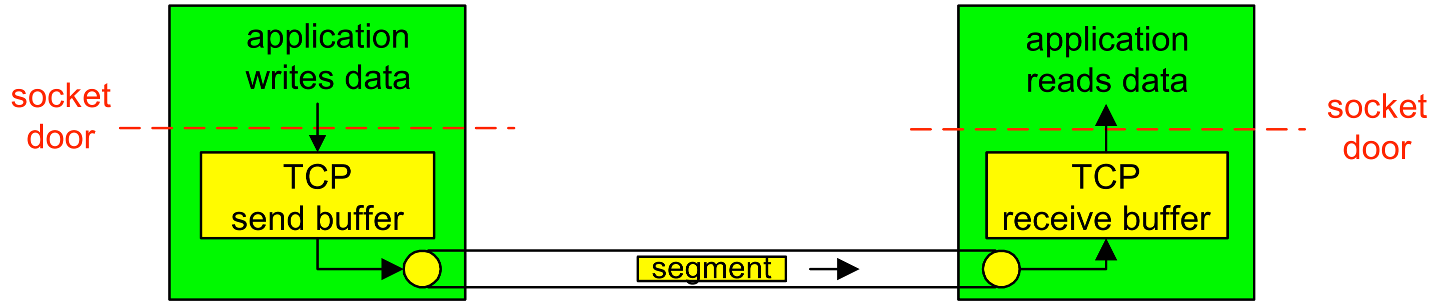
- point-to-point:
  - one sender, one receiver
- reliable, in-order byte stream:
  - no "message boundaries"
- pipelined:
  - TCP congestion and flow control set window size
- send & receive buffers



- full duplex data:
  - bi-directional data flow in same connection
- connection-oriented:
  - handshaking inits sender, receiver state before data exchange
- congestion controlled:
  - won't overwhelm network
  - (tackle later in semester)

TCP: Overview

- point-to-point:
  - one sender, one receiver
- reliable, in-order byte stream:
  - no "message boundaries"
- pipelined:
  - TCP congestion and flow control set window size
- send & receive buffers

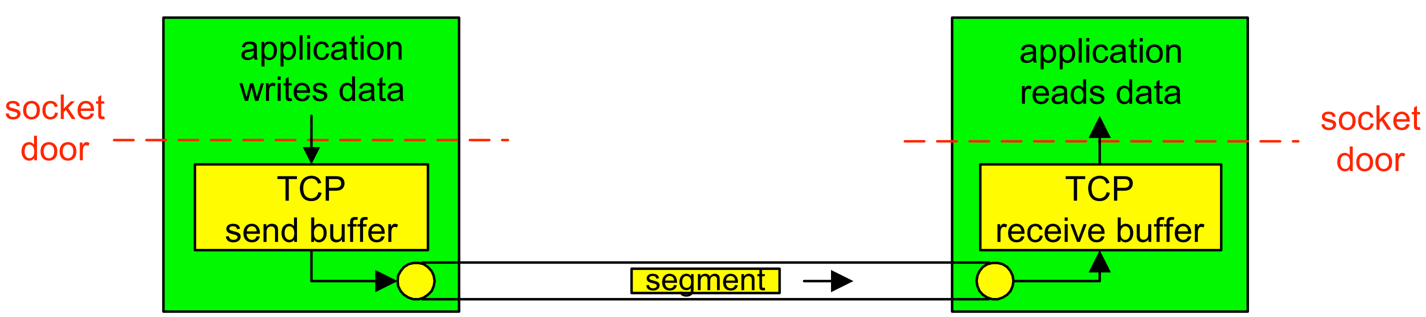


- full duplex data:
  - bi-directional data flow in same connection
- connection-oriented:
  - handshaking inits sender, receiver state before data exchange
- congestion controlled:
  - won't overwhelm network
  - (tackle later in semester)
- flow controlled:
  - won't overwhelm receiver

TCP: Overview

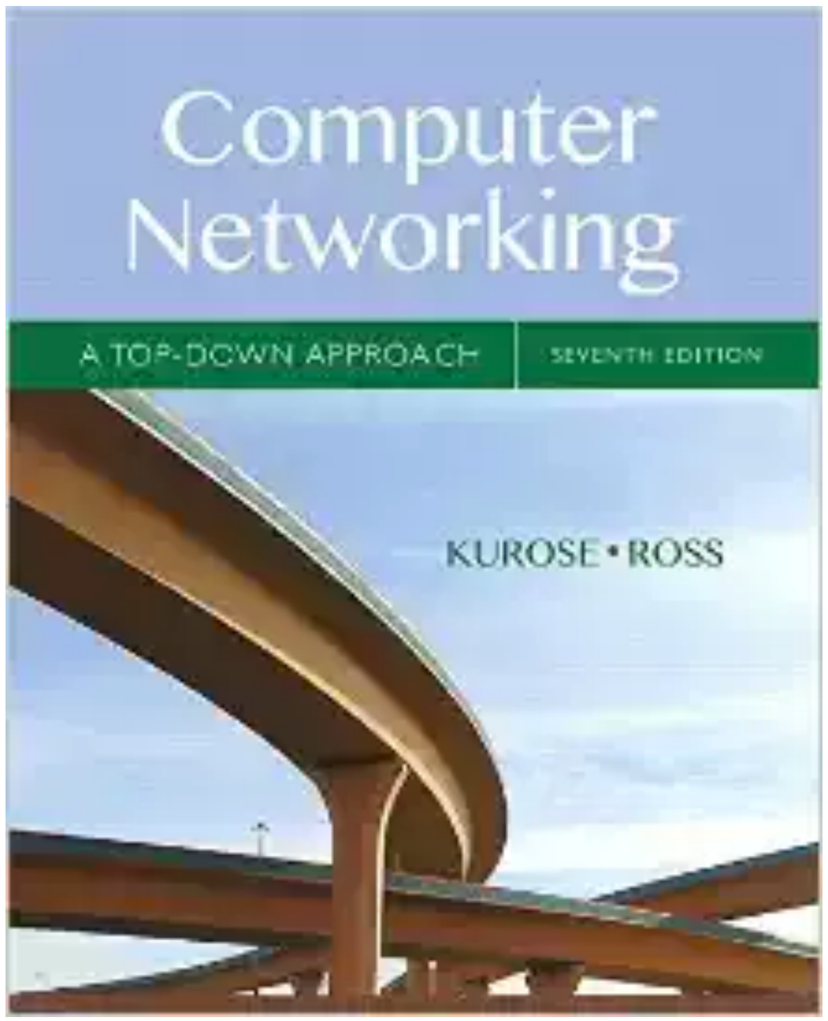
RFCs: 793, 1122, 1323, 2018, 2883, 3042, 3390, 5681, 6298, 6675, etc

- **point-to-point:**
  - one sender, one receiver
- **reliable, in-order byte stream:**
  - no "message boundaries"
- **pipelined:**
  - TCP congestion and flow control set window size
- **send & receive buffers**



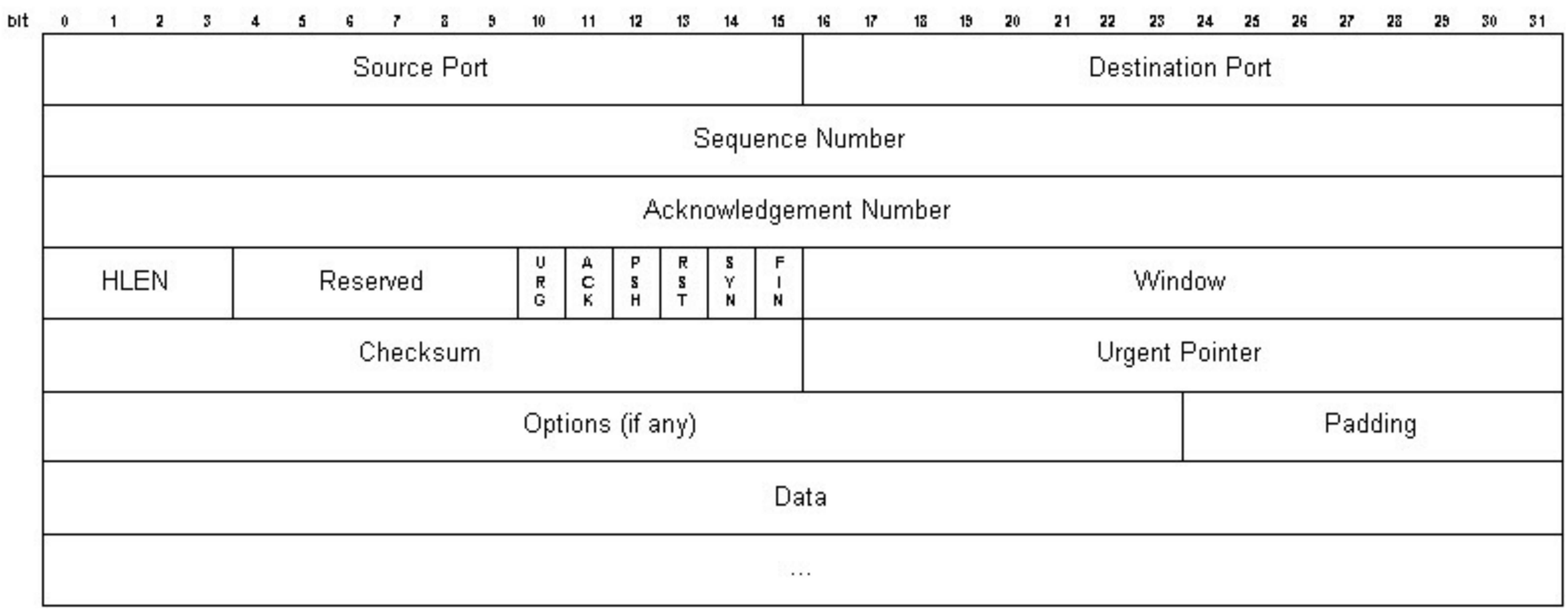
- **full duplex data:**
  - bi-directional data flow in same connection
- **connection-oriented:**
  - handshaking inits sender, receiver state before data exchange
- **congestion controlled:**
  - won't overwhelm network
  - (tackle later in semester)
- **flow controlled:**
  - won't overwhelm receiver

Reading Along ...

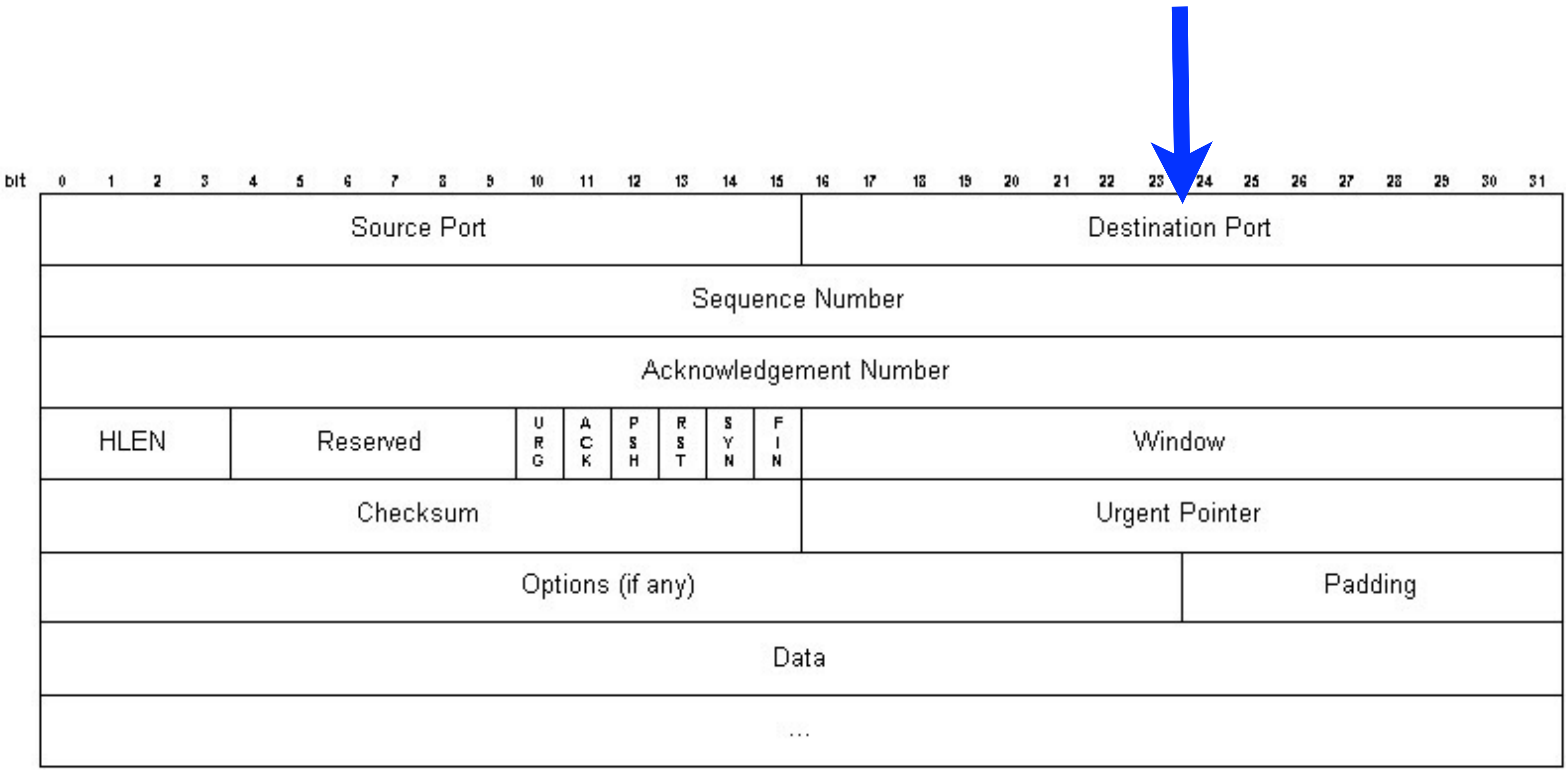


- 3.5: Connection-oriented transport: TCP
  - segment structure

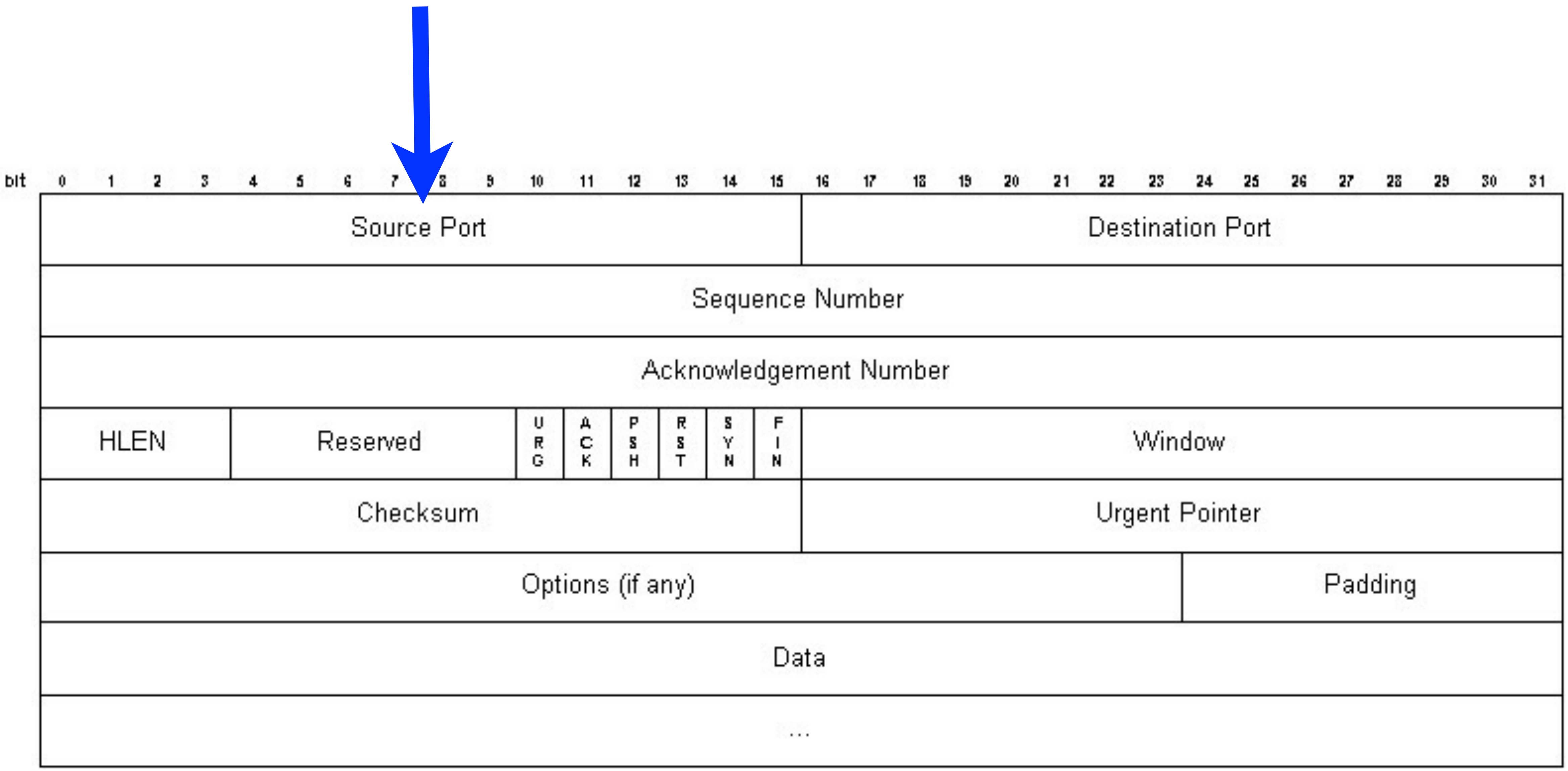
TCP Segment Structure



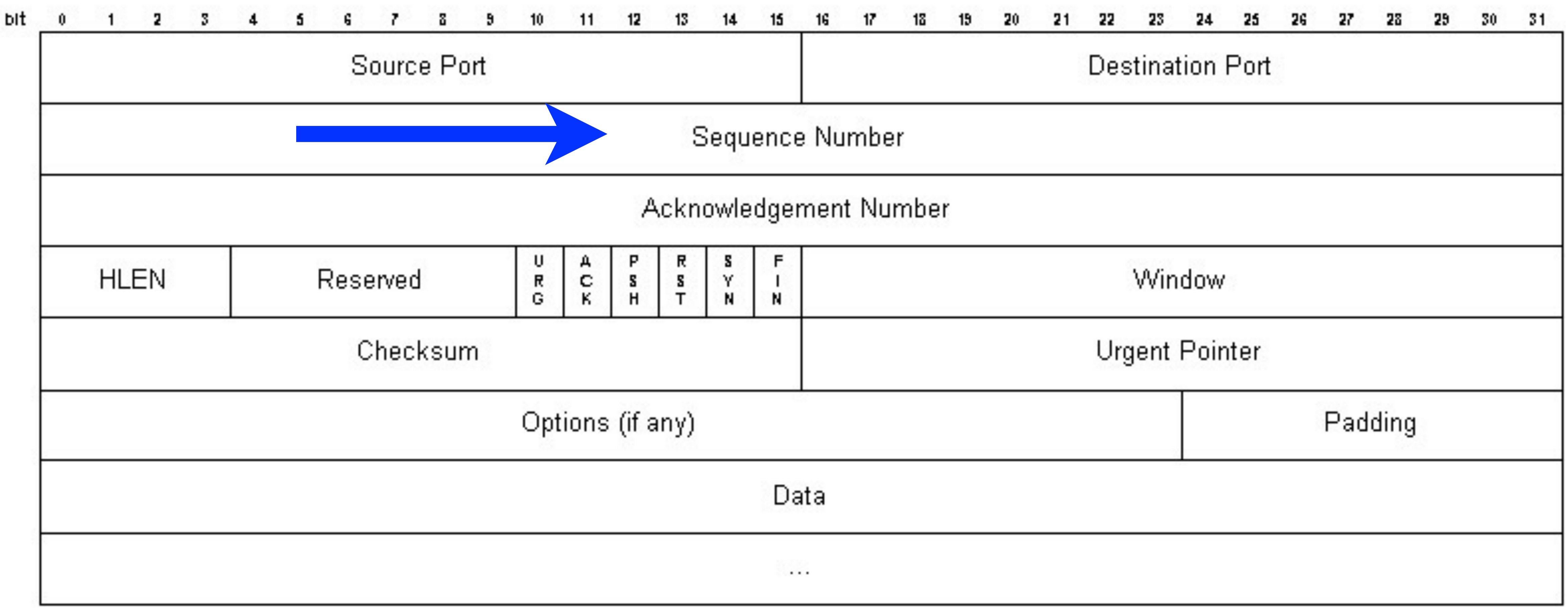
TCP Segment Structure



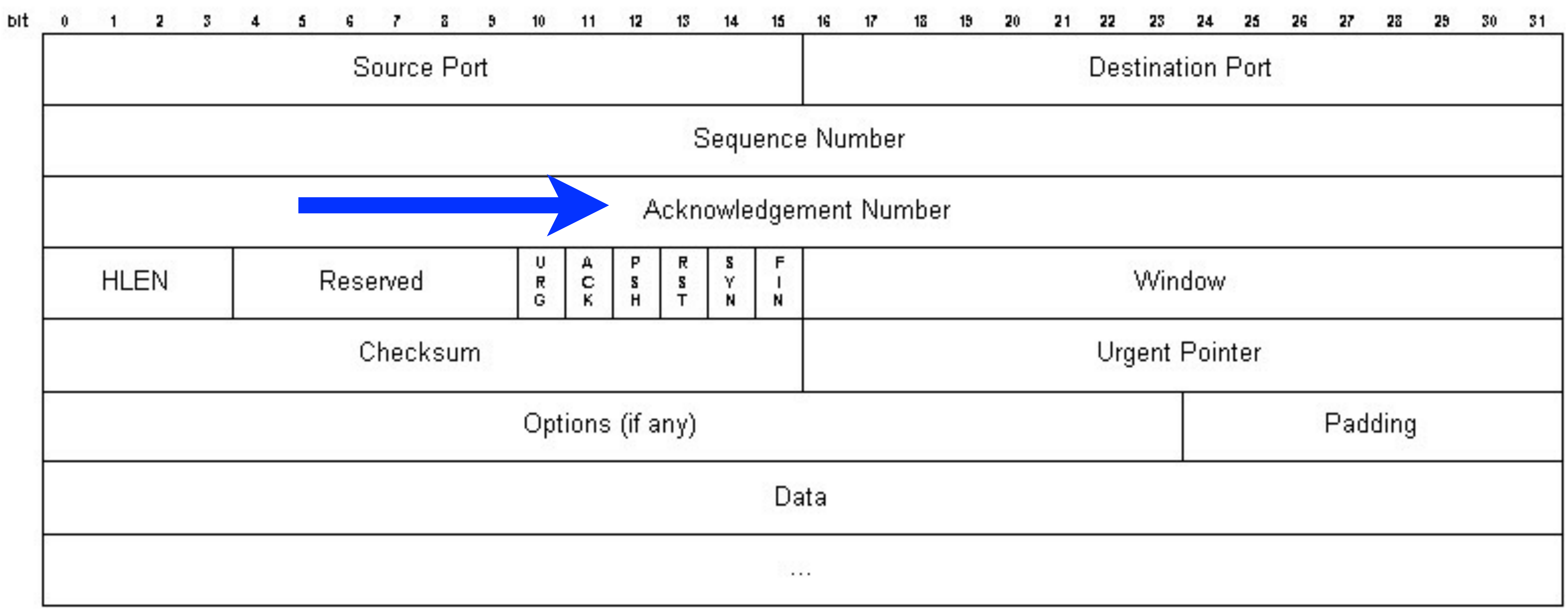
TCP Segment Structure



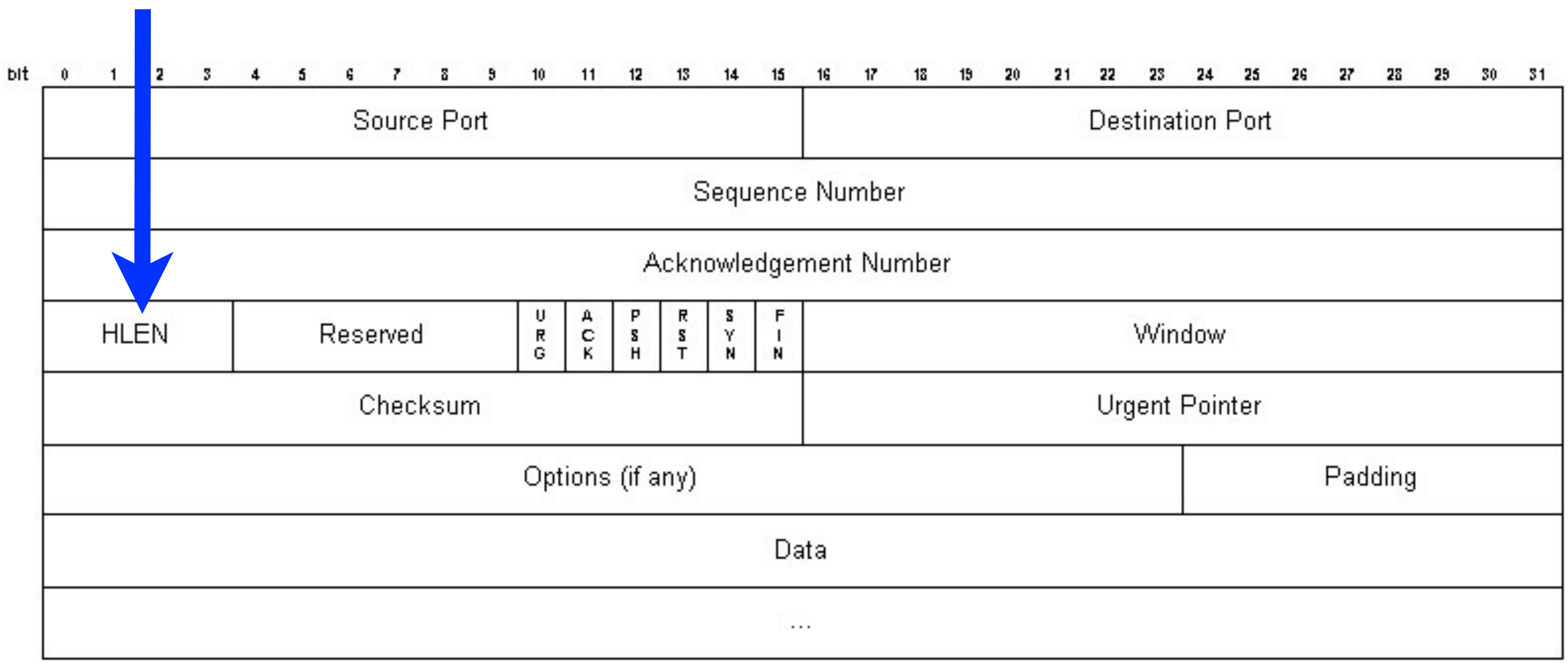
TCP Segment Structure



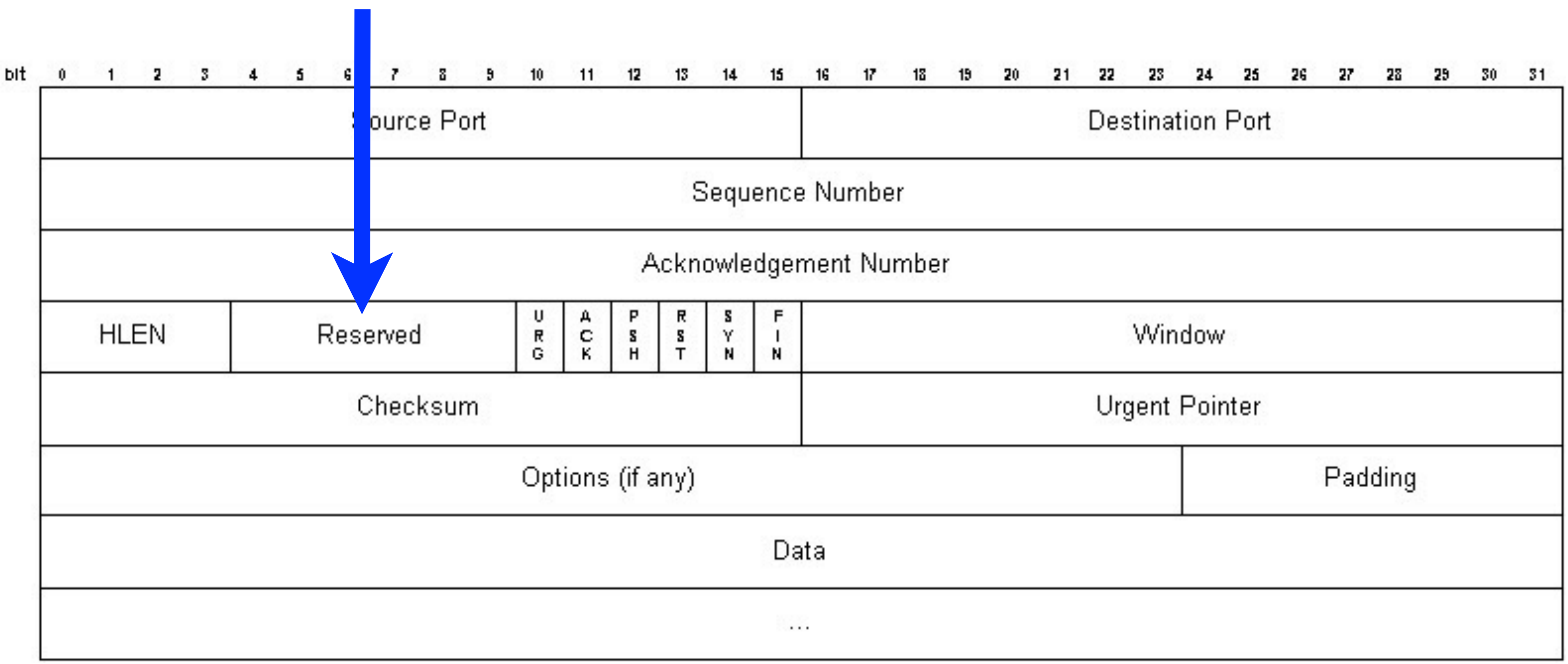
TCP Segment Structure



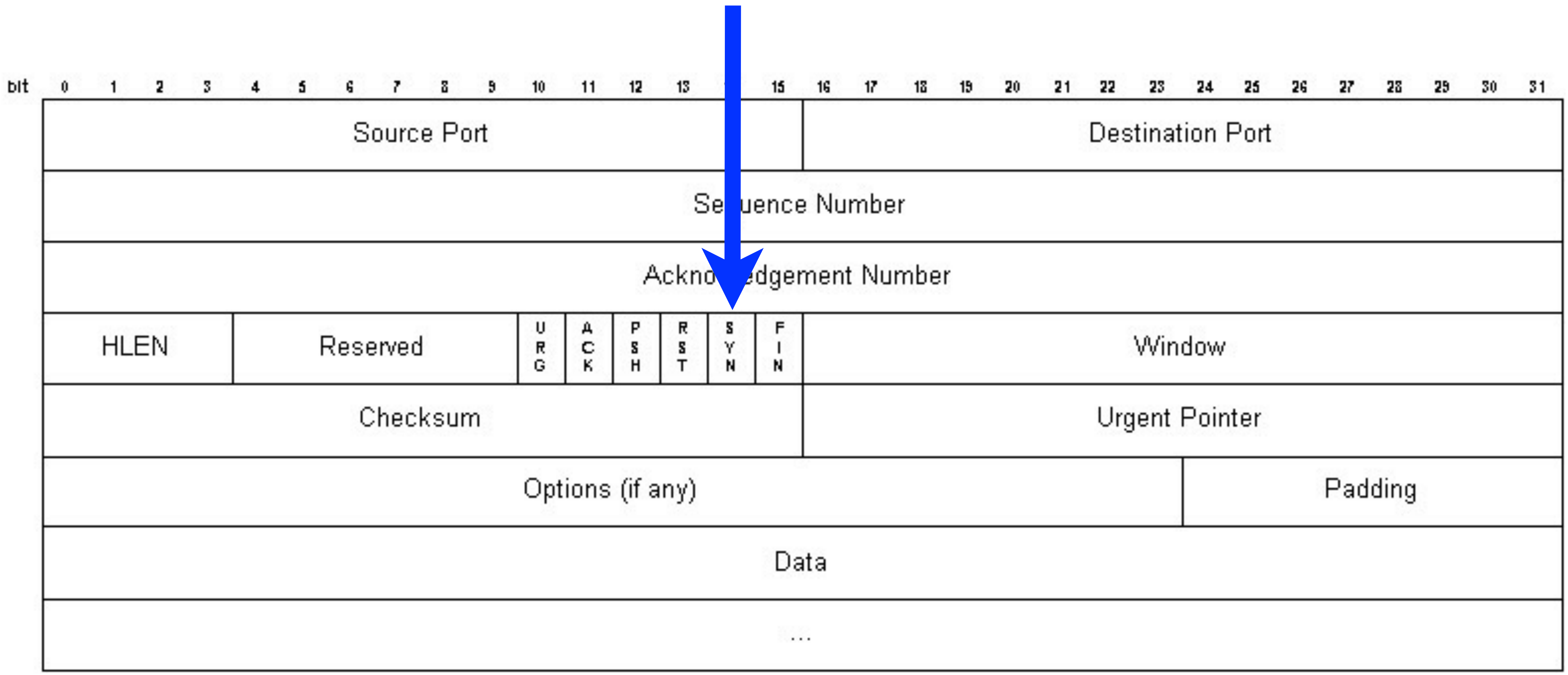
TCP Segment Structure



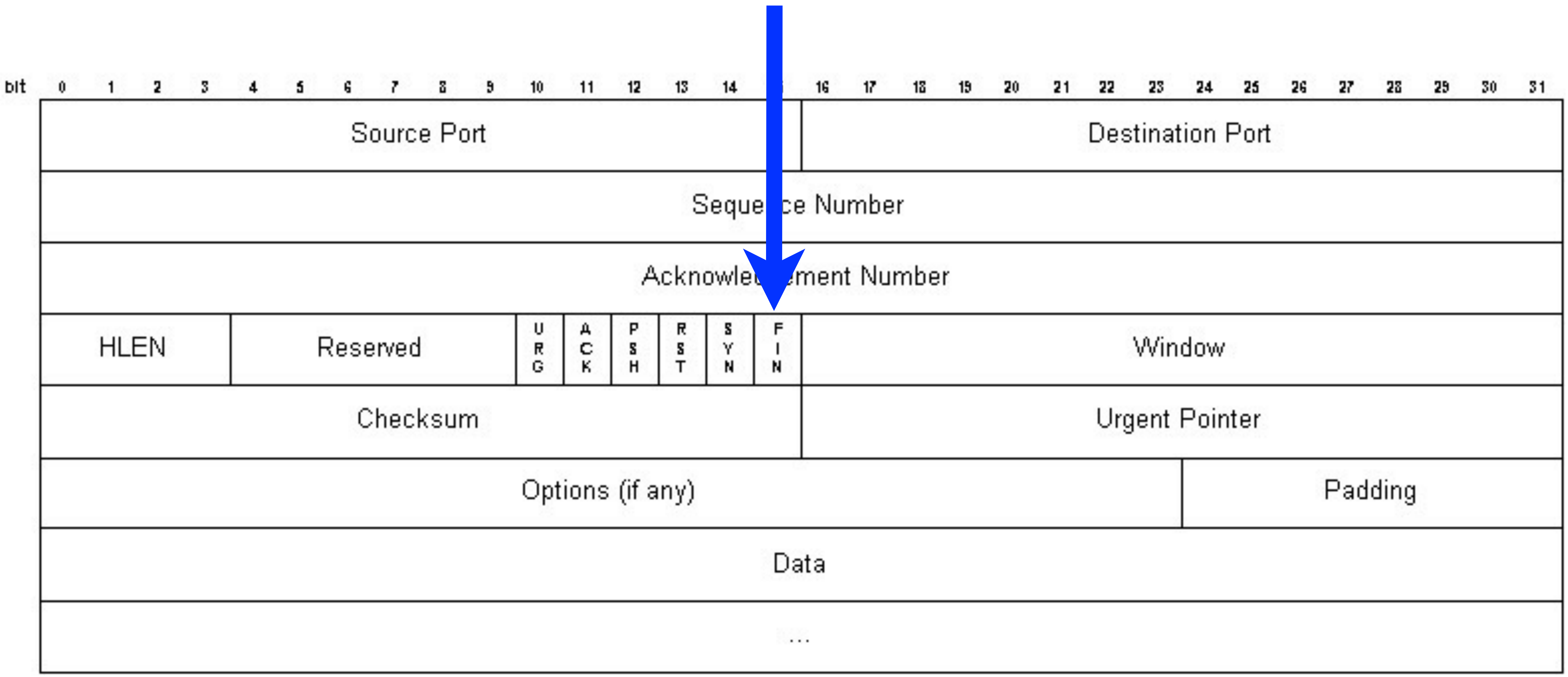
TCP Segment Structure



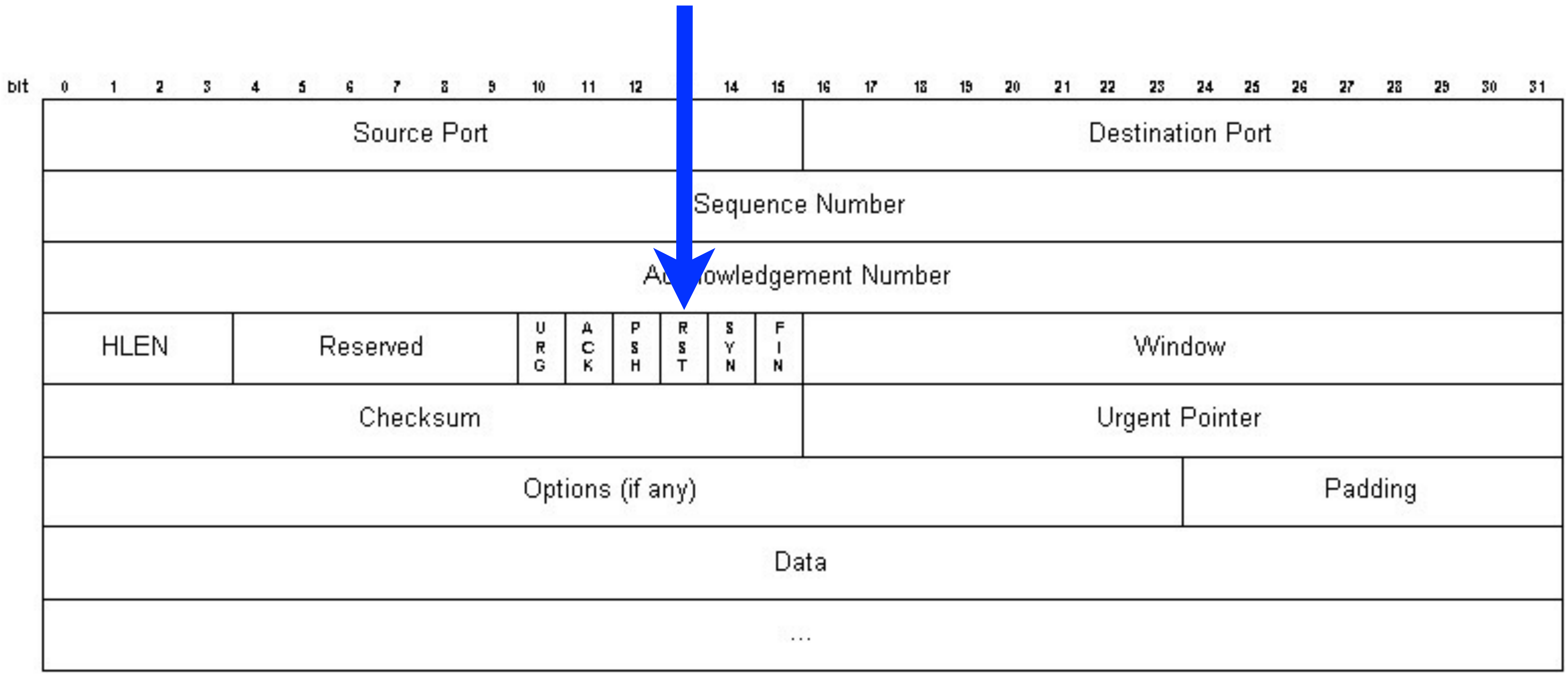
TCP Segment Structure



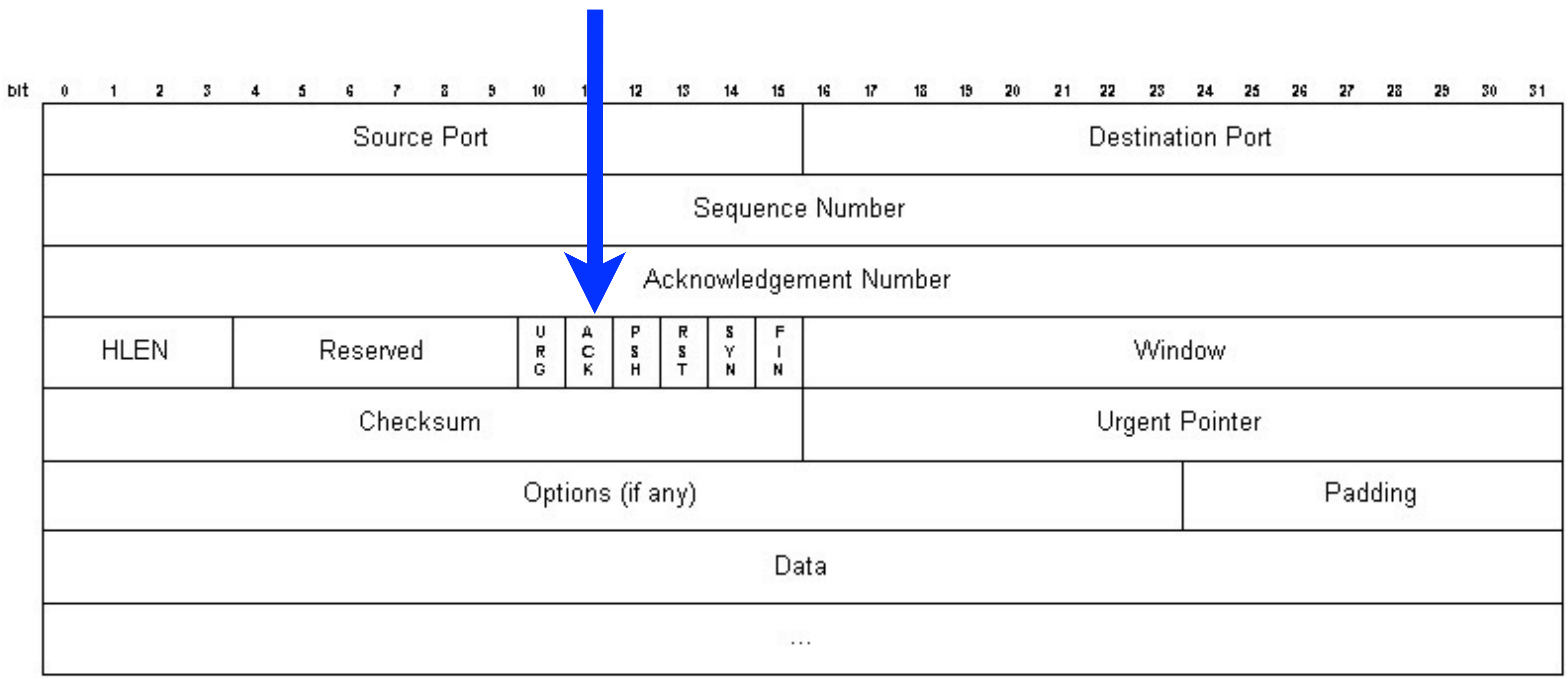
TCP Segment Structure



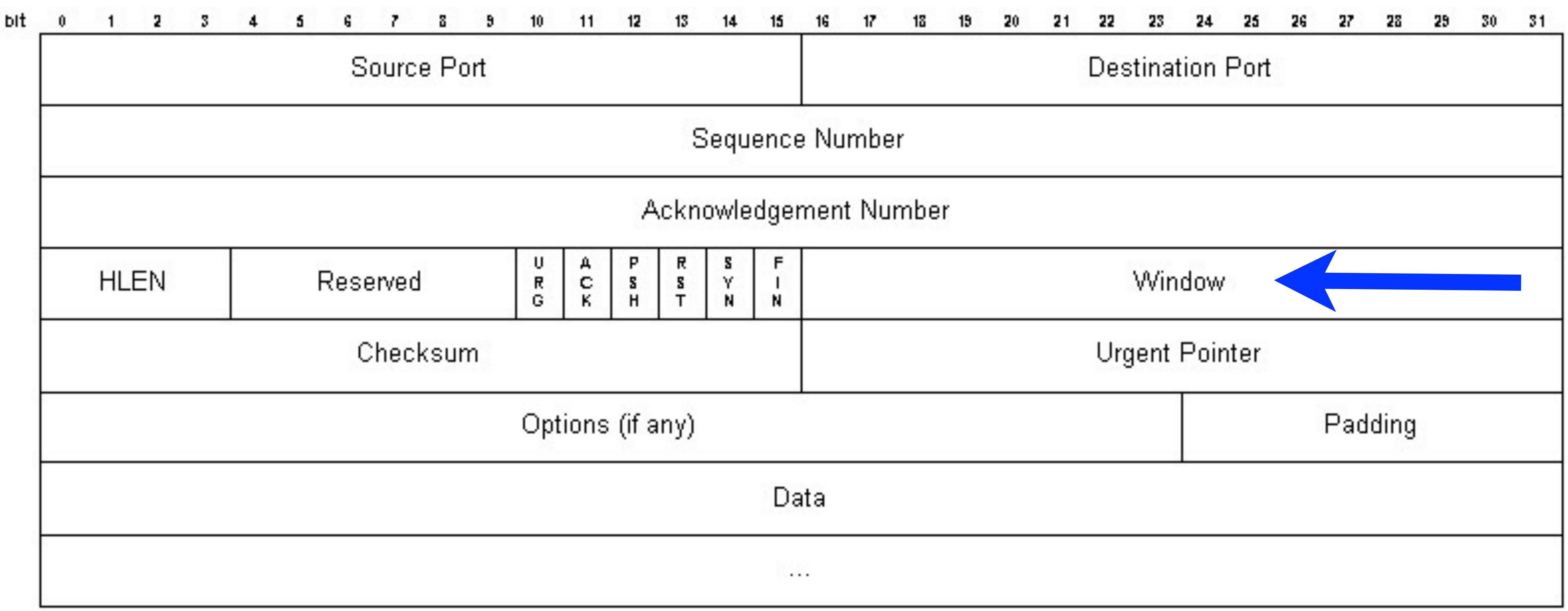
TCP Segment Structure



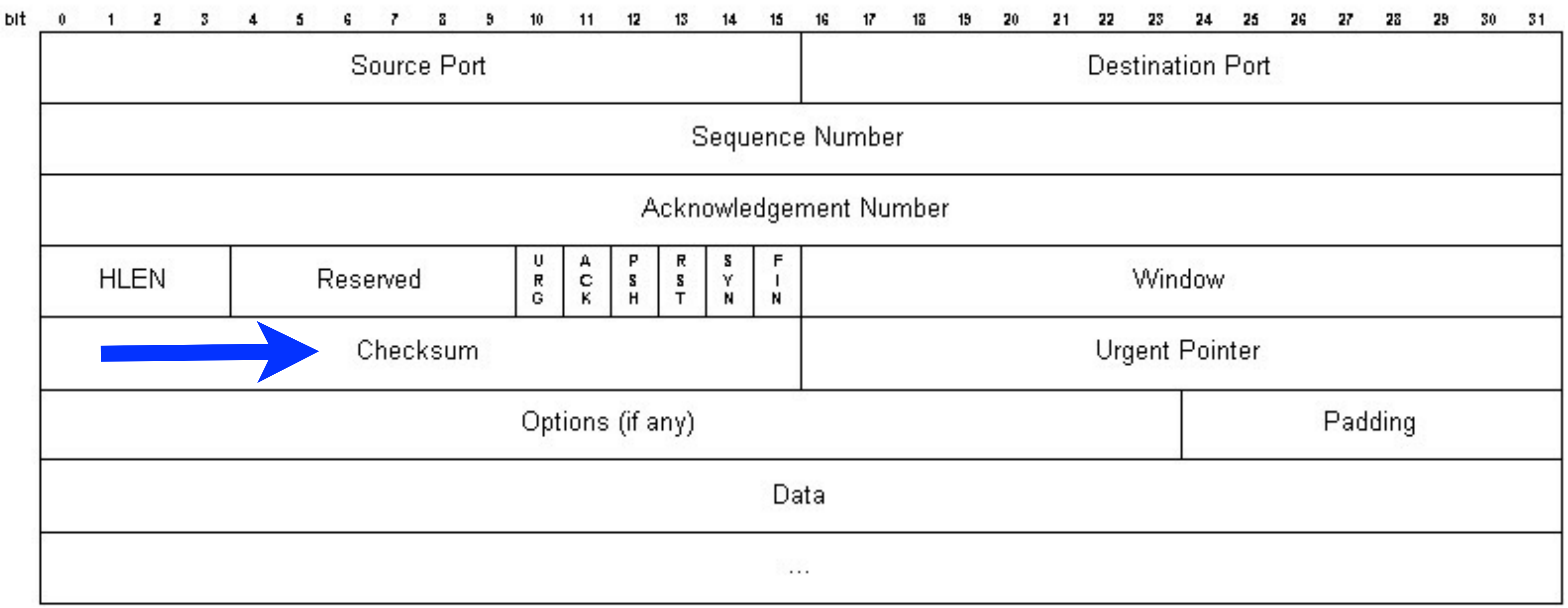
TCP Segment Structure



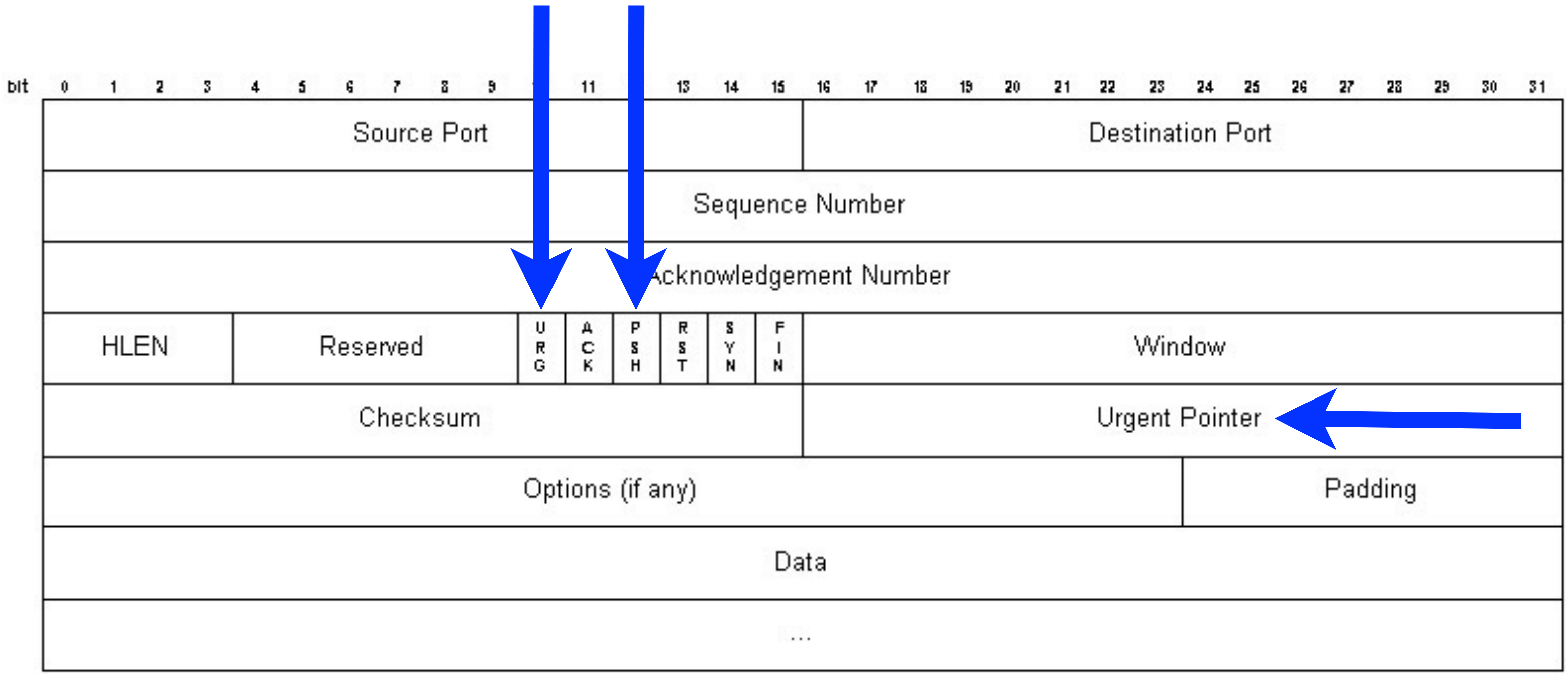
TCP Segment Structure



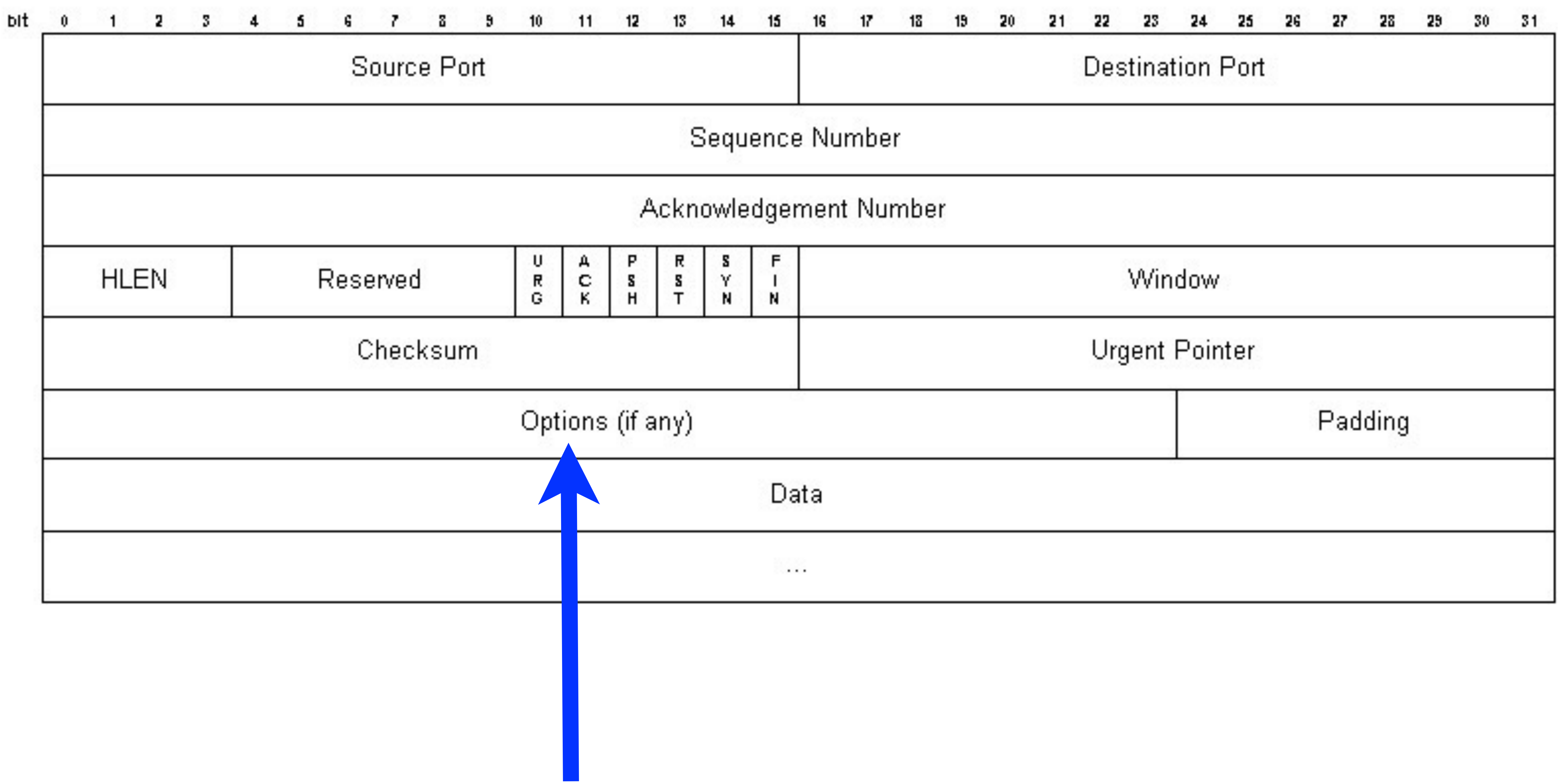
TCP Segment Structure



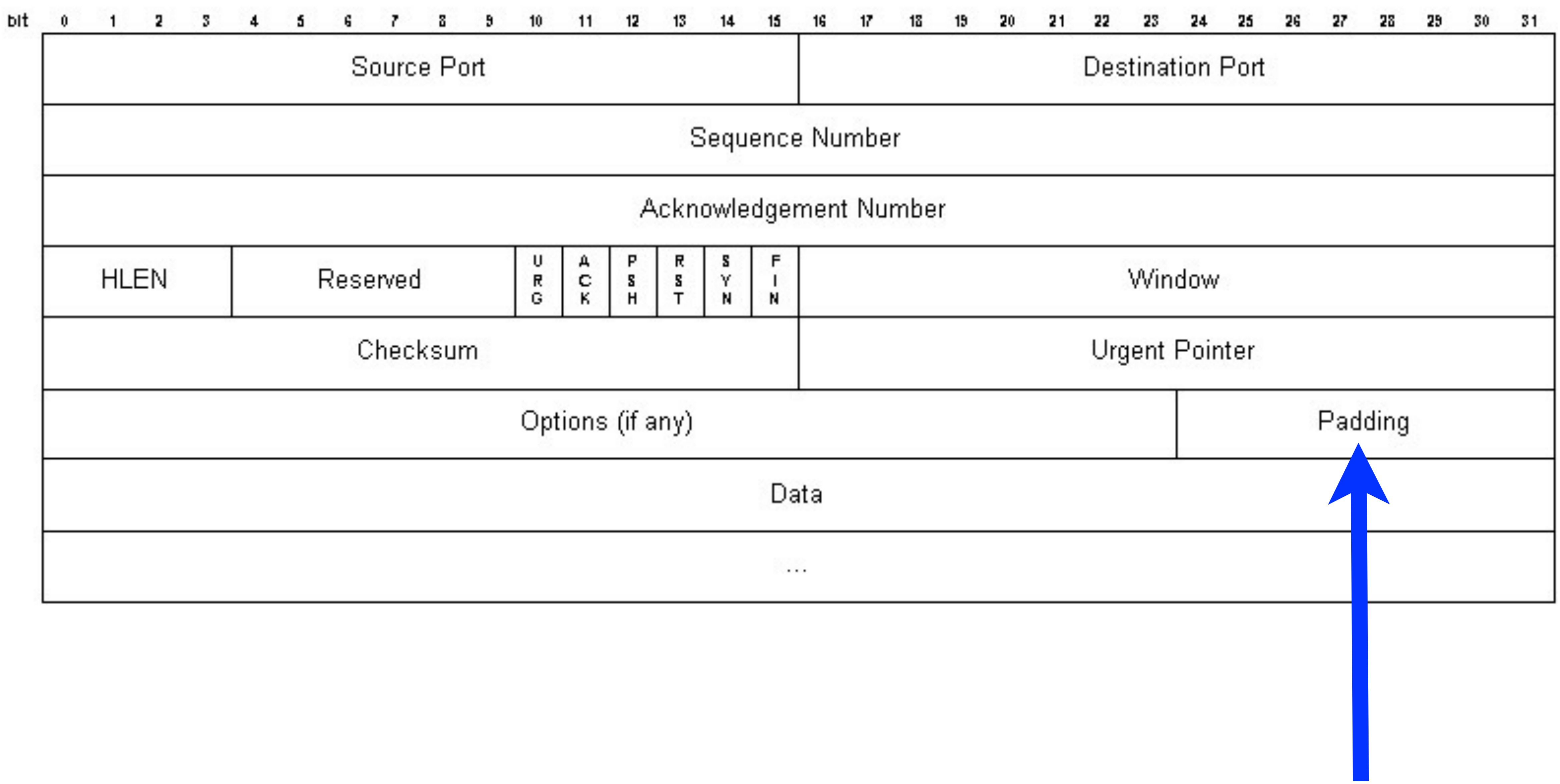
TCP Segment Structure



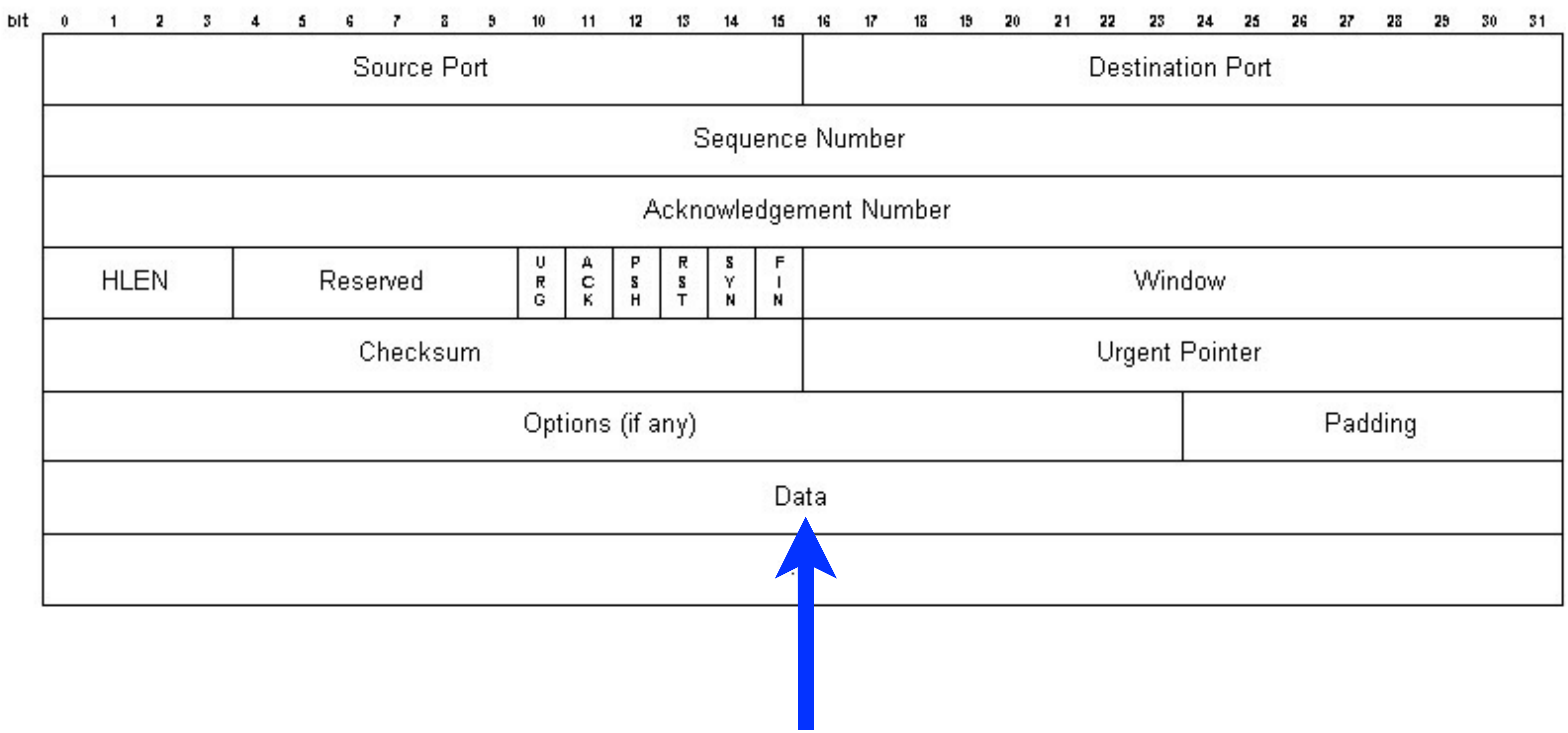
TCP Segment Structure



TCP Segment Structure



TCP Segment Structure



TCP seq. #'s and ACKs

TCP seq. #'s and ACKs

Seq. #'s:

- byte stream "number" of first byte in segment's data

TCP seq. #'s and ACKs

Seq. #'s:

- byte stream "number" of first byte in segment's data

ACKs:

- seq # of next byte expected from other side
- cumulative ACK

TCP seq. #'s and ACKs

Seq. #'s:

- byte stream "number" of first byte in segment's data

ACKs:

- seq # of next byte expected from other side
- cumulative ACK

Q: how receiver handles out-of-order segments

TCP seq. #'s and ACKs

Seq. #'s:

- byte stream "number" of first byte in segment's data

ACKs:

- seq # of next byte expected from other side
- cumulative ACK

Q: how receiver handles out-of-order segments

- A: TCP spec doesn't say, - up to implementor

TCP seq. #'s and ACKs

Seq. #'s:

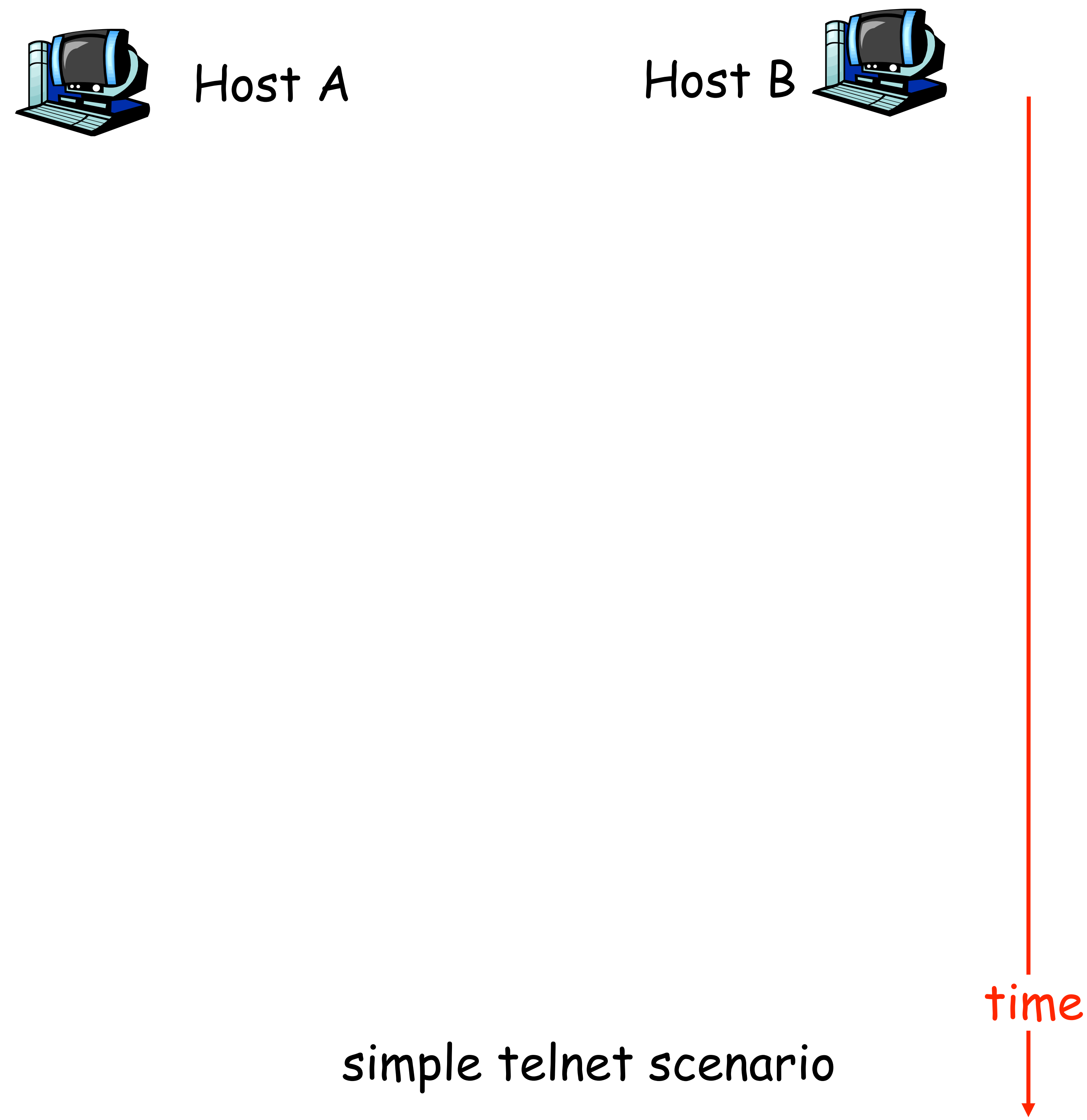
- byte stream "number" of first byte in segment's data

ACKs:

- seq # of next byte expected from other side
- cumulative ACK

Q: how receiver handles out-of-order segments

- A: TCP spec doesn't say, - up to implementor



simple telnet scenario

TCP seq. #'s and ACKs

Seq. #'s:

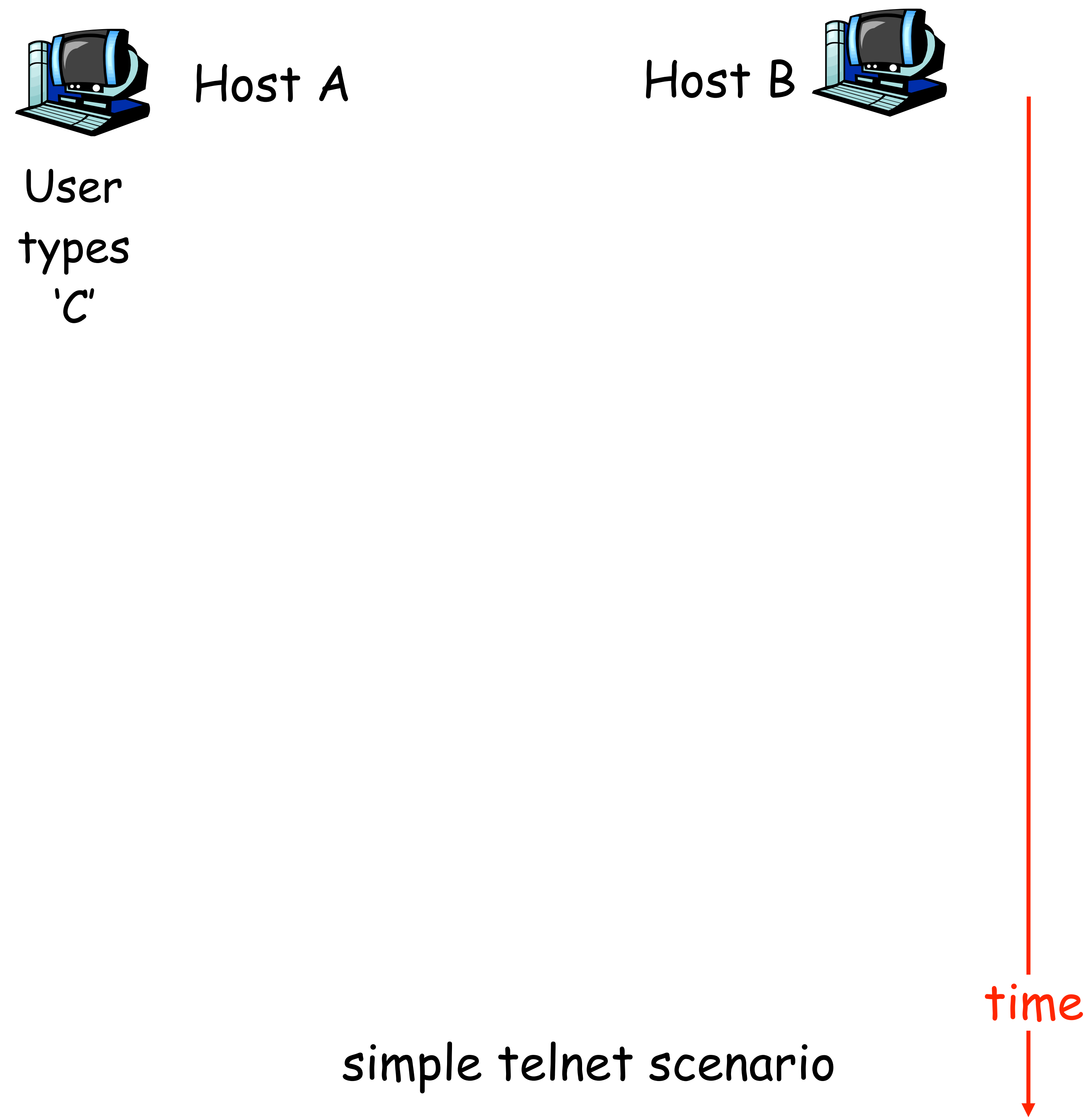
- byte stream "number" of first byte in segment's data

ACKs:

- seq # of next byte expected from other side
- cumulative ACK

Q: how receiver handles out-of-order segments

- A: TCP spec doesn't say, - up to implementor



simple telnet scenario

TCP seq. #'s and ACKs

Seq. #'s:

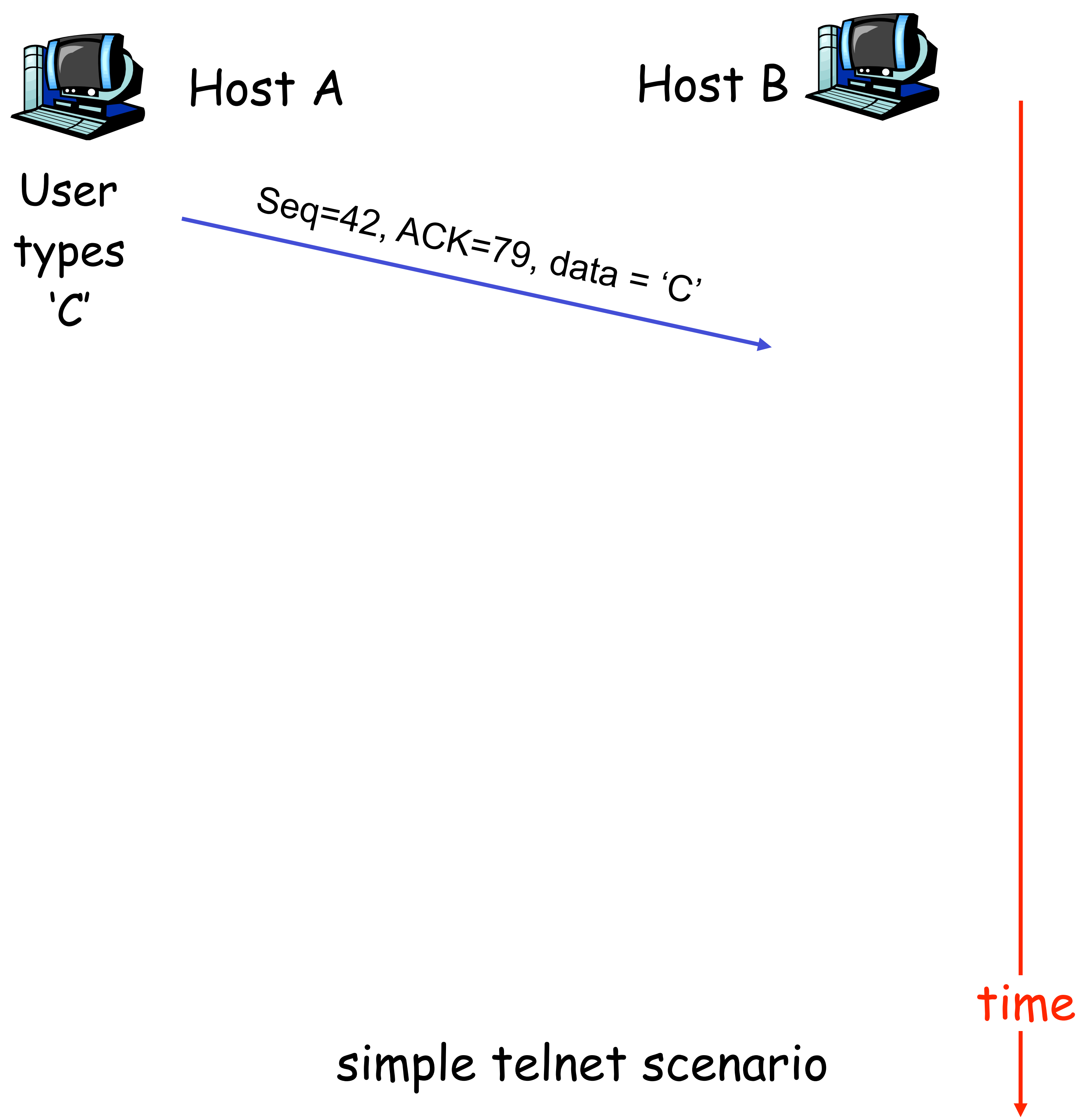
- byte stream "number" of first byte in segment's data

ACKs:

- seq # of next byte expected from other side
- cumulative ACK

Q: how receiver handles out-of-order segments

- A: TCP spec doesn't say, - up to implementor



simple telnet scenario

TCP seq. #'s and ACKs

Seq. #'s:

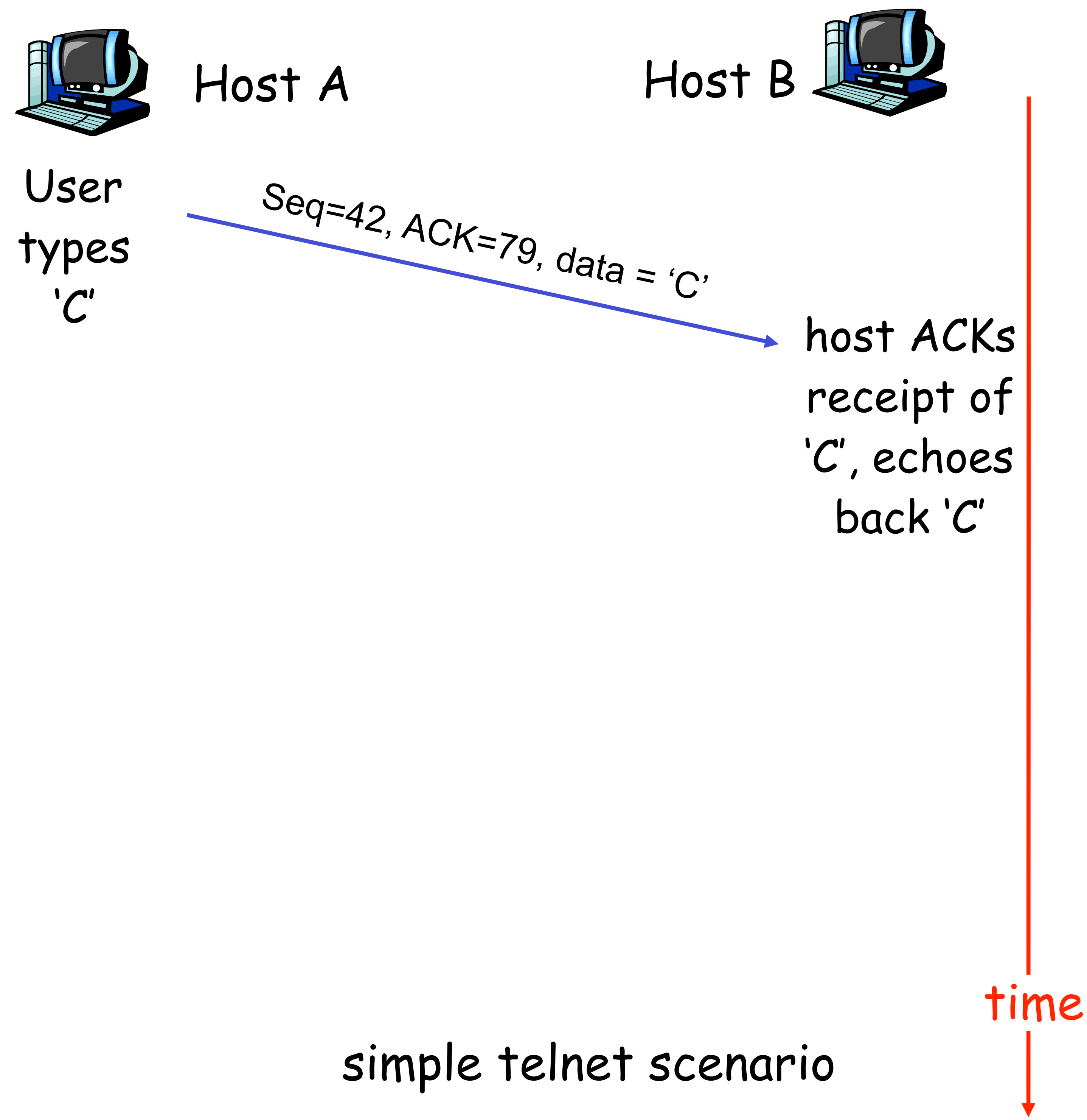
- byte stream "number" of first byte in segment's data

ACKs:

- seq # of next byte expected from other side
- cumulative ACK

Q: how receiver handles out-of-order segments

- A: TCP spec doesn't say, - up to implementor



simple telnet scenario

TCP seq. #'s and ACKs

Seq. #'s:

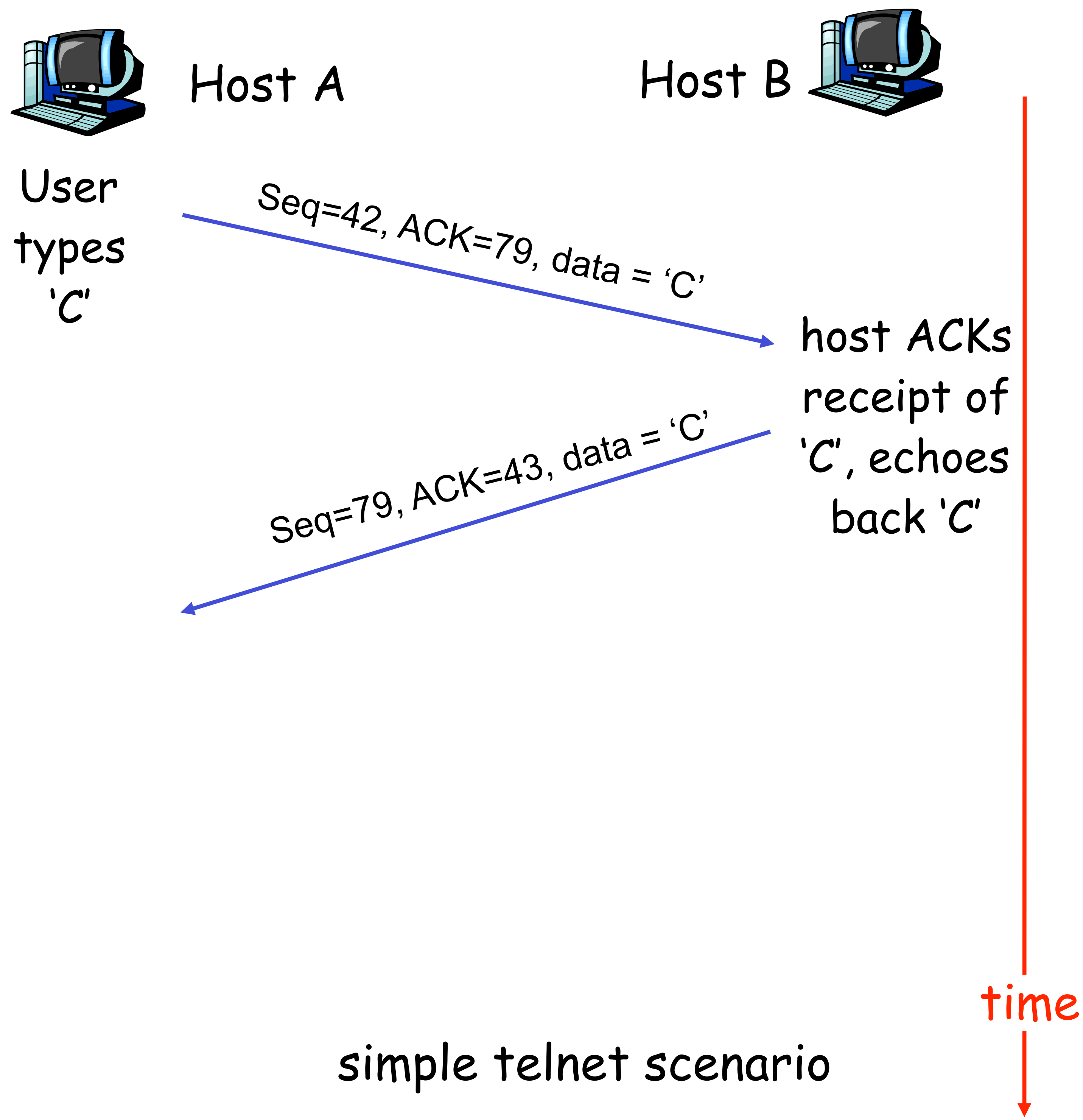
- byte stream "number" of first byte in segment's data

ACKs:

- seq # of next byte expected from other side
- cumulative ACK

Q: how receiver handles out-of-order segments

- A: TCP spec doesn't say, - up to implementor



simple telnet scenario

TCP seq. #'s and ACKs

Seq. #'s:

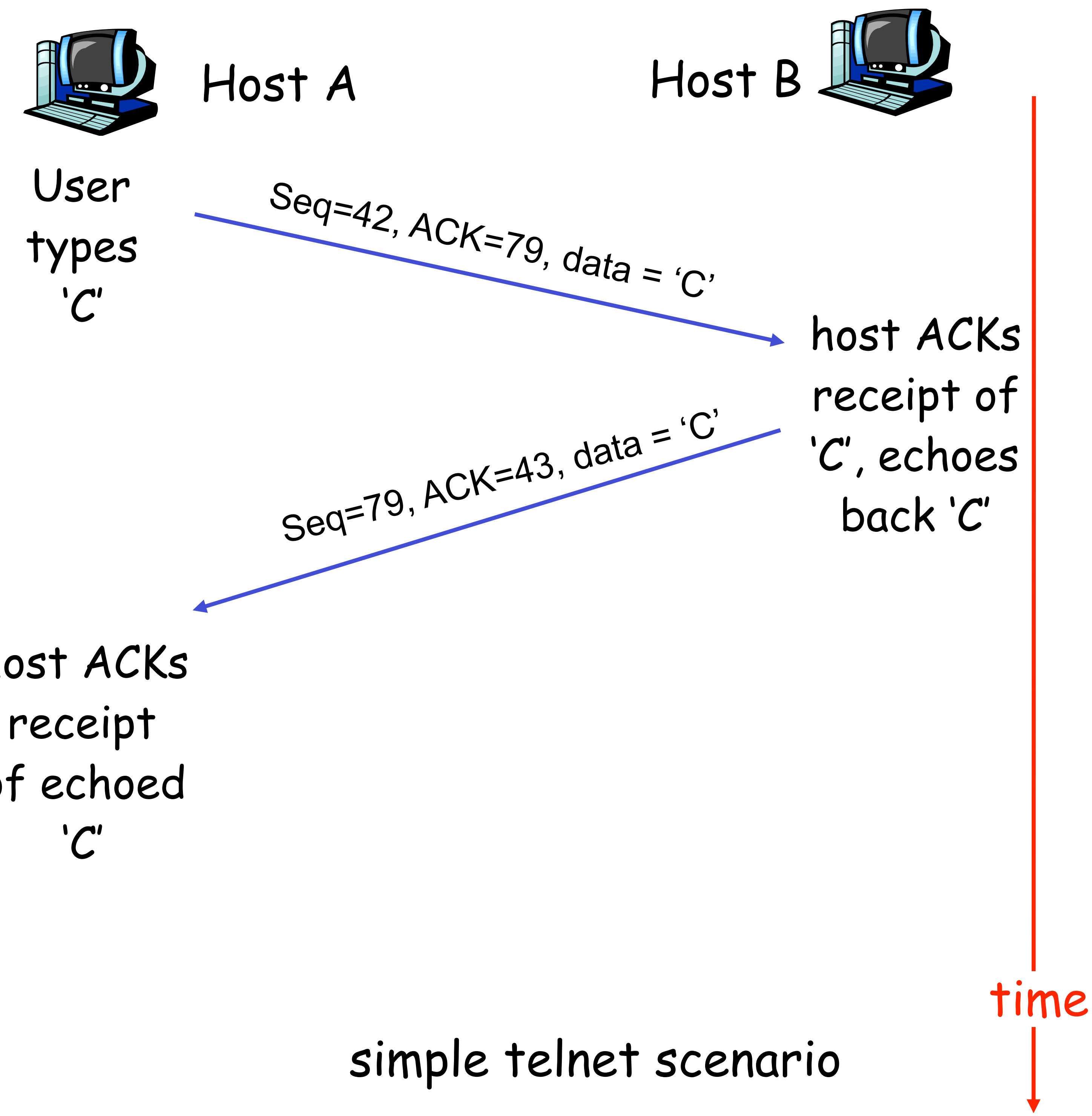
- byte stream "number" of first byte in segment's data

ACKs:

- seq # of next byte expected from other side
- cumulative ACK

Q: how receiver handles out-of-order segments

- A: TCP spec doesn't say, - up to implementor



simple telnet scenario

TCP seq. #'s and ACKs

Seq. #'s:

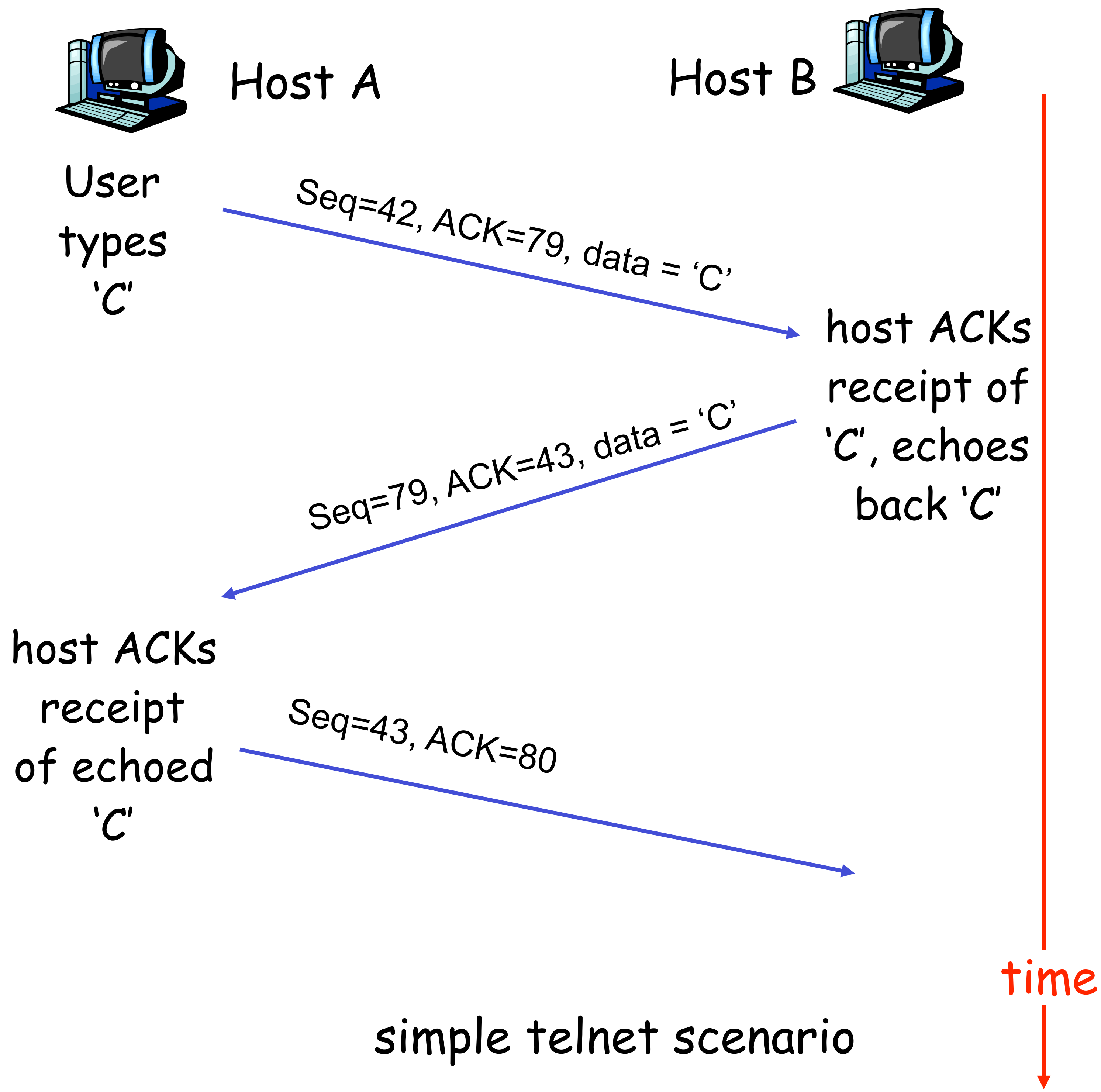
- byte stream "number" of first byte in segment's data

ACKs:

- seq # of next byte expected from other side
- cumulative ACK

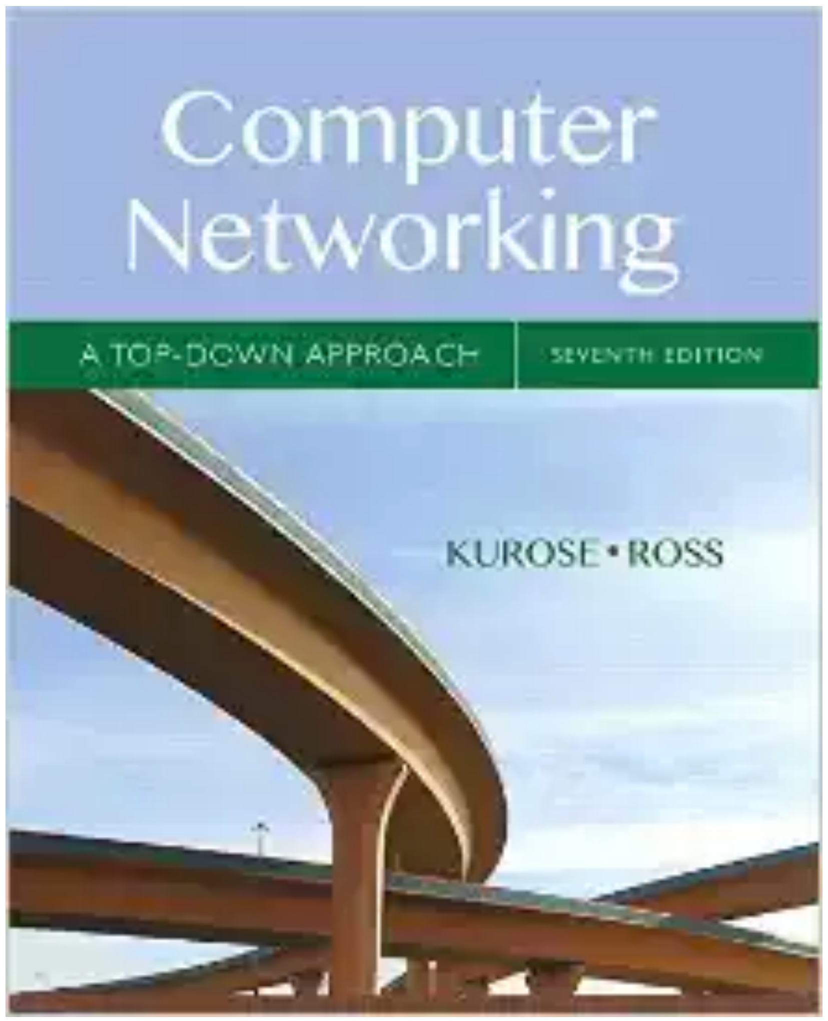
Q: how receiver handles out-of-order segments

- A: TCP spec doesn't say, - up to implementor



simple telnet scenario

Reading Along ...



- 3.5: Connection-oriented transport: TCP
    - connection management

TCP Connection Management

Recall: TCP sender, receiver establish “connection” before exchanging data segments

- initialize TCP variables:
  - seq. #s
  - buffers, flow control info (e.g. `RcvWindow`)
  - etc.

TCP Connection Management

Recall: TCP sender, receiver establish "connection" before exchanging data segments

- initialize TCP variables:
  - seq. #s
  - buffers, flow control info (e.g. `RcvWindow`)
  - etc.

Three way handshake:

TCP Connection Management

Recall: TCP sender, receiver establish "connection" before exchanging data segments

- initialize TCP variables:
  - seq. #s
  - buffers, flow control info (e.g. `RcvWindow`)
  - etc.

Three way handshake:

Step 1: client host sends TCP SYN segment to server

- specifies initial seq #
- no data

TCP Connection Management

Recall: TCP sender, receiver establish "connection" before exchanging data segments

- initialize TCP variables:
  - seq. #s
  - buffers, flow control info (e.g. **`RcvWindow`**)
  - etc.

Three way handshake:

Step 1: client host sends TCP SYN segment to server

- specifies initial seq #
- no data

Step 2: server host receives SYN, replies with SYNACK segment

- server allocates buffers
- specifies server initial seq. #

TCP Connection Management

Recall: TCP sender, receiver establish "connection" before exchanging data segments

- initialize TCP variables:
  - seq. #s
  - buffers, flow control info (e.g. `RcvWindow`)
  - etc.

Three way handshake:

Step 1: client host sends TCP SYN segment to server
- specifies initial seq #
- no data

Step 2: server host receives SYN, replies with SYNACK segment
- server allocates buffers
- specifies server initial seq. #

Step 3: client receives SYNACK, replies with ACK segment, which may contain data

TCP Connection Management (cont.)

TCP Connection Management (cont.)

Closing a connection:

TCP Connection Management (cont.)

Closing a connection:

client closes socket:

```
close(sd);
```

TCP Connection Management (cont.)

Closing a connection:

client closes socket:
```
close(sd);
```

Step 1: client end system sends TCP FIN control segment to server

TCP Connection Management (cont.)

Closing a connection:

client closes socket:
`close(sd);`

Step 1: client end system sends TCP FIN control segment to server

Step 2: server receives FIN, replies with ACK. Closes connection, sends FIN.

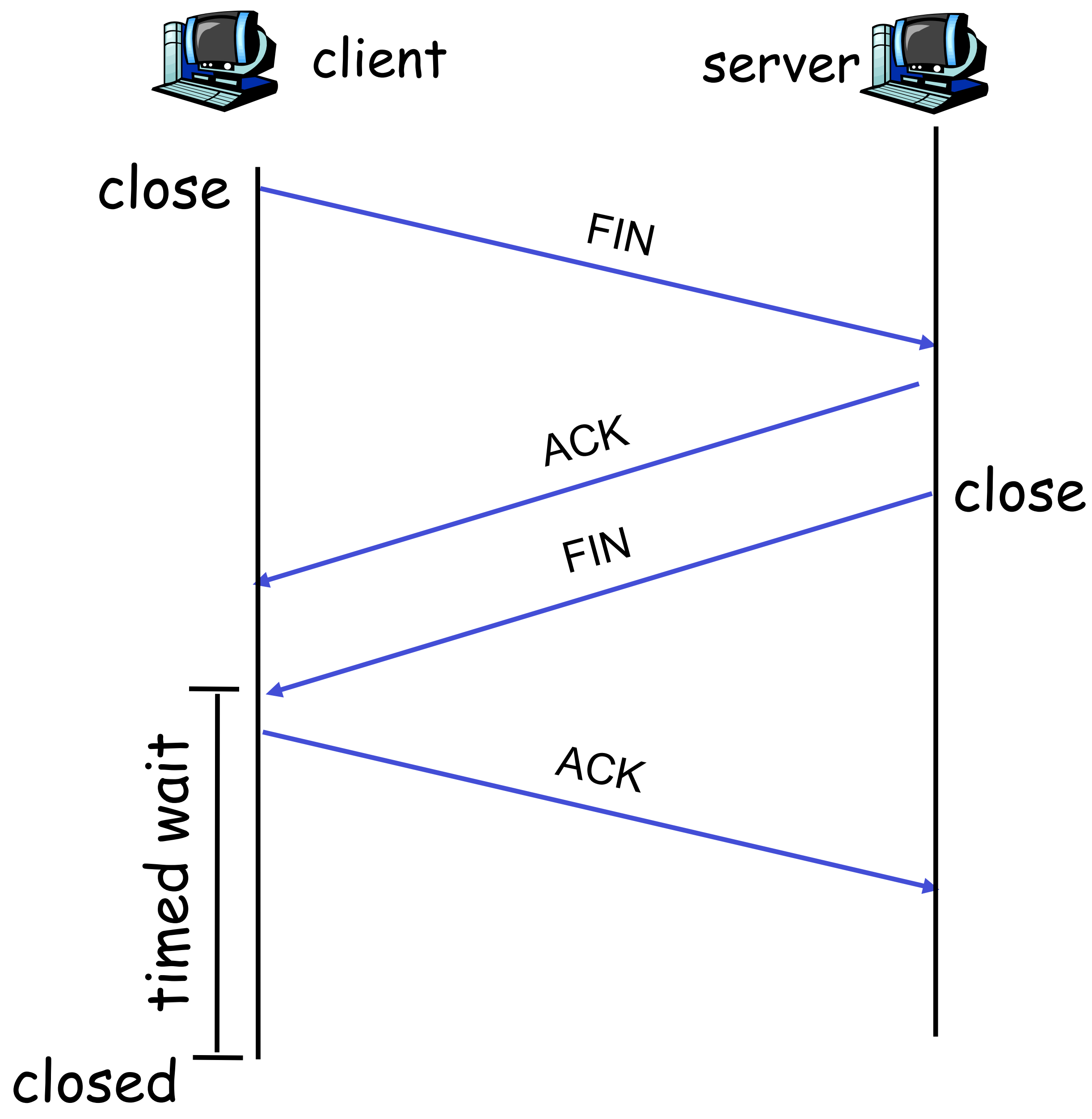
TCP Connection Management (cont.)

Closing a connection:

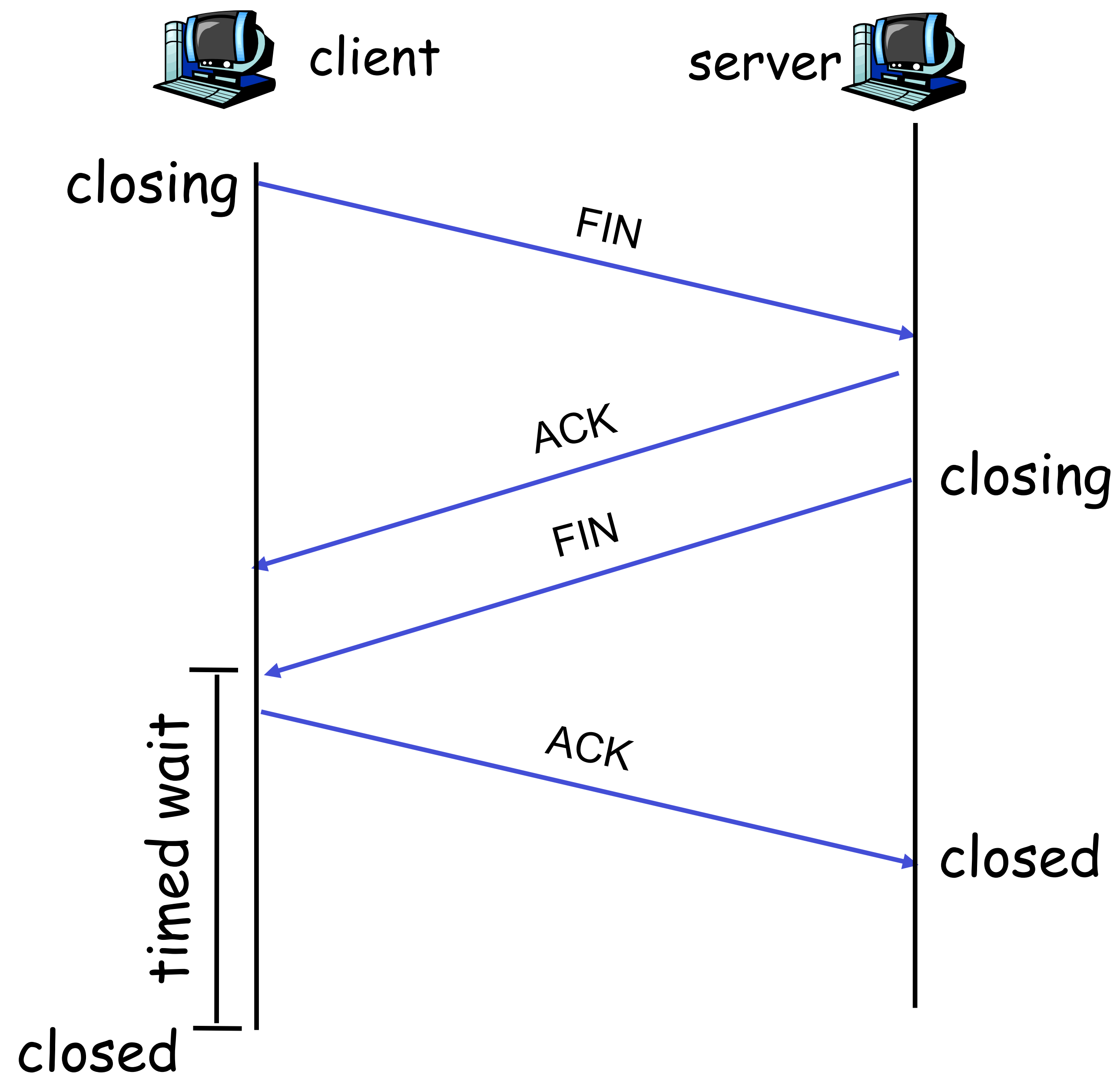
client closes socket: `close(sd);`

Step 1: client end system sends TCP FIN control segment to server

Step 2: server receives FIN, replies with ACK. Closes connection, sends FIN.



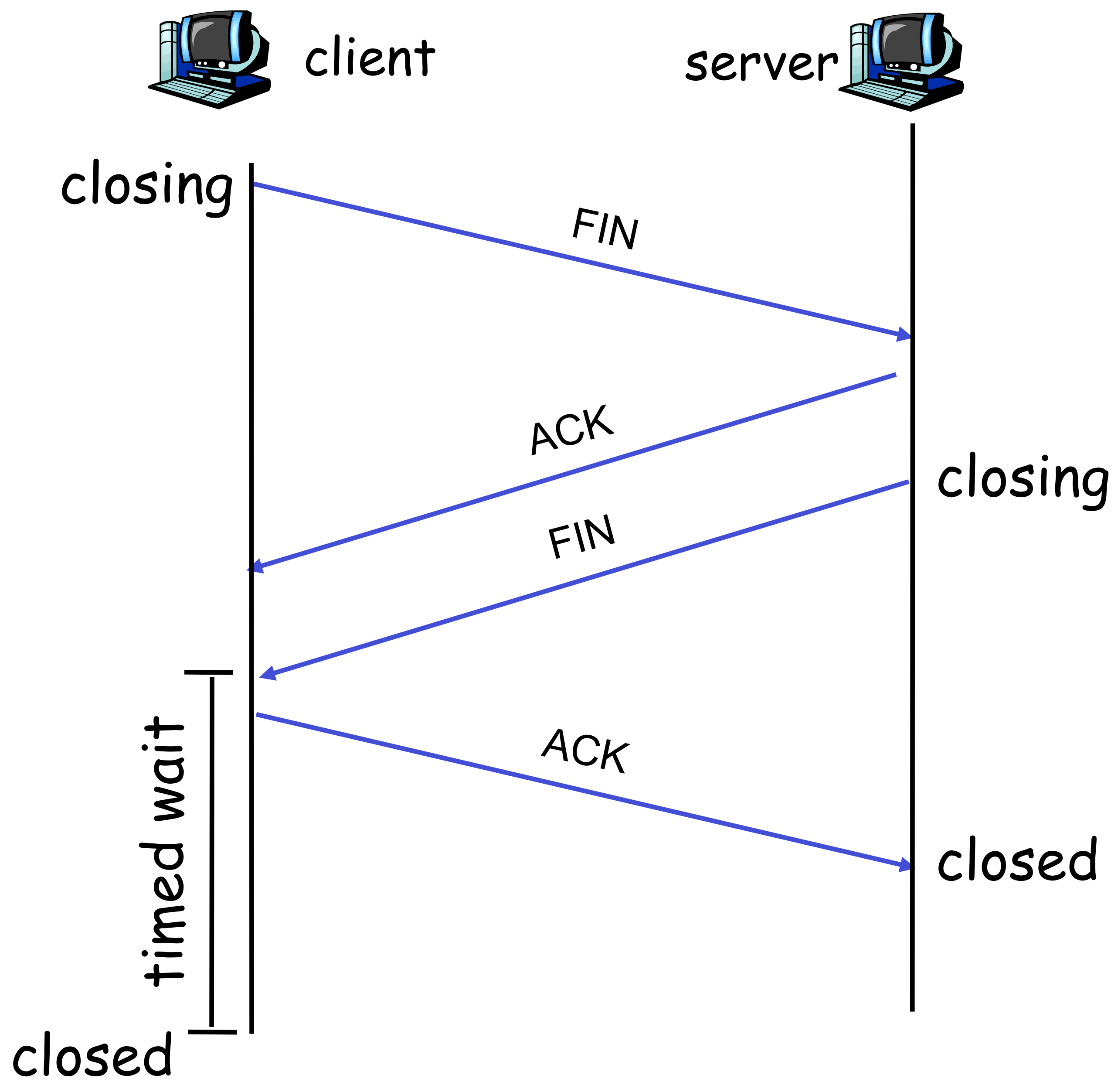
TCP Connection Management (cont.)



TCP Connection Management (cont.)

Step 3: client receives FIN, replies with ACK.

- Enters "timed wait" - will respond with ACK to received FINs

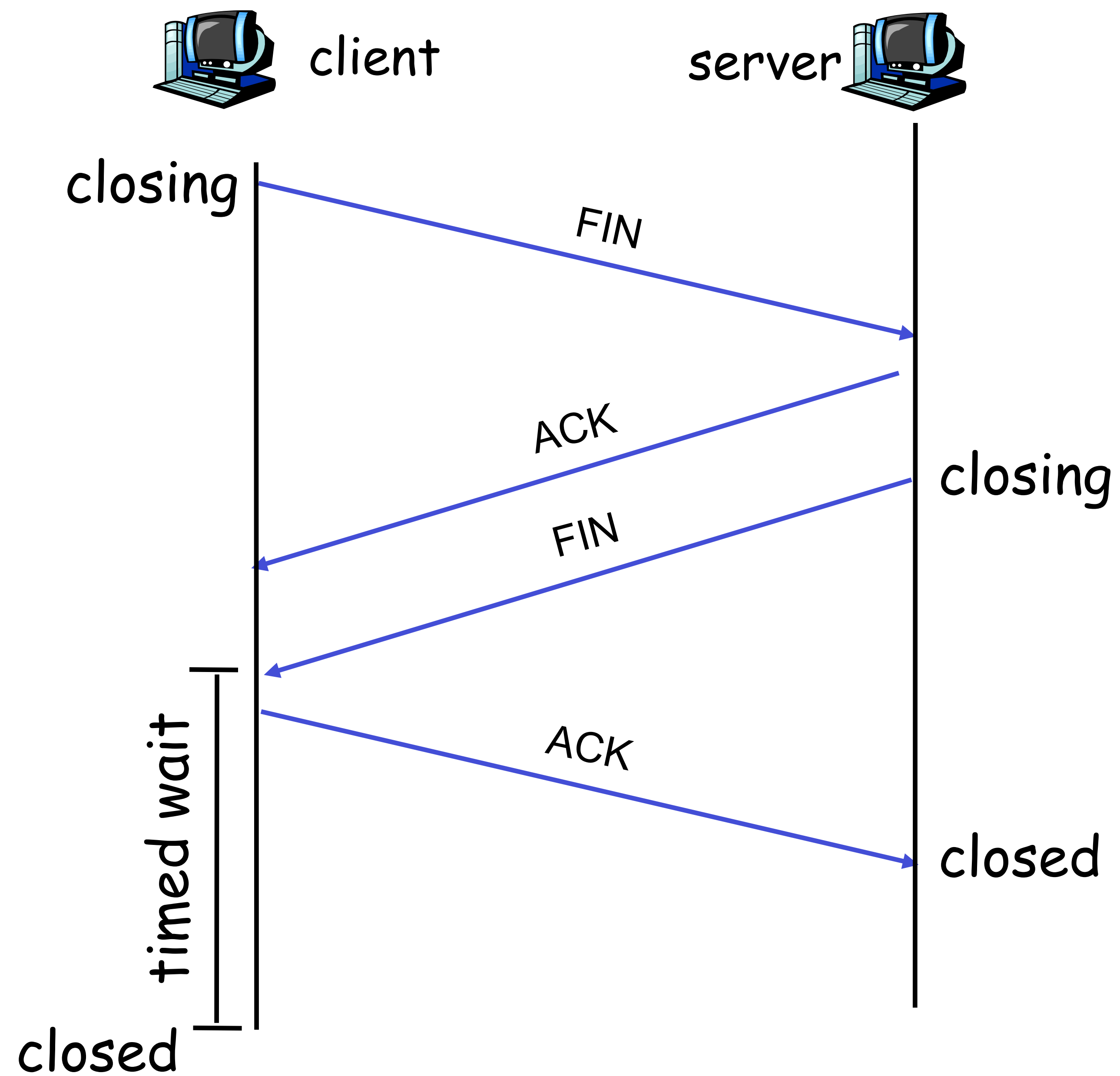


TCP Connection Management (cont.)

Step 3: client receives FIN, replies with ACK.

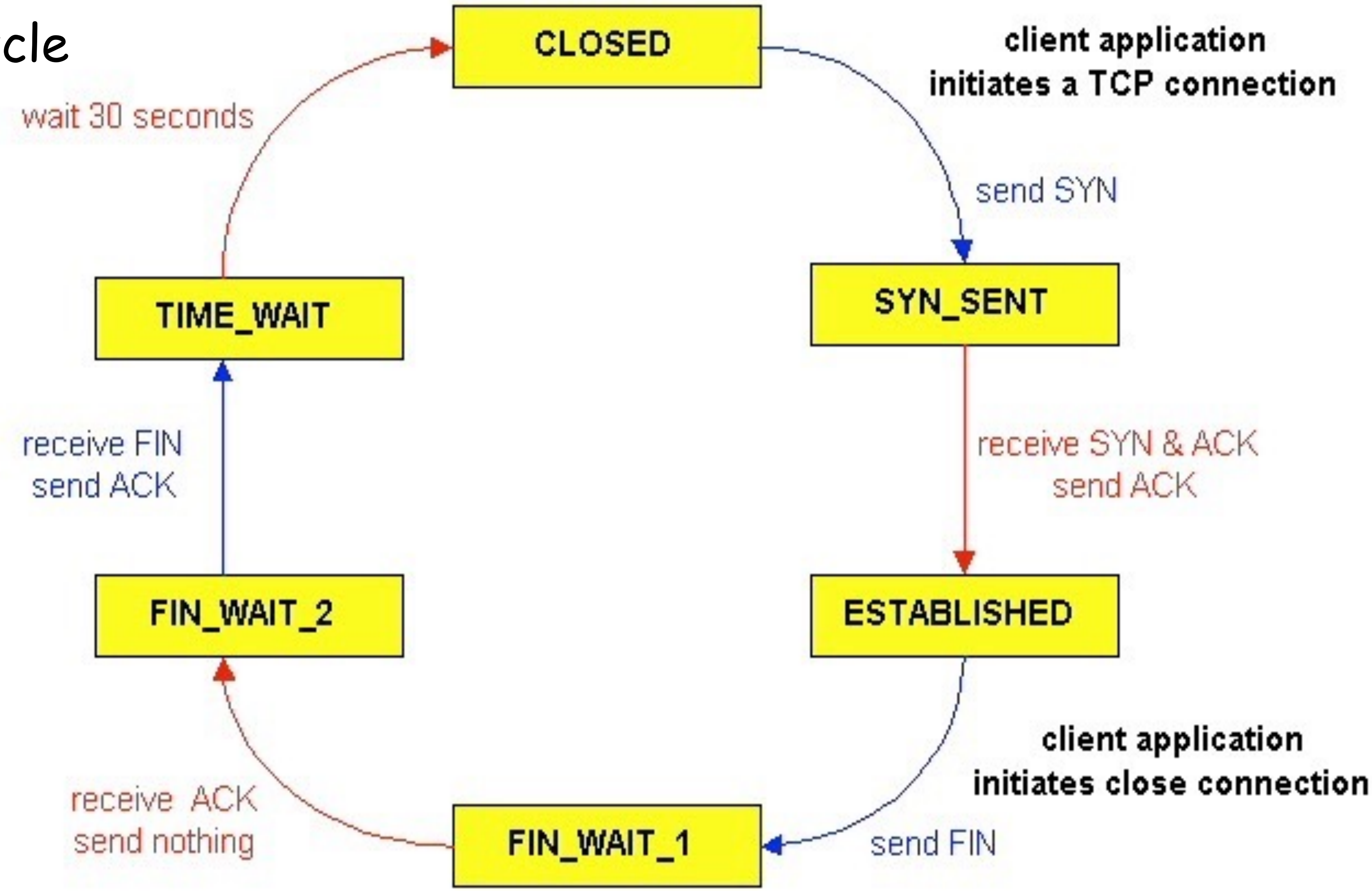
- Enters "timed wait" - will respond with ACK to received FINs

Step 4: server, receives ACK. Connection closed.



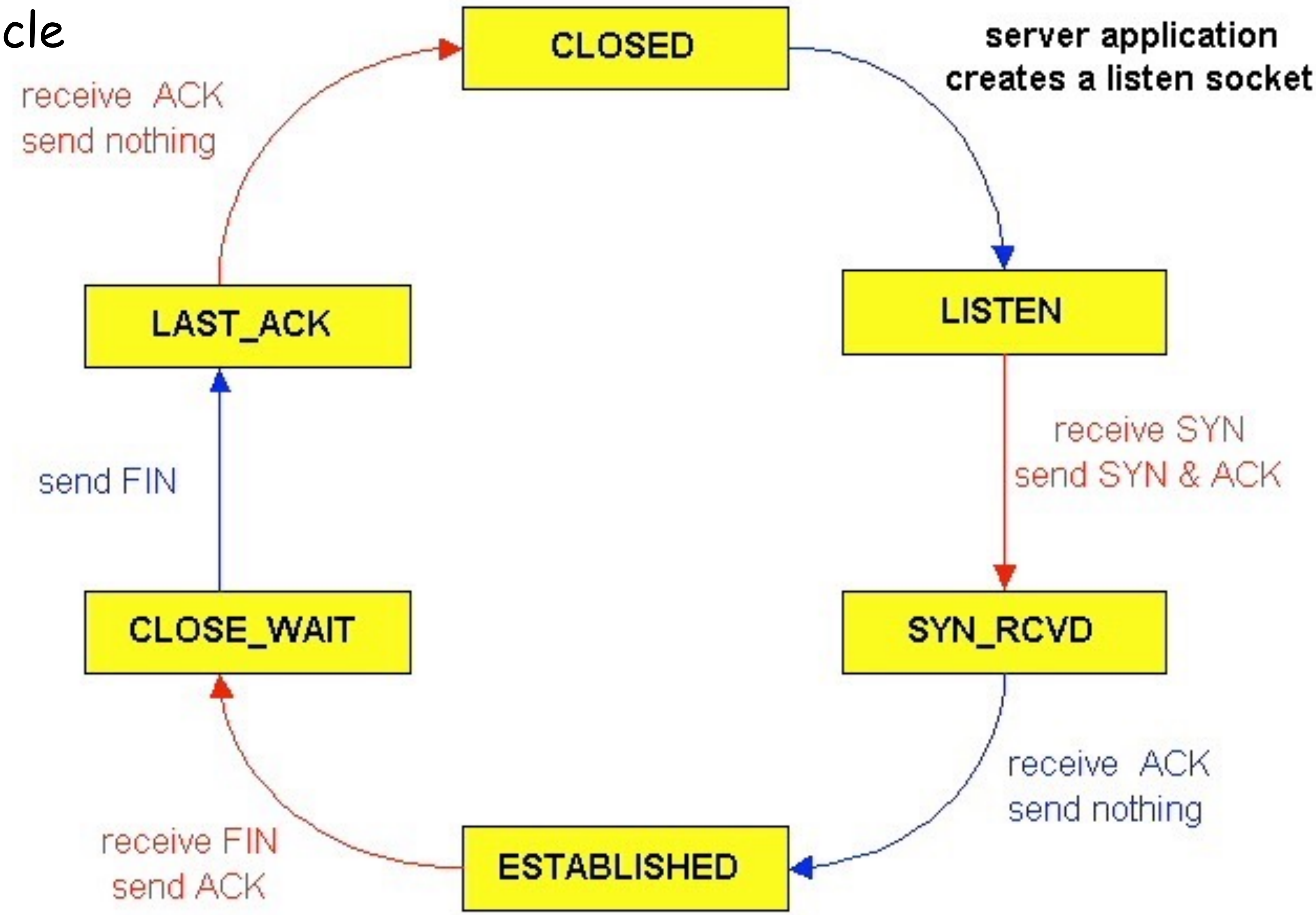
TCP Connection Management (cont)

TCP client lifecycle

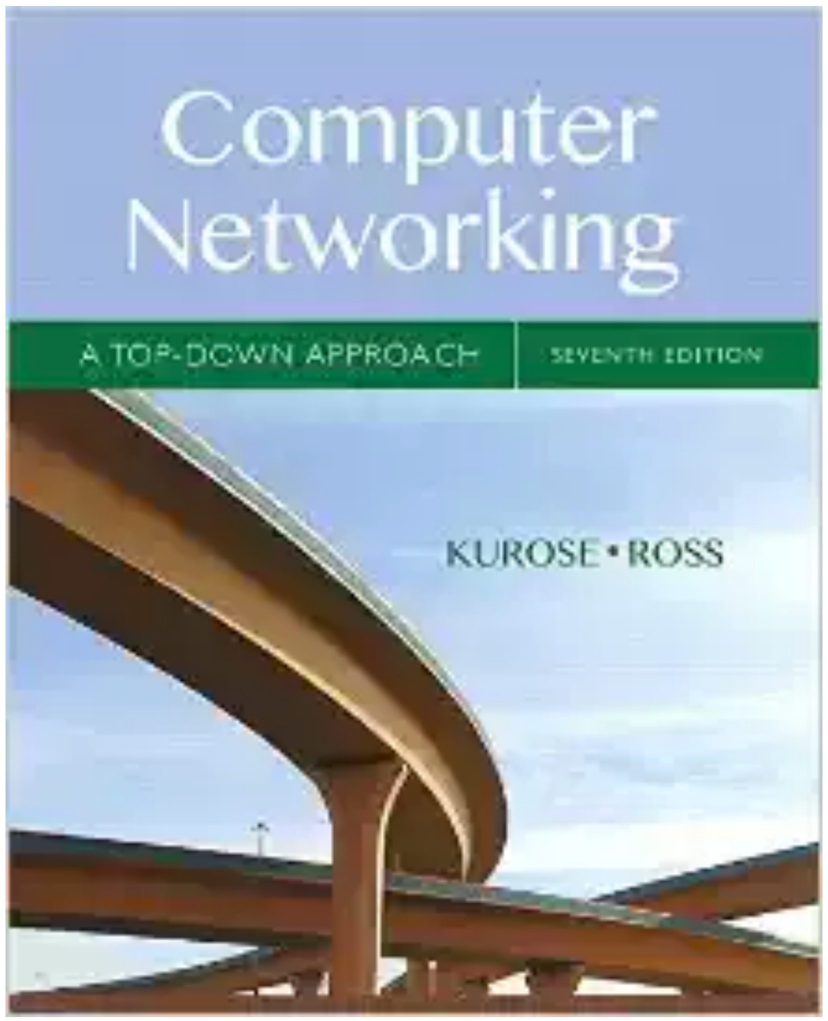


TCP Connection Management (cont)

TCP server lifecycle



Reading Along ...



- 3.5: Connection-oriented transport: TCP
  - timeouts

TCP Timeouts and the RTT

TCP Timeouts and the RTT

- Q: How to set TCP timeout value?

TCP Timeouts and the RTT

- Q: How to set TCP timeout value?
  - longer than RTT …

TCP Timeouts and the RTT

- Q: How to set TCP timeout value?
  - longer than RTT ...
  - ... but RTT varies

Measured RTTs

- ping measurements from eecslab-5

Measured RTTs

- ping measurements from eecslab-5

Remote Host	RTT (msec)

Measured RTTs

- ping measurements from eecslab-5

Remote Host	RTT (msec)
eecslab-6.case.edu	0.5

Measured RTTs

- ping measurements from eecslab-5

Remote Host	RTT (msec)
eecslab-6.case.edu	0.5
www.ohiou.edu	7.9

Measured RTTs

- ping measurements from eecslab-5

Remote Host	RTT (msec)
eecslab-6.case.edu	0.5
www.ohiou.edu	7.9
www.icir.org	71

Measured RTTs

- ping measurements from eecslab-5

Remote Host	RTT (msec)
eecslab-6.case.edu	0.5
www.ohiou.edu	7.9
www.icir.org	71
www.iij.ad.jp	171

TCP Timeouts and the RTT

TCP Timeouts and the RTT

- Q: How to set TCP timeout value?
  - longer than RTT …
  - … but RTT varies

TCP Timeouts and the RTT

- Q: How to set TCP timeout value?
  - longer than RTT ...
  - ... but RTT varies
- Q: What if we set the timeout too short?

TCP Timeouts and the RTT

- Q: How to set TCP timeout value?
  - longer than RTT ...
  - ... but RTT varies
- Q: What if we set the timeout too short?
  - premature timeout
  - unnecessary retransmissions

TCP Timeouts and the RTT

- Q: How to set TCP timeout value?
  - longer than RTT …
  - … but RTT varies
- Q: What if we set the timeout too short?
  - premature timeout
  - unnecessary retransmissions
- Q: What if we set the timeout too long?

TCP Timeouts and the RTT

- Q: How to set TCP timeout value?
  - longer than RTT …
  - … but RTT varies
- Q: What if we set the timeout too short?
  - premature timeout
  - unnecessary retransmissions
- Q: What if we set the timeout too long?
  - slow reaction to loss

TCP Timeouts and the RTT

TCP Timeouts and the RTT

- Q: How do we estimate the RTT?

TCP Timeouts and the RTT

- Q: How do we estimate the RTT?
  - measure time between data transmission and ACK

TCP Timeouts and the RTT

- Q: How do we estimate the RTT?
  - measure time between data transmission and ACK
  - do not time retransmissions
    - (Karn/Partridge algorithm)

TCP Timeouts and the RTT

- Q: How do we estimate the RTT?
  - measure time between data transmission and ACK
  - do not time retransmissions
    - (Karn/Partridge algorithm)
- RTT will vary across measurements

TCP Timeouts and the RTT

- Q: How do we estimate the RTT?
  - measure time between data transmission and ACK
  - do not time retransmissions
    - (Karn/Partridge algorithm)
- RTT will vary across measurements
  - want something smoother
  - instead of just the current RTT, use an average of the last few measurements
  - the "smoothed RTT" (SRTT)

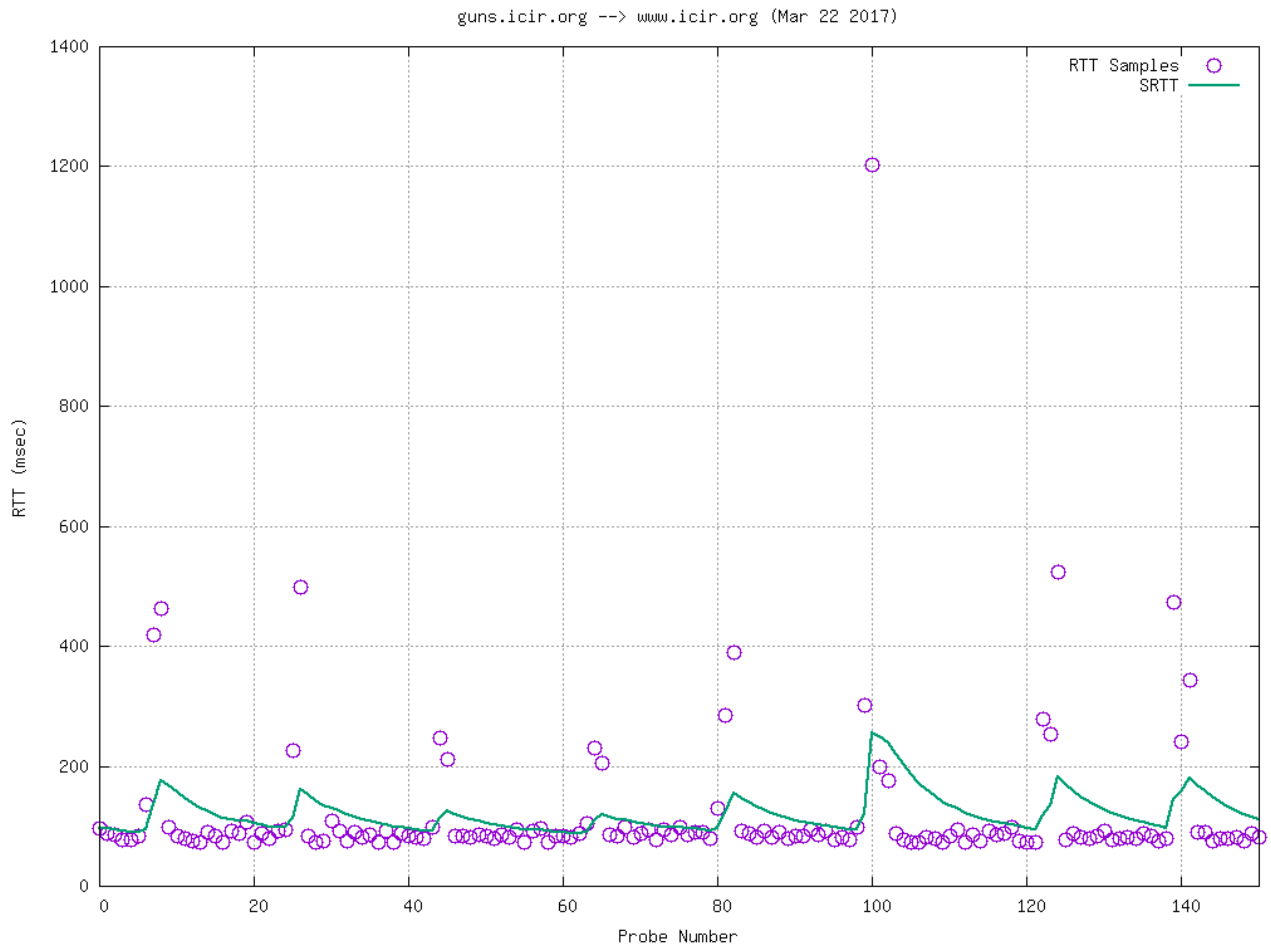
TCP RTO

- RFC 6298

$$\mathbf{SRTT = (1-\ \alpha)*SRTT + \alpha*R}$$

- Exponential weighted moving average
- influence of past sample decreases exponentially fast
- standard value: $\alpha = 0.125$

Example SRTT Computation



TCP RTO

TCP RTO

Setting the timeout

- `SRTT` plus "safety margin"
  - large variation in `SRTT` -> larger safety margin

TCP RTO

Setting the timeout

- **SRTT** plus "safety margin"
  - large variation in **SRTT** -> larger safety margin

- first, estimate of how much SRTT deviates from the RTT sample:

TCP RTO

Setting the timeout

- **SRTT** plus "safety margin"
  - large variation in **SRTT** -> larger safety margin

- first, estimate of how much SRTT deviates from the RTT sample:

$$\mathbf{RTTVAR = (1-\beta)*RTTVAR + \beta*|R-SRTT|}$$

(β = 0.25, per standard)

TCP RTO

Setting the timeout

- **SRTT** plus "safety margin"
  - large variation in **SRTT** -> larger safety margin

- first, estimate of how much SRTT deviates from the RTT sample:

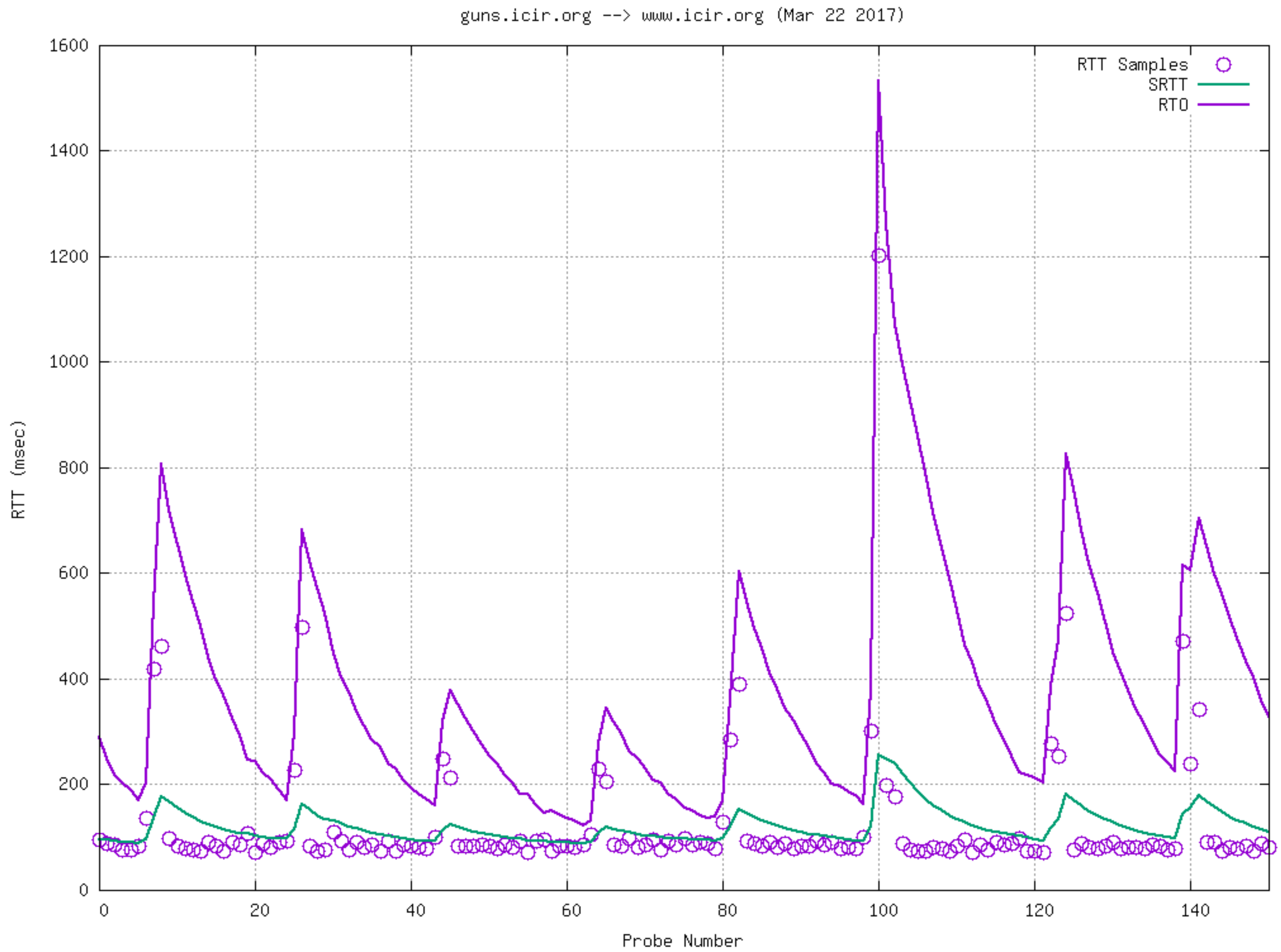
RTTVAR = (1-β)*RTTVAR + β*|R-SRTT|

(β = 0.25, per standard)

Then set timeout interval:

RTO = SRTT + 4*RTTVAR

Example RTO Computation



TCP RTO

TCP RTO

- But, one issue remains ...

TCP RTO

- But, one issue remains ...
  - where do we start?!

TCP RTO

- But, one issue remains ...
  - where do we start?!

- RFC 1122, RFC 2988 specify 3 seconds
  - why?

TCP RTO

- But, one issue remains ...
  - where do we start?!

- RFC 1122, RFC 2988 specify 3 seconds
  - why?

- RFC 1122 published in 1989. RFC 2988 published in 2000.

TCP RTO

- But, one issue remains ...
  - where do we start?!
- RFC 1122, RFC 2988 specify 3 seconds
  - why?
- RFC 1122 published in 1989. RFC 2988 published in 2000.
  - does this constant still apply?

TCP Initial RTO

TCP Initial RTO

- Matt Sargent
- Case PhD, 2015

TCP Initial RTO

- Matt Sargent
- Case PhD, 2015

- Hypothesizes that an initial RTO developed in 1989 might not be appropriate anymore

- Conducts an empirical investigation

TCP Initial RTO

TCP Initial RTO

- How do we choose?

TCP Initial RTO

- How do we choose?
  - look at some data!
  - investigated eight datasets (spanning 6 years)
  - datasets are packet traces!

TCP Initial RTO

- How do we choose?
  - look at some data!
  - investigated eight datasets (spanning 6 years)
  - datasets are packet traces!
- Lowering the InitRTO from 3 seconds to 1 second has cost:
  - .... less than 0.1% of the connections spuriously retransmit the SYN in most datasets
  - .... 1.1% of the connections in a wireless network spuriously retransmit the SYN

TCP Initial RTO

TCP Initial RTO

- Connections experience performance boost when SYN is lost and requires retransmission
  - i.e., connections only wait for 1sec when previously waited 3sec

TCP Initial RTO

- Connections experience performance boost when SYN is lost and requires retransmission
  - i.e., connections only wait for 1sec when previously waited 3sec

- Connections realizing >= 10% boost:
  - 10-43% (across datasets)
- Connections realizing >= 50% boost:
  - 17-73% (across datasets)

TCP Initial RTO

- The IETF decided that Matt's evidence was compelling

- The initial RTO was lowered from 3sec to 1sec
  - specified in RFC 6298

- Details of Matt's investigation are given in appendix of RFC 6298

TCP RTO

TCP RTO

- One more thing ...
- What if we retransmit and the resent segment is dropped?

TCP RTO

- One more thing ...

- What if we retransmit and the resent segment is dropped?
  - backoff the RTO
  - RTO *= 2
  - (similar to collision backoff in Ethernet)

- Part of TCP's congestion control mechanism

Magic Numbers

Magic Numbers

- Where do the constants in the RTO mechanism come from?
  - e.g., alpha = 0.125
  - e.g., initRTO = 1sec
  - e.g., we multiple RTTVAR by 4

Magic Numbers

- Where do the constants in the RTO mechanism come from?
  - e.g., alpha = 0.125
  - e.g., initRTO = 1sec
  - e.g., we multiple RTTVAR by 4

- Rough consensus …
  - but, it still stinks
  - we strive for mechanisms that do not require magic constants