



# Computer Networks

## CMSC 417 : Spring 2024

COMPUTER SCIENCE
UNIVERSITY OF MARYLAND

## Transport Layer Protocols (UDP, TCP)
## (Textbook chapter 5)

Nirupam Roy

**Tu-Th 2:00-3:15pm**
**CSI 2117**

**March 7th, 2024**




## Connection-less mux/demux:

- User Datagram Protocol (UDP)

## Connection-oriented mux/demux:

- Transmission Control Protocol (TCP)

# Connectionless demultiplexing

- Create sockets with port numbers:

```
DatagramSocket mySocket1 = new
   DatagramSocket(99111);
DatagramSocket mySocket2 = new
   DatagramSocket(99222);
```

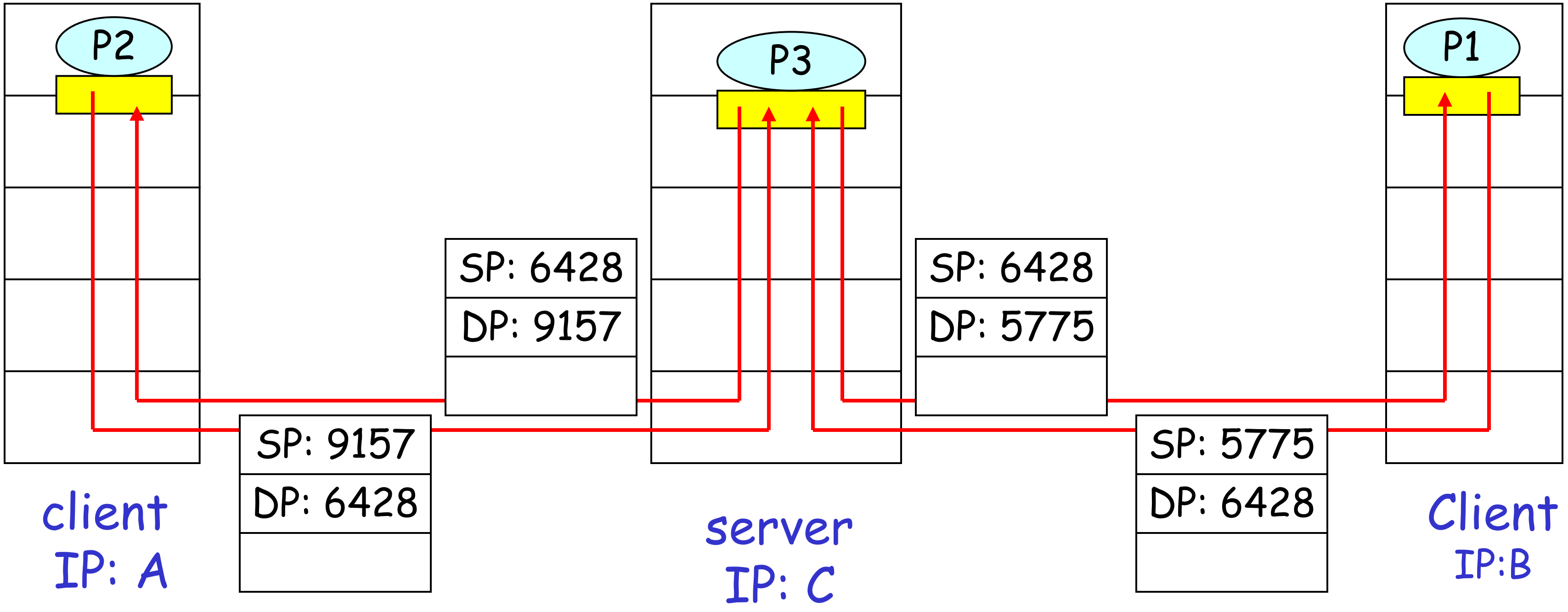
- UDP socket identified by two-tuple:

(dest IP address, dest port number)

- When host receives UDP segment:
  - checks destination port number in segment
  - directs UDP segment to socket with that port number
- IP datagrams with different source IP addresses and/or source port numbers directed to same socket

# Connectionless demux (cont)

```
DatagramSocket serverSocket = new DatagramSocket(6428);
```



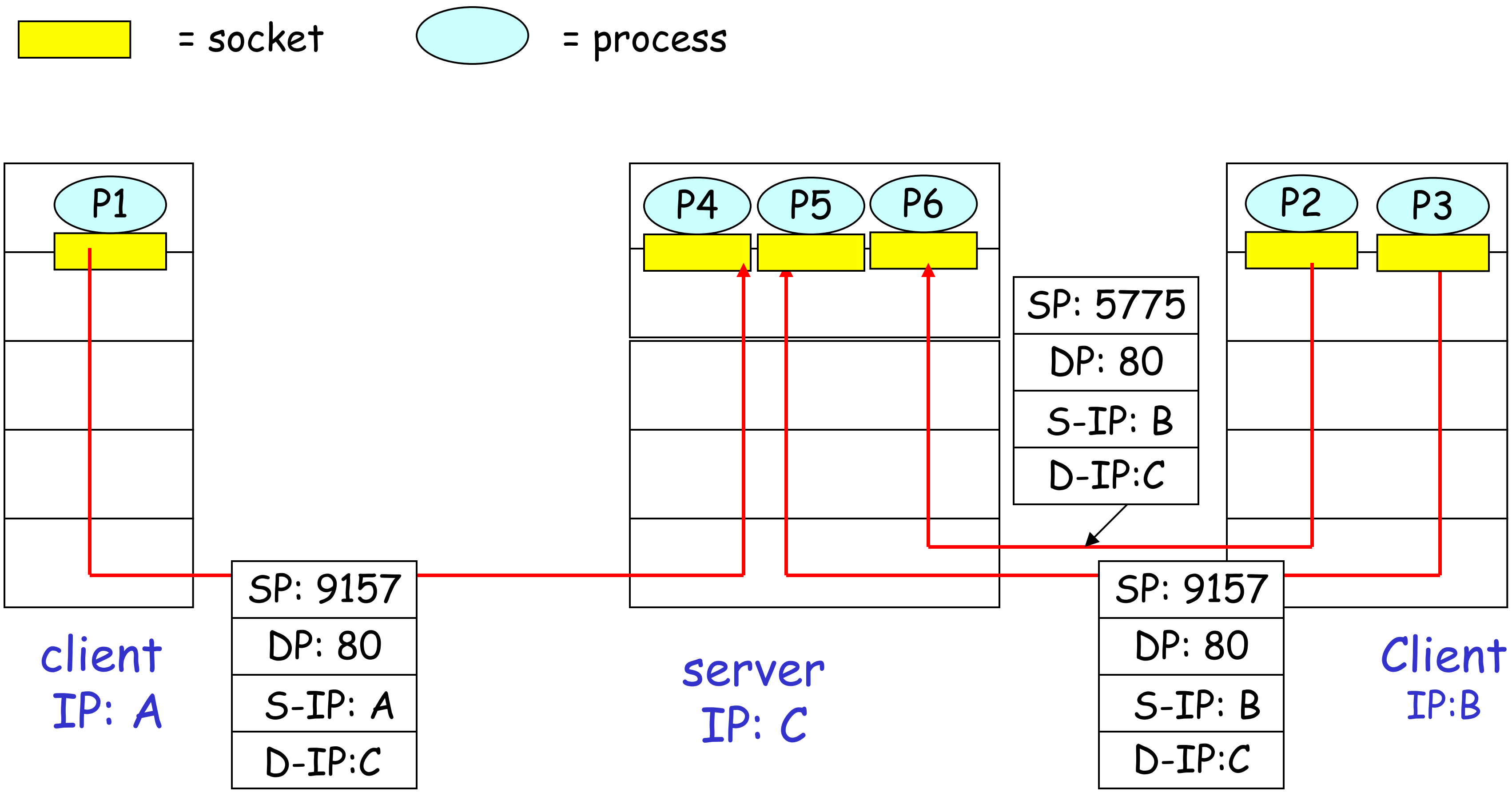

SP provides "return address"

# Connection-oriented demux

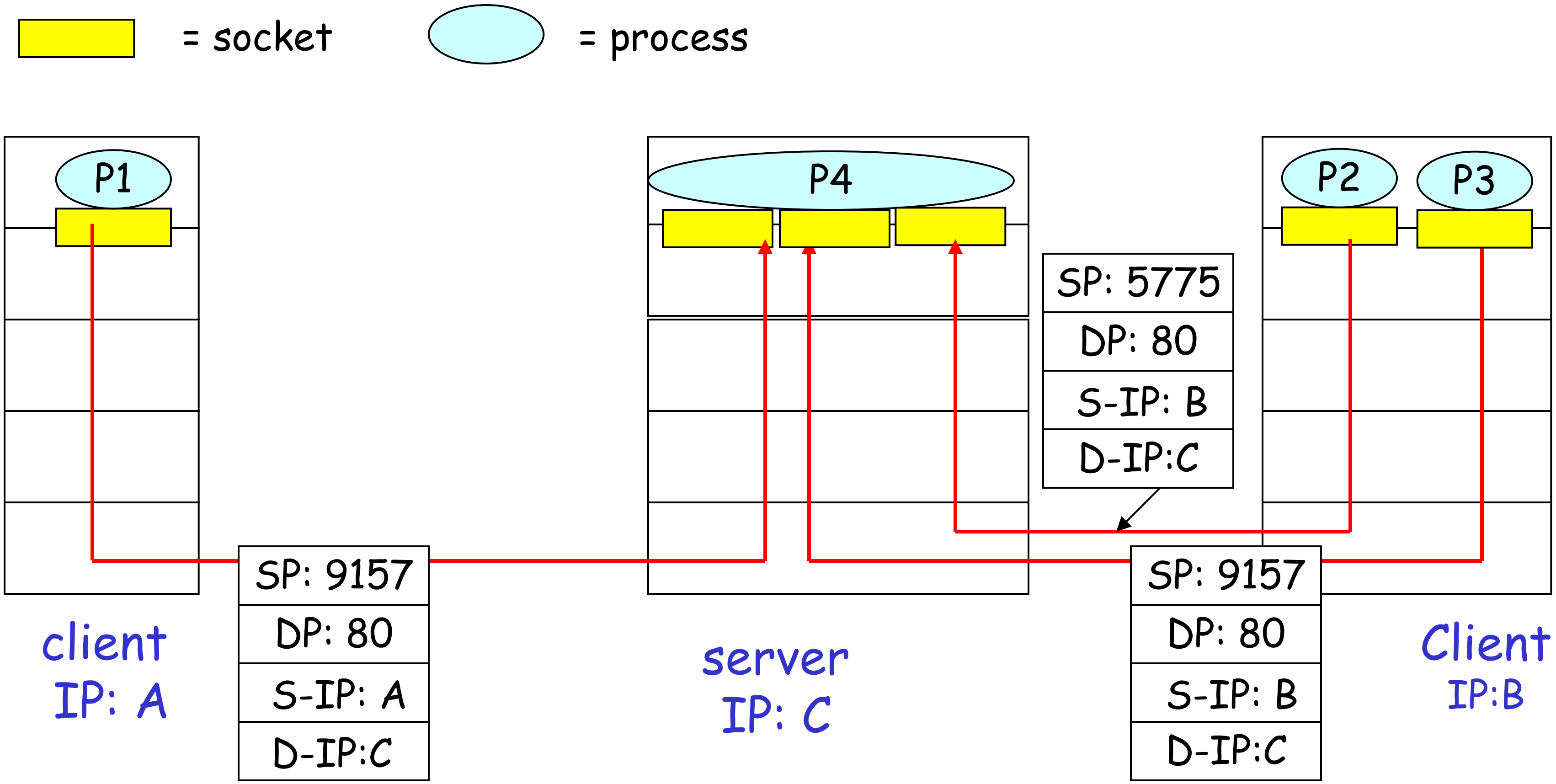
- TCP socket identified by 4-tuple:
  - source IP address
  - source port number
  - dest IP address
  - dest port number
- recv host uses all four values to direct segment to appropriate socket

- Server host may support many simultaneous TCP sockets:
  - each socket identified by its own 4-tuple
- Web servers have different sockets for each connecting client
  - non-persistent HTTP will have different socket for each request

# Connection-oriented demux (cont)



# Connection-oriented demux: Threaded Web Server



# (2) Abstraction of services

# IP Protocol Stack: Key Abstractions


- Transport layer is where we "pay the piper"
  - Provide applications with good abstractions
  - Without support or feedback from the network

# End-to-end Protocols
# (UDP, TCP)

# End-to-end Protocols

- Common properties that a transport protocol can be expected to provide
  - Guarantees message delivery
  - Delivers messages in the same order they were sent
  - Delivers at most one copy of each message
  - Supports arbitrarily large messages
  - Supports synchronization between the sender and the receiver
  - Allows the receiver to apply flow control to the sender
  - Supports multiple application processes (Mux/DeMux)

# End-to-end Protocols

- Typical limitations of the network on which transport protocol will operate
  - Drop messages/Packet loss
  - Reorder messages/Out of order delivery
  - Deliver duplicate copies of a given message
  - Limit messages to some finite size
  - Deliver messages after an arbitrarily long delay

# User Datagram Protocol (UDP)

# UDP: User Datagram Protocol [RFC 768]

- Bare minimum Internet transport protocol
- "best effort" service, UDP segments may be:
  - lost
  - delivered out of order to app
- *connectionless:*
  - no handshaking between UDP sender, receiver
  - each UDP segment handled independently of others

# User Datagram Protocol (UDP)

- Datagram messaging service
  - Demultiplexing: port numbers
  - Detecting corruption: checksum
- Lightweight communication between processes
  - Send and receive messages
  - Avoid overhead of any ordered, reliable delivery

| | |
|---|---|
| SRC port | DST port |
| length | checksum |
| DATA | |

# UDP checksum

Goal: detect "errors" (e.g., flipped bits) in transmitted segment

Sender:

- treat segment contents as sequence of 16-bit integers
- checksum: addition (1's complement sum) of segment contents
- sender puts checksum value into UDP checksum field

Receiver:

- compute checksum of received segment
- check if computed checksum equals checksum field value:
  - NO - error detected
  - YES - no error detected.

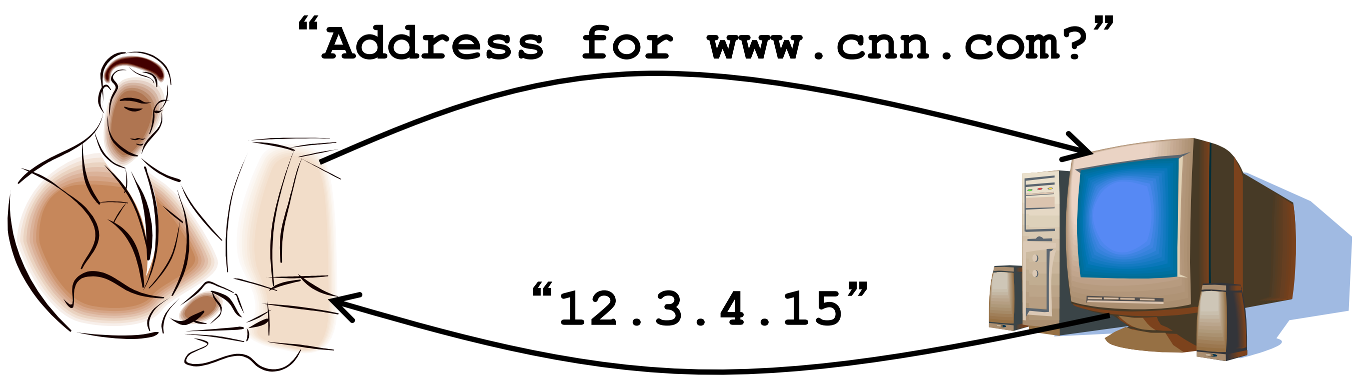
Is UDP service == IP service ??

# Advantages of UDP

- Fine-grain control
  - UDP sends as soon as the application writes
- No connection set-up delay
  - UDP sends without establishing a connection
- No connection state
  - No buffers, parameters, sequence #s, etc.
- Small header overhead
  - UDP header is only eight-bytes long

# Popular Applications That Use UDP

- Multimedia streaming
  - Retransmitting packets is not always worthwhile
  - E.g., phone calls, video conferencing, gaming, IPTV
- Simple query-response protocols
  - Overhead of connection establishment is overkill
  - E.g., Domain Name System (DNS), DHCP, etc.



How to introduce reliability in transport?

What are the expectations from a reliable packet delivery system?

- Guaranteed delivery
- Ordered delivery
- At-most one copy of the message
- (any or all of the above)

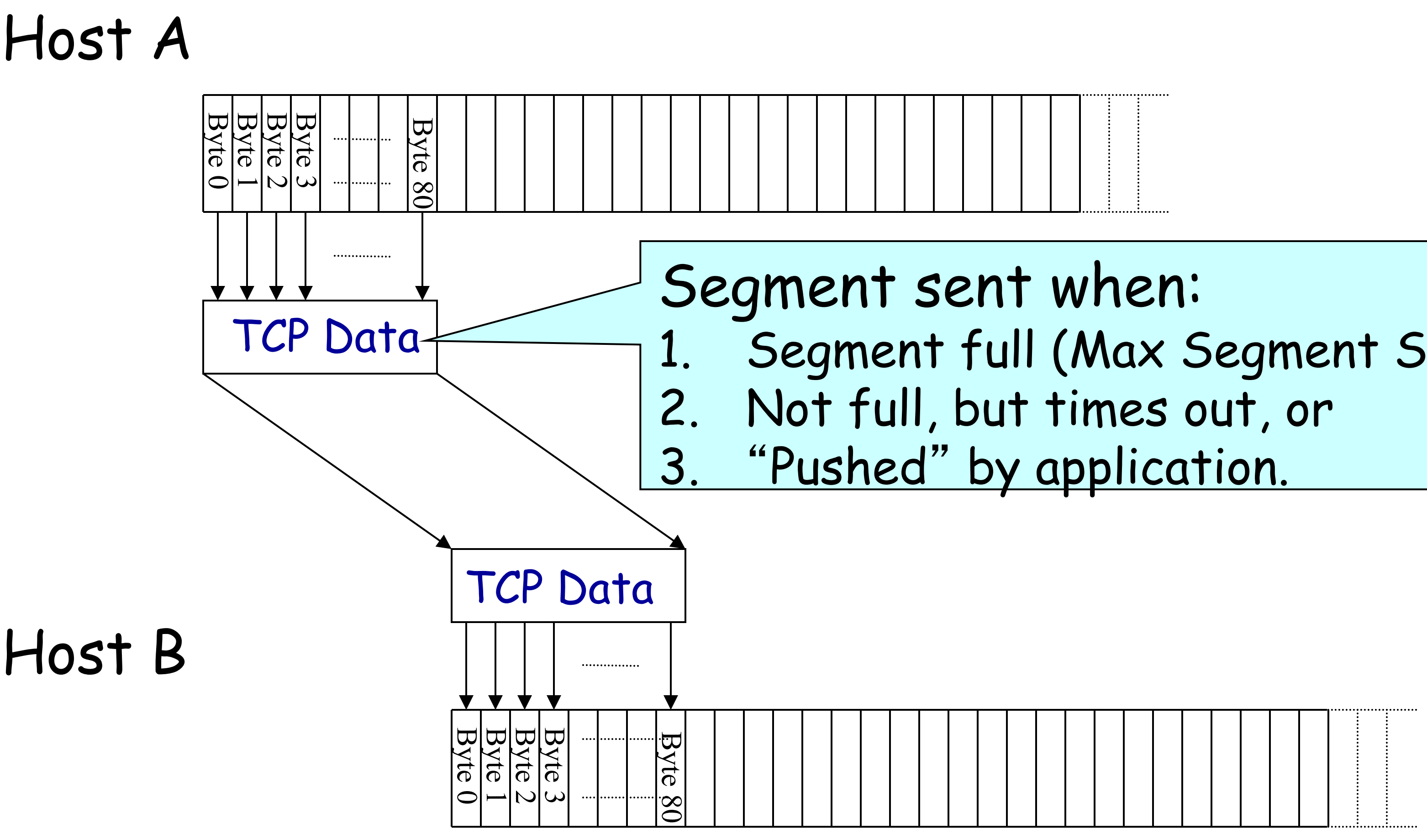
What are the features of the underlying 'best effort' network?

- Packet loss
- Out of order delivery
- Multiple copies
- Delay variations

- A non-zero probability of packet delivery

# Transmission Control Protocol (TCP)

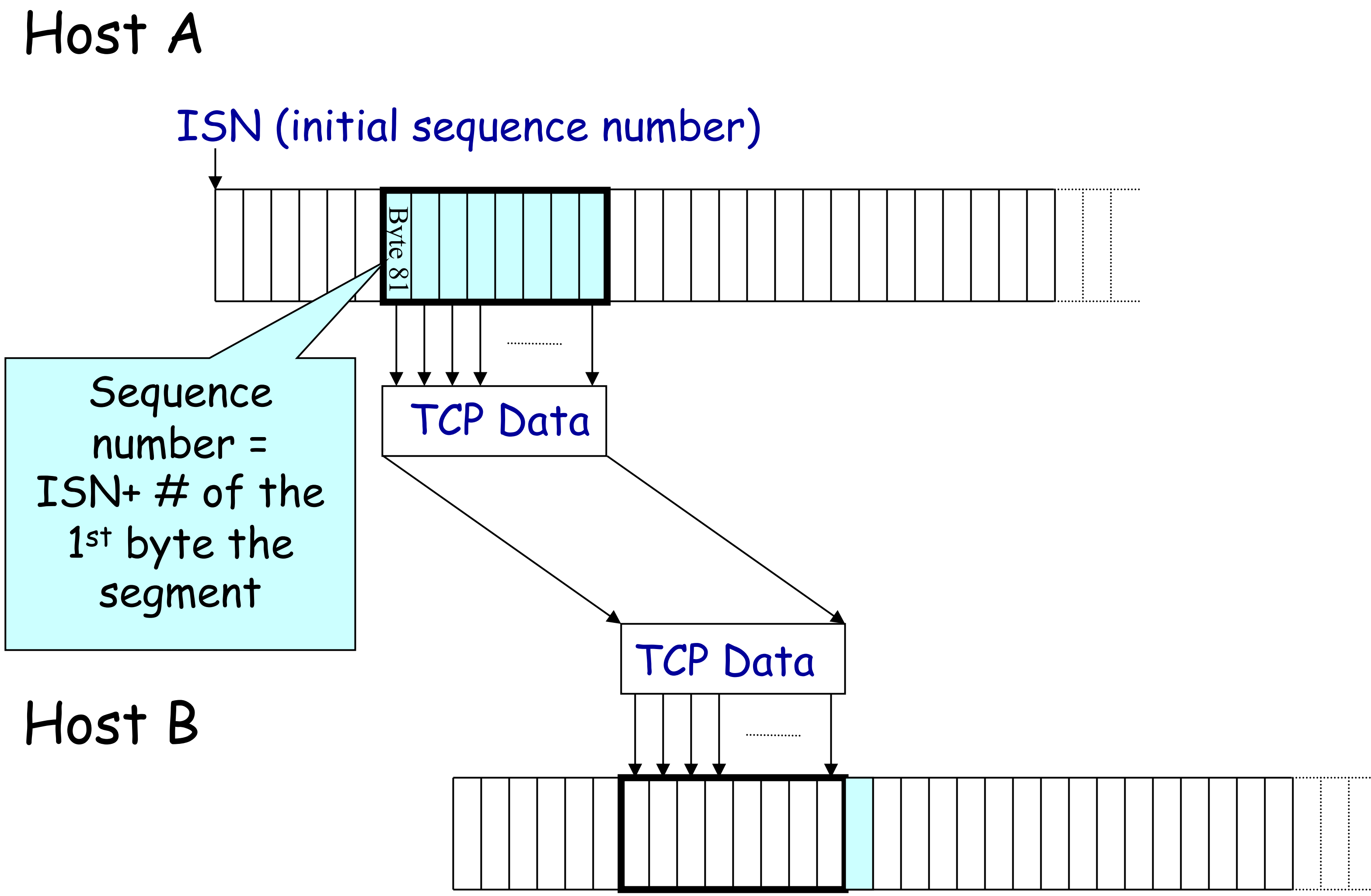
# TCP "Segments"



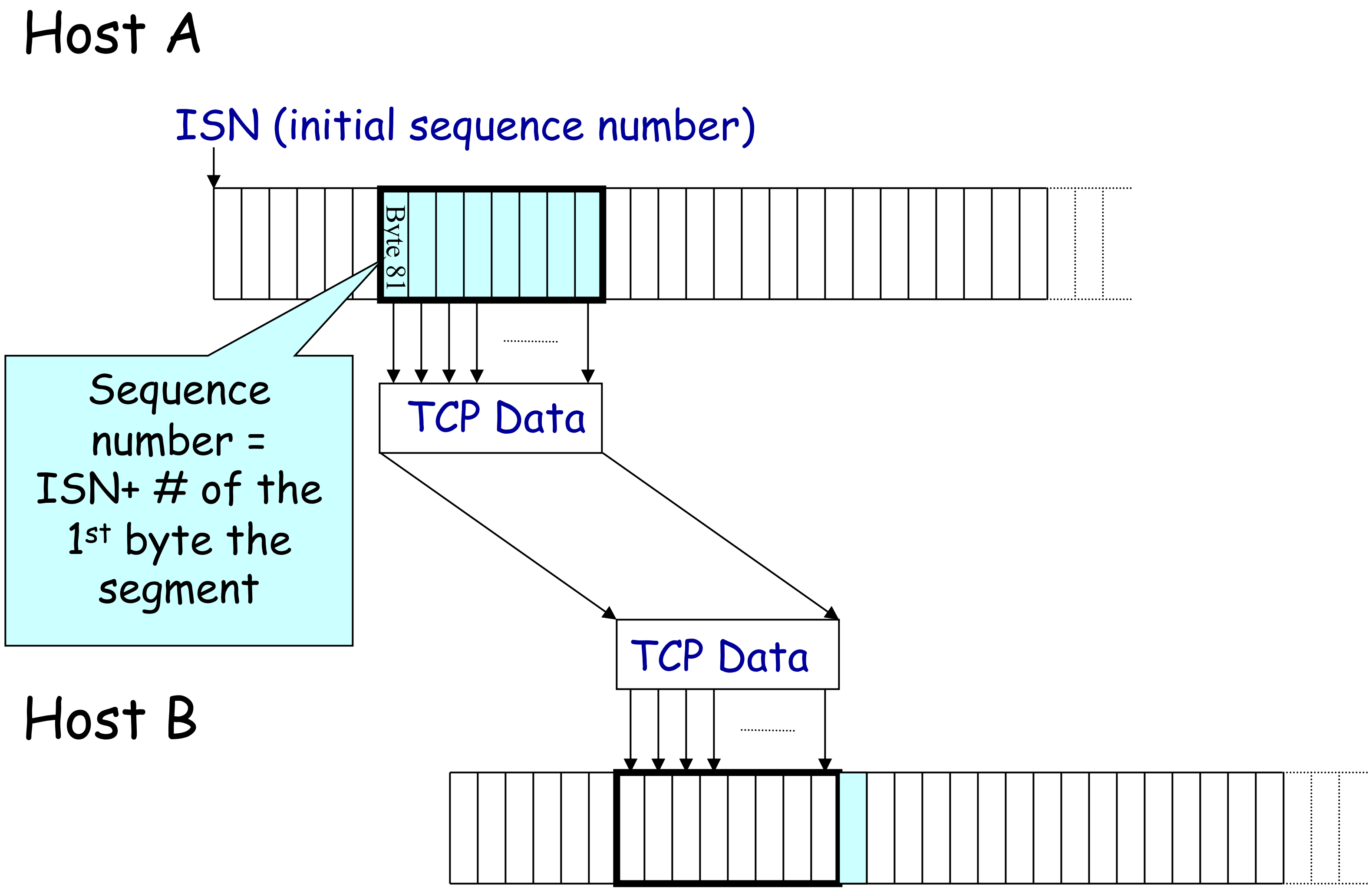

Segment sent when:

1. Segment full (Max Segment Size),
2. Not full, but times out, or
3. "Pushed" by application.

# Sequence Number

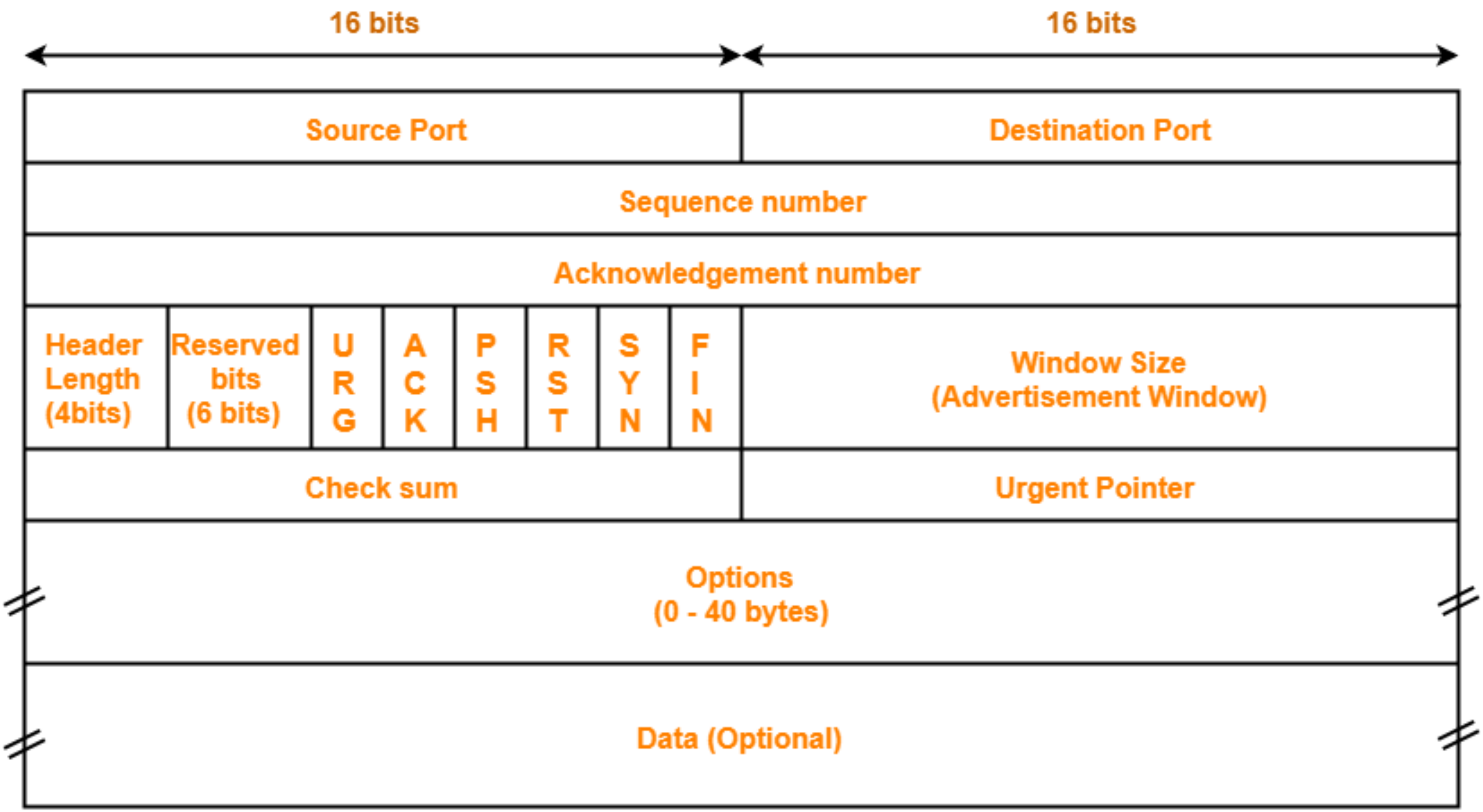


# Sequence Number



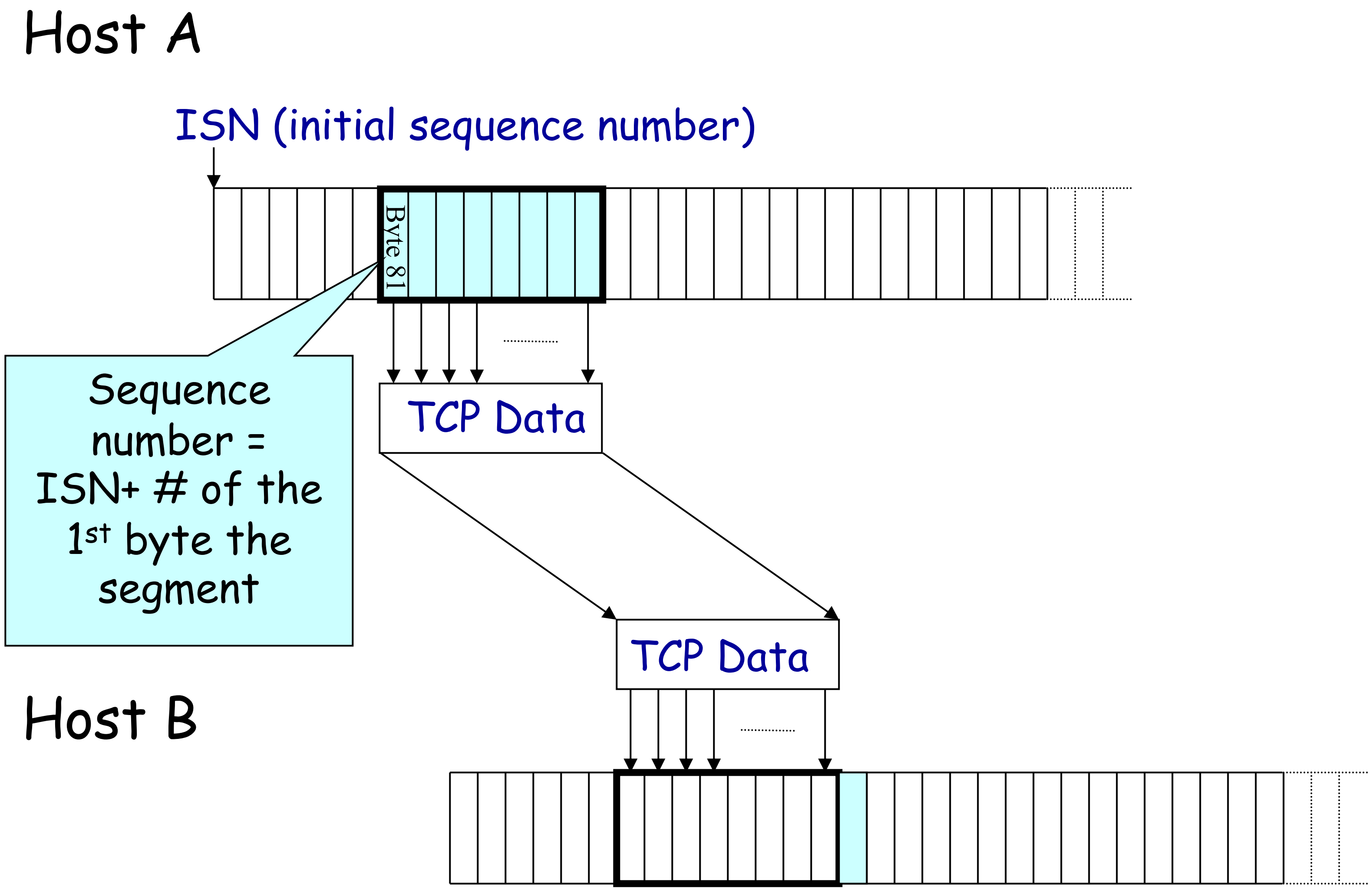

**How are acknowledgements identified?**

# TCP Header



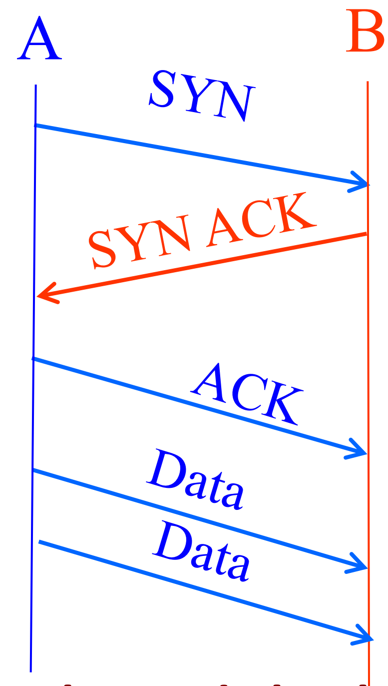

TCP Header

# Acknowledgement Number



# Starting and Ending a Connection: TCP Handshakes (Three-Way Handshake)

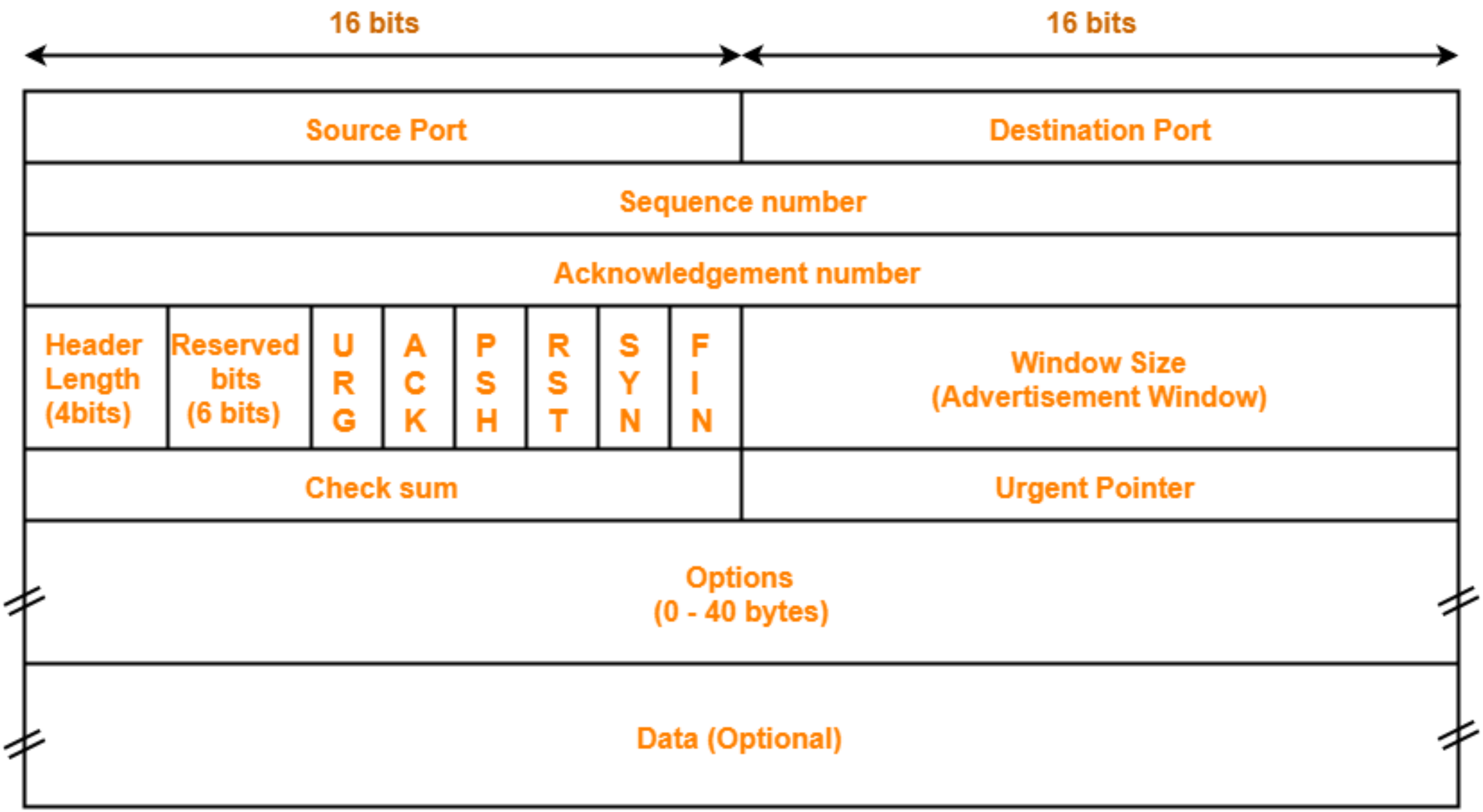
# Establishing a TCP Connection



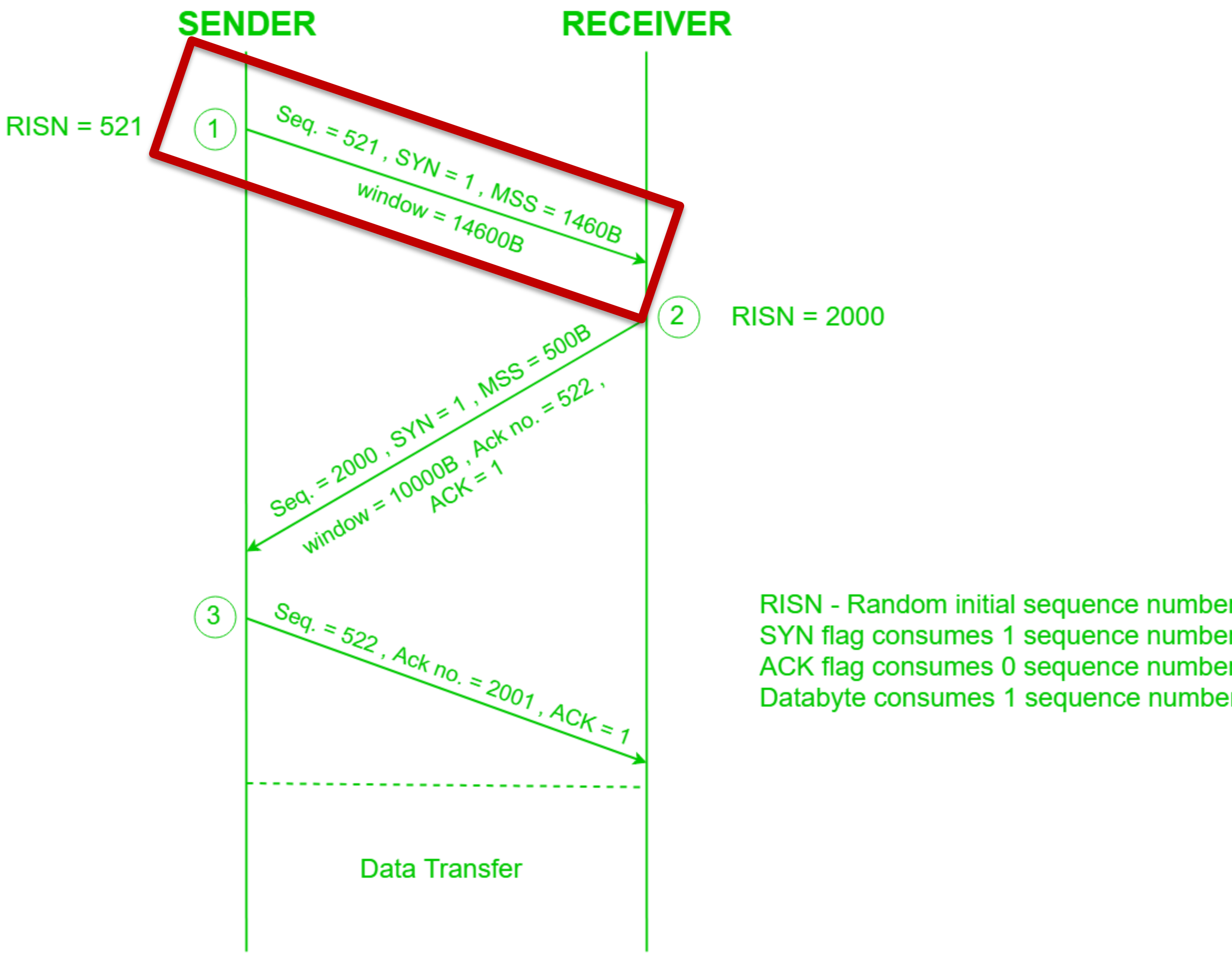

Each host tells its ISN to the other host.

- Three-way handshake to establish connection
  – Host A sends a **SYN** (open) to the host B
  – Host B returns a SYN acknowledgment (**SYN ACK**)
  – Host A sends an **ACK** to acknowledge the SYN ACK

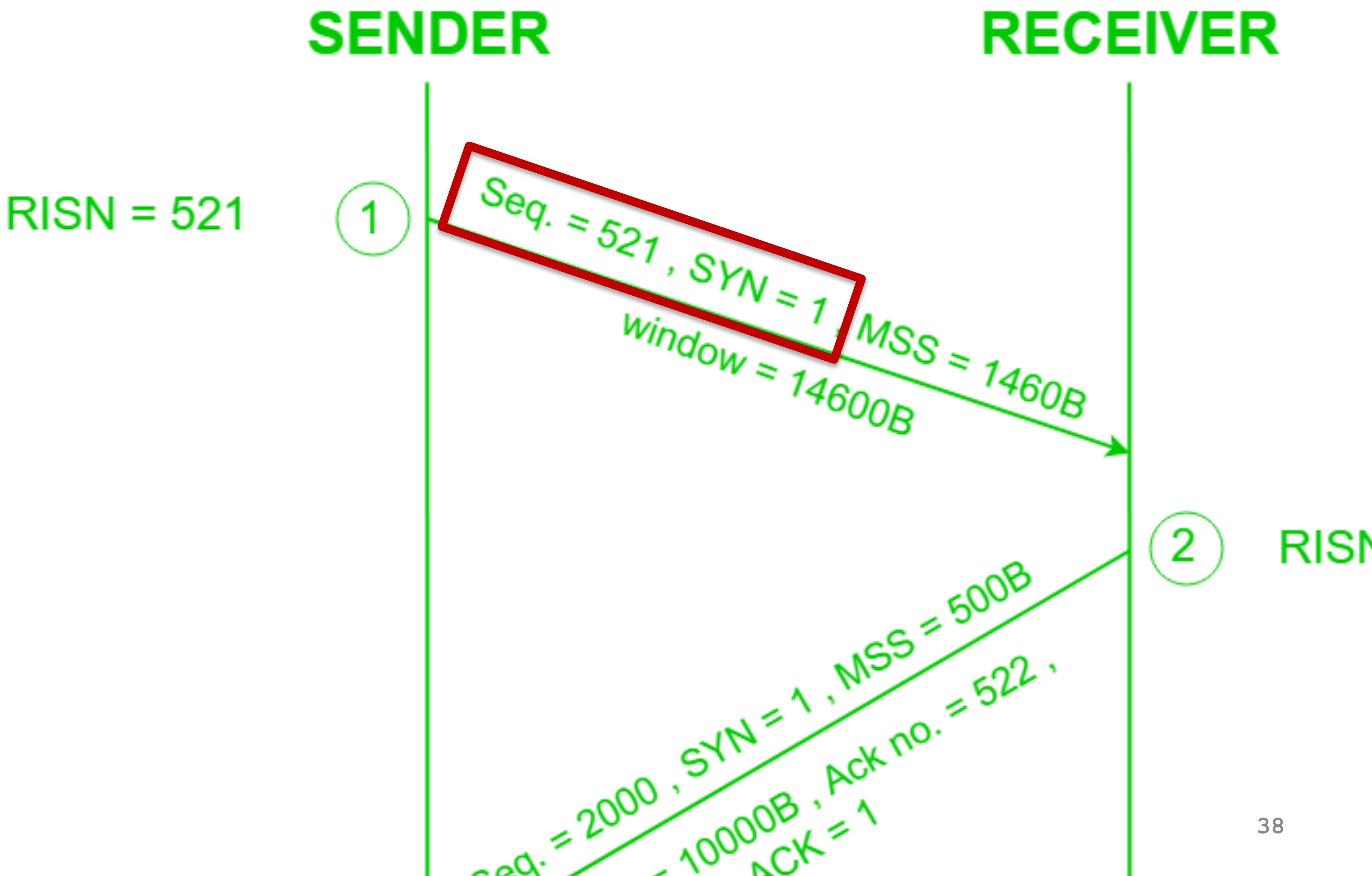
# TCP Header




TCP Header



SENDER
RECEIVER
RISN = 521
1
Seq. = 521 , SYN = 1 , MSS = 1460B
window = 14600B
2
RISN = 2000
Seq. = 2000 , SYN = 1 , MSS = 500B
window = 10000B , Ack no. = 522 ,
ACK = 1
3
Seq. = 522 , Ack no. = 2001 , ACK = 1
Data Transfer
RISN - Random initial sequence number
SYN flag consumes 1 sequence number
ACK flag consumes 0 sequence number
Databyte consumes 1 sequence number

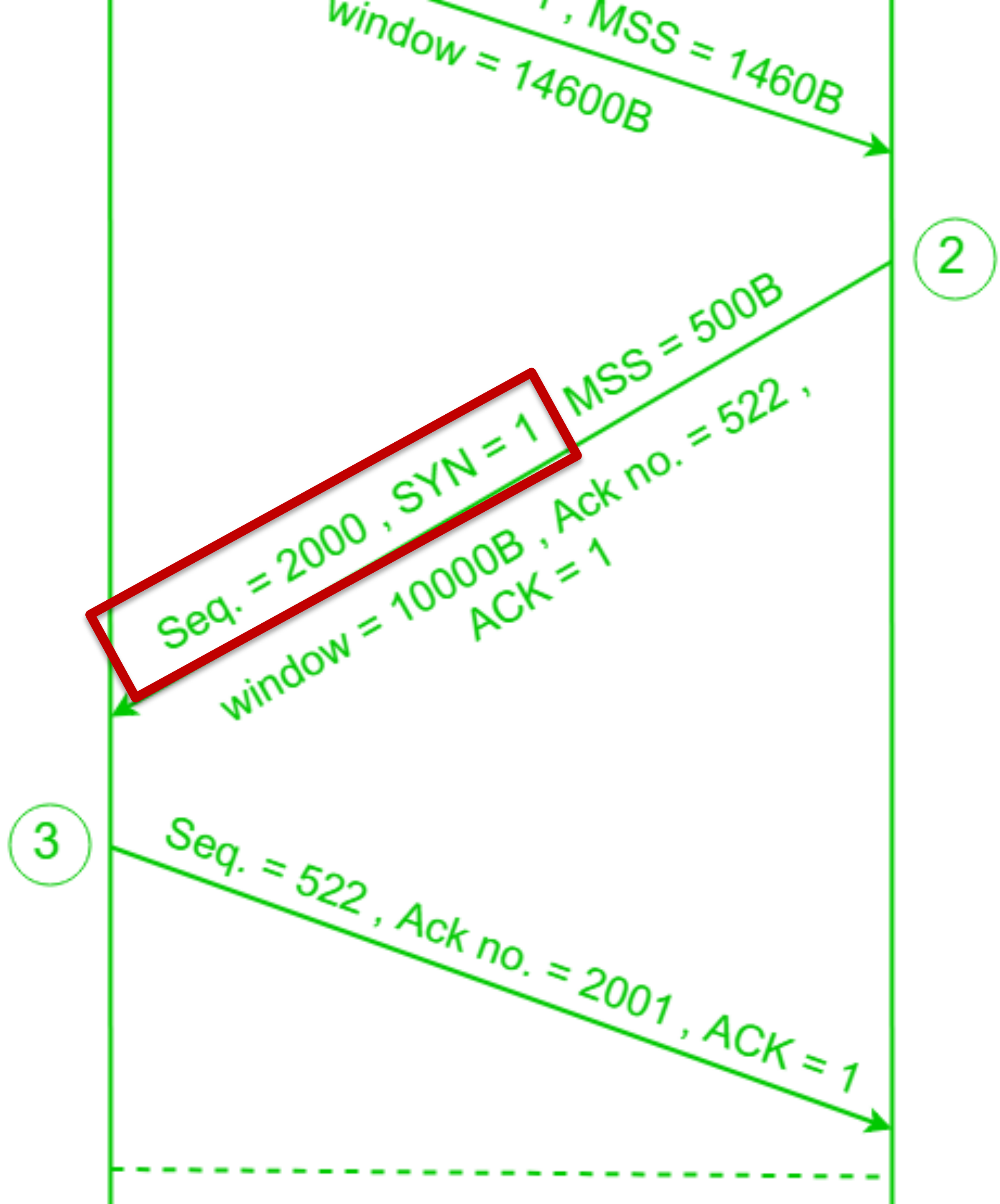

SENDER
RECEIVER
RISN = 521
1
Seq. = 521 , SYN = 1 , MSS = 1460B
window = 14600B
2
RISN
Seq. = 2000 , SYN = 1 , MSS = 500B
10000B , Ack no. = 522 ,
ACK = 1

# Step 1: A’s Initial SYN Packet

Flags: SYN
FIN
RST
PSH
URG
ACK

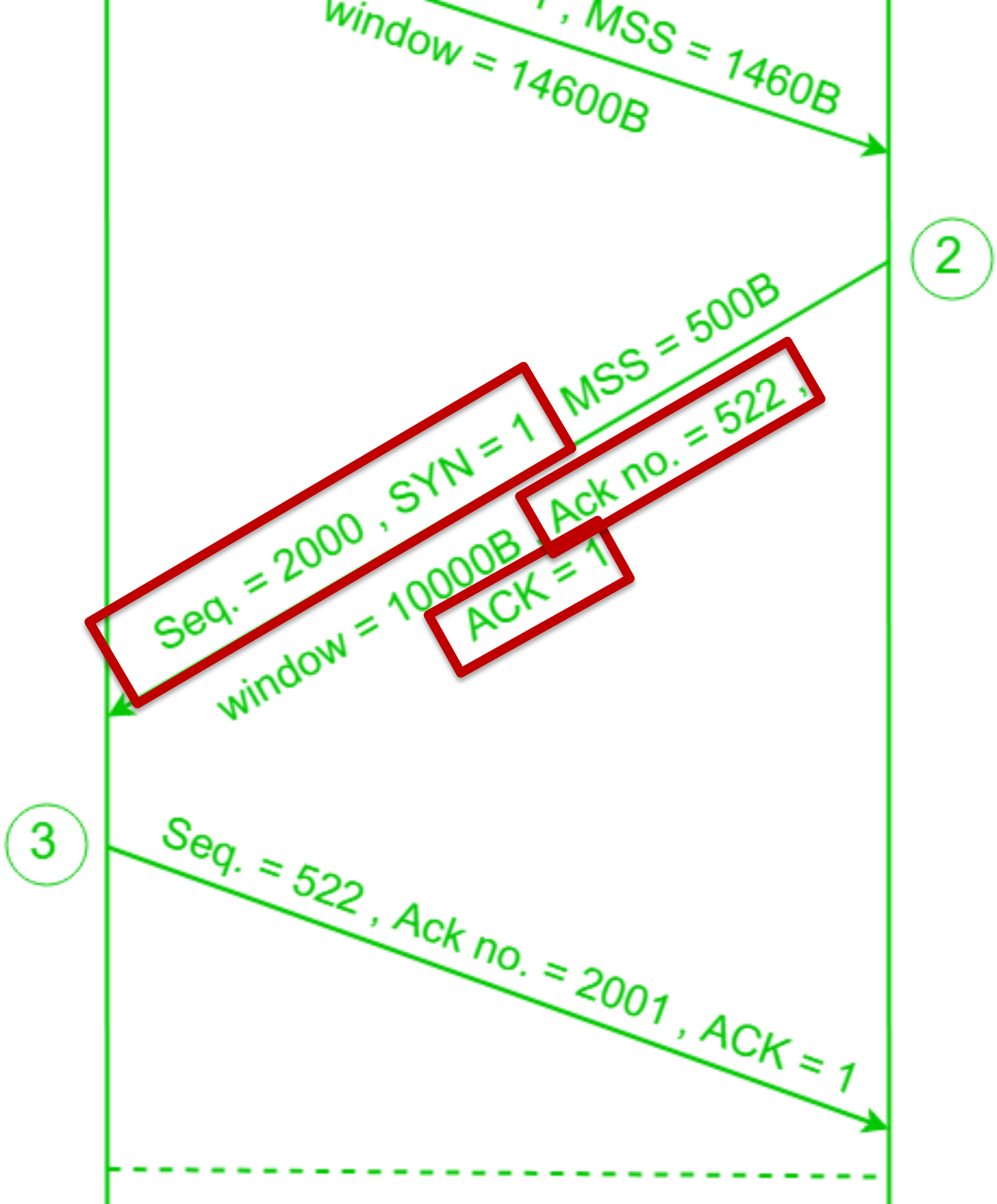
| | | | |
|---|---|---|---|
| A’s port | | B’s port | |
| A’s Initial Sequence Number | | | |
| Acknowledgment | | | |
| 20 | 0 | Flags | Advertised window |
| Checksum | | Urgent pointer | |
| Options (variable) | | | |

**A tells B it wants to open a connection…**




RISN = 20

RISN - R
SYN flag
ACK flag
Databyte




RISN = 20

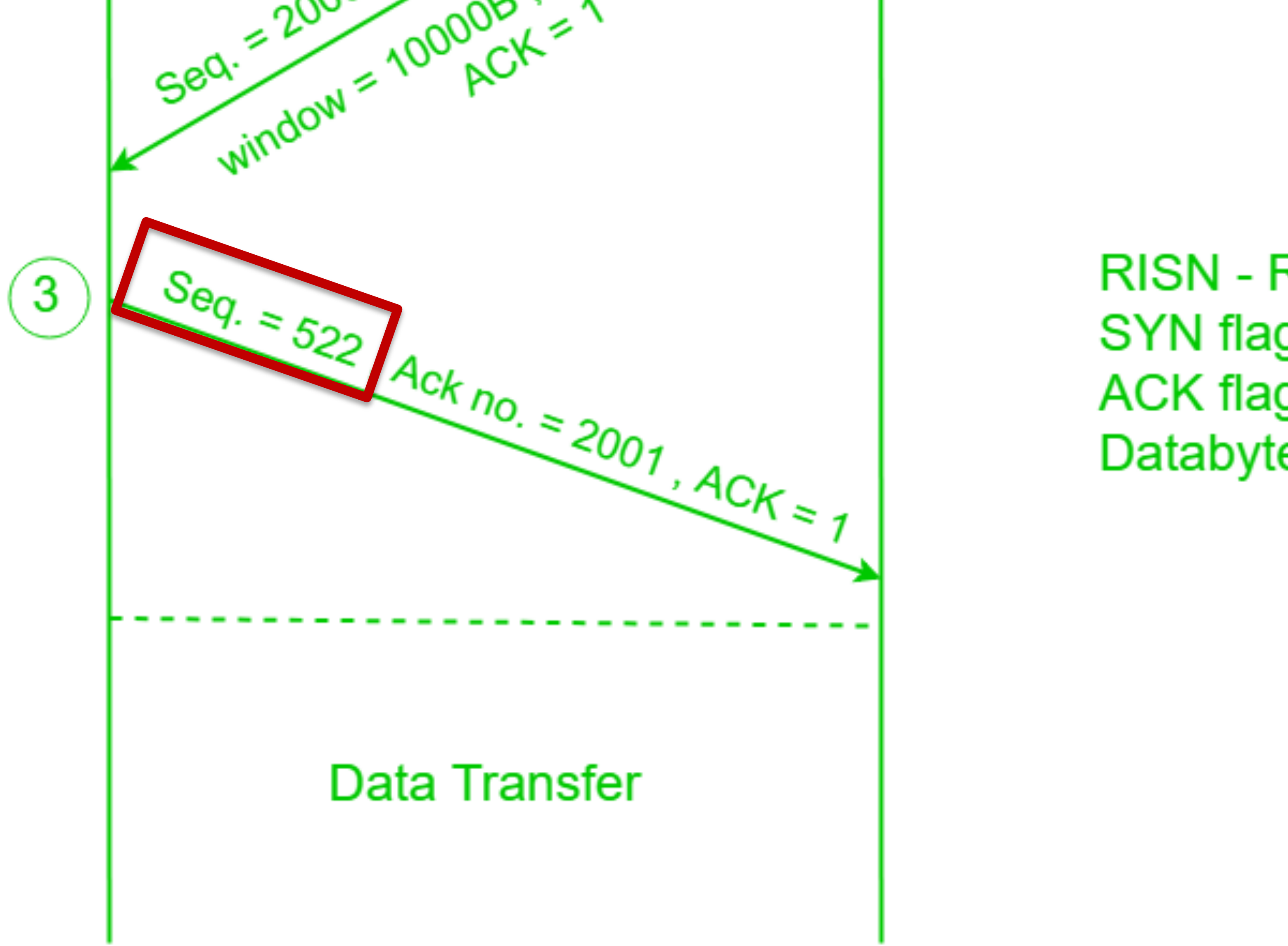
RISN - R
SYN flag
ACK flag
Databyte

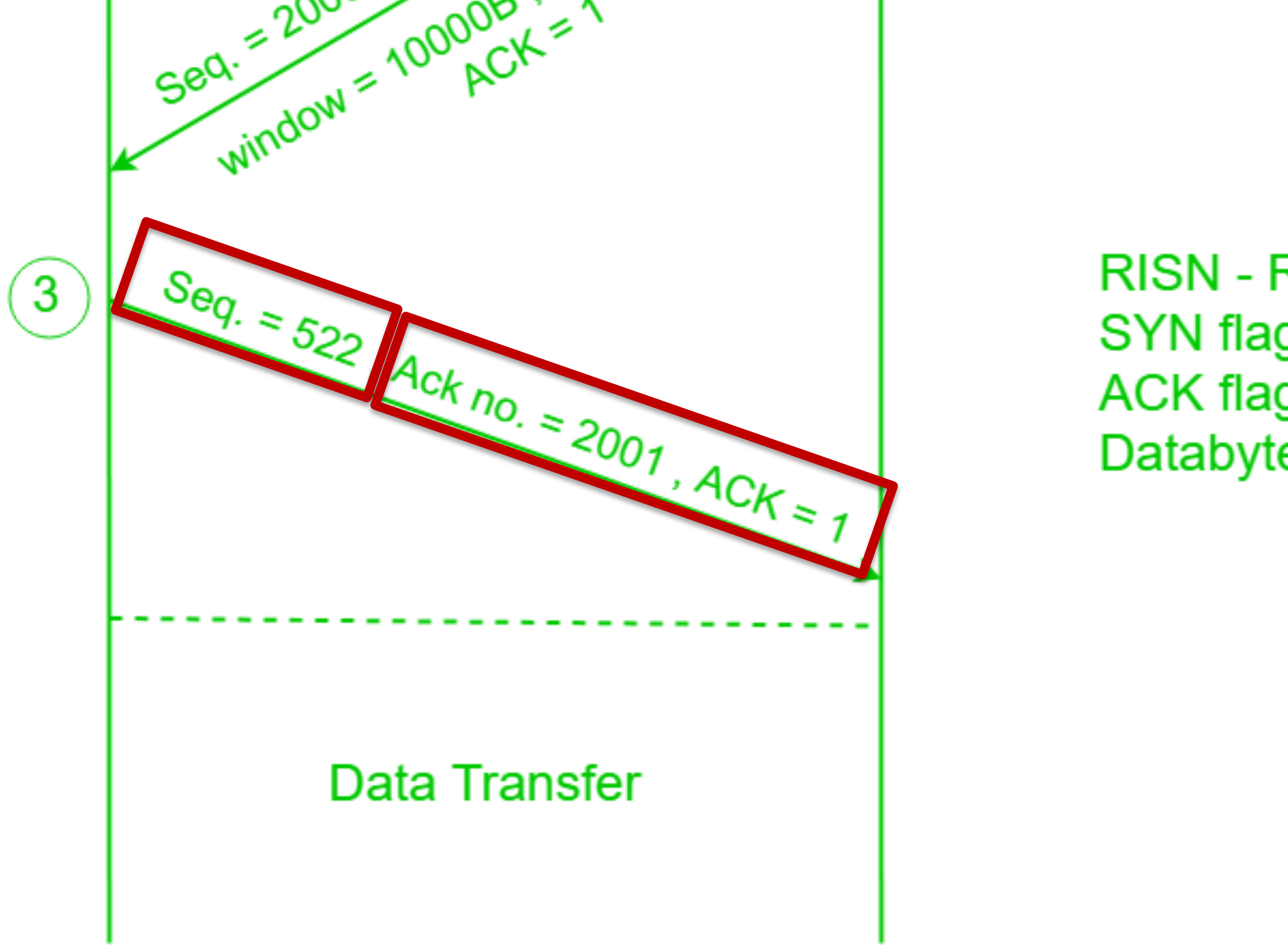
# Step 2: B's SYN-ACK Packet

Flags: SYN
FIN
RST
PSH
URG
ACK

| | | | |
|---|---|---|---|
| B's port | | A's port | |
| B's Initial Sequence Number | | | |
| A's ISN plus 1 | | | |
| 20 | 0 | Flags | Advertised window |
| Checksum | | Urgent pointer | |
| Options (variable) | | | |

**B tells A it accepts, and is ready to hear the next byte…**
**… upon receiving this packet, A can start sending data**


Seq. = 200
window = 10000B
ACK = 1
3
Seq. = 522
Ack no. = 2001 , ACK = 1
Data Transfer
RISN - R
SYN flag
ACK flag
Databyte


Seq. = 200
window = 10000B
ACK = 1
3
Seq. = 522
Ack no. = 2001 , ACK = 1
Data Transfer
RISN - R
SYN flag
ACK flag
Databyte

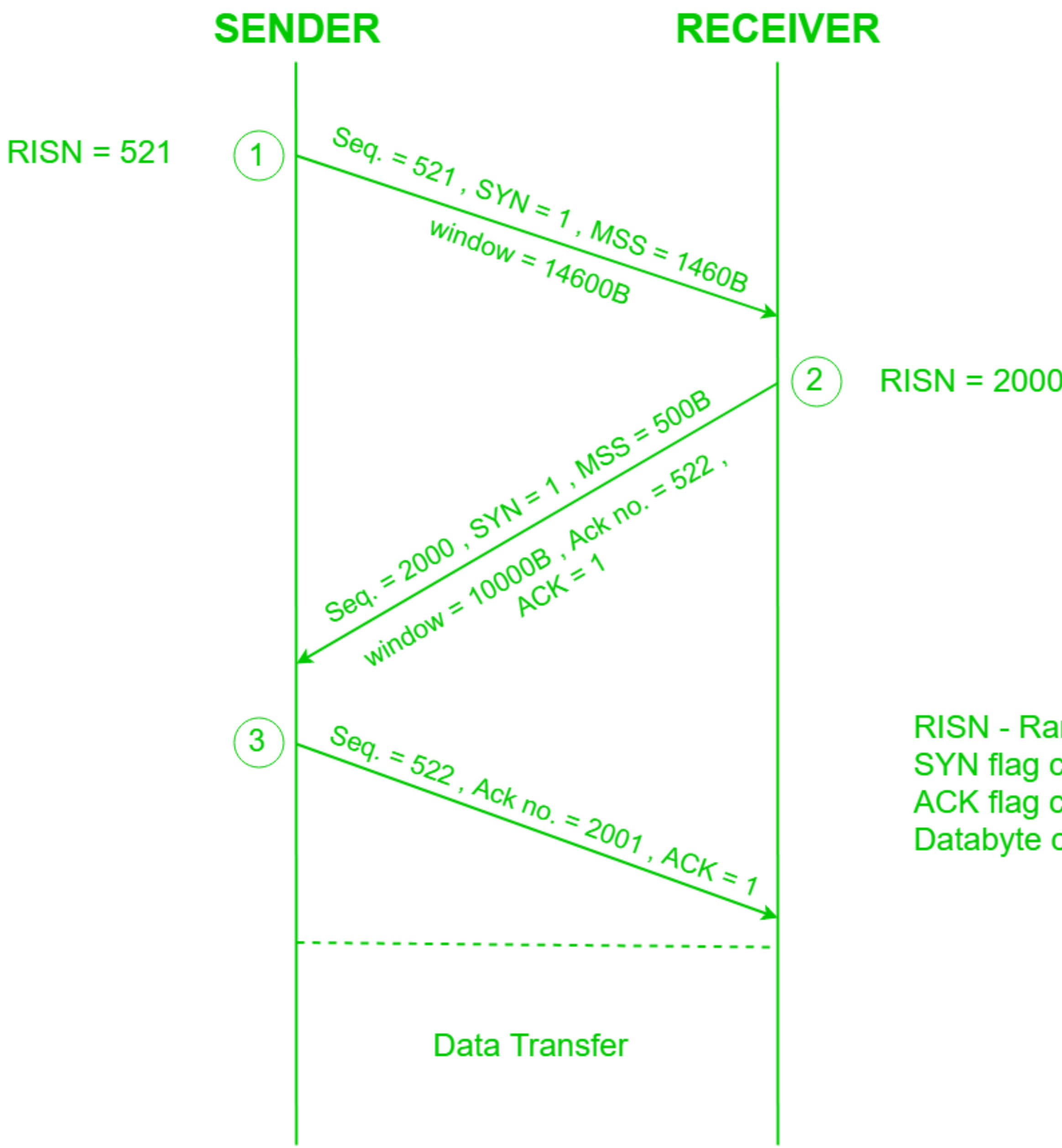
# Step 3: A’s ACK of the SYN-ACK

Flags:
SYN
FIN
RST
PSH
URG
ACK

<table>
<tr><td colspan="3">A’s port</td><td>B’s port</td></tr>
<tr><td colspan="4">Sequence number</td></tr>
<tr><td colspan="4">B’s ISN plus 1</td></tr>
<tr><td>20</td><td>0</td><td>Flags</td><td>Advertised window</td></tr>
<tr><td colspan="3">Checksum</td><td>Urgent pointer</td></tr>
<tr><td colspan="4">Options (variable)</td></tr>
</table>

**A tells B it is okay to start sending**

**… upon receiving this packet, B can start sending data**




RISN - Random initial sequence number
SYN flag consumes 1 sequence number
ACK flag consumes 0 sequence number
Databyte consumes 1 sequence number

# What is really happening?

- Exchange of ISNs
- (a) Alice --> Bob SYNchronize with my Initial Sequence Number of X
- (b) Alice <-- Bob I received your syn, I ACKnowledge that I am ready for [X+1]
- (c) Alice <-- Bob SYNchronize with my Initial Sequence Number of Y
- (d) Alice --> Bob I received your syn, I ACKnowledge that I am ready for [Y+1]

- Piggybacking information
- (a) Alice --> Bob : SYN
- (b) Alice <-- Bob : SYN+ACK
- (c) Alice --> Bob : ACK

- Can data piggyback on 3-way handshake packets?

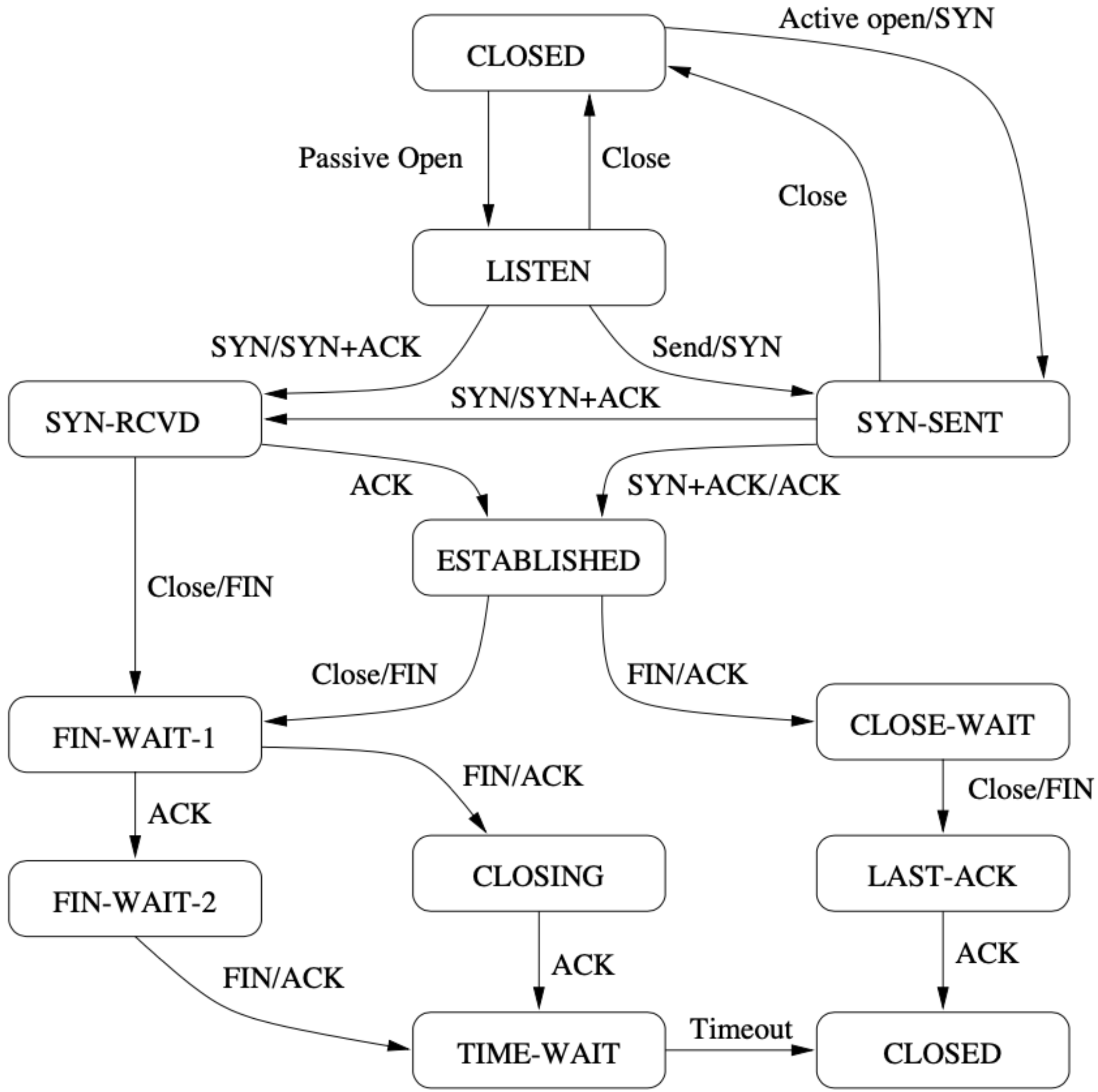



**Fig. 4.** TCP state-transition diagram

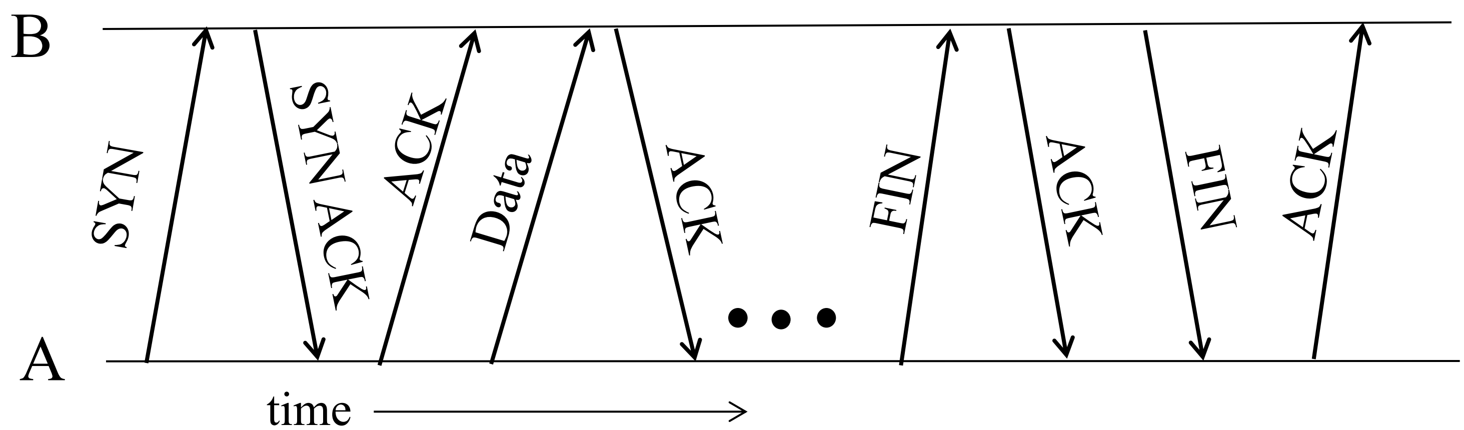


[Ref] Efficient Reachability Analysis of Hierarchical Reactive Machines

# What if the SYN Packet Gets Lost?

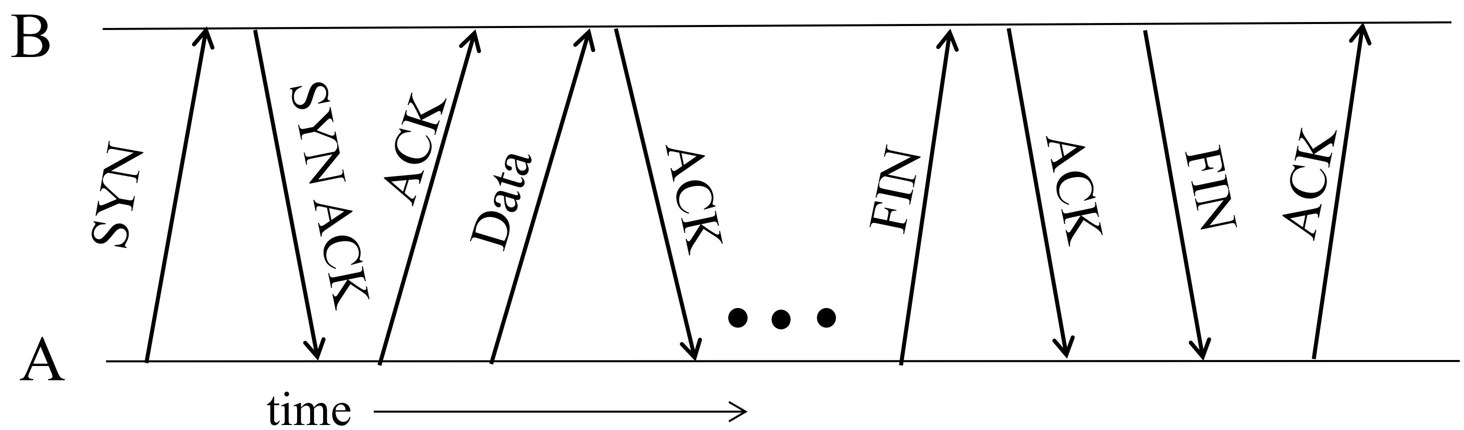
- Suppose the SYN packet gets lost
  - Packet is lost inside the network, or
  - Server rejects the packet (e.g., listen queue is full)
- Eventually, no SYN-ACK arrives
  - Sender sets a timer and wait for the SYN-ACK
  - … and retransmits the SYN if needed
- How should the TCP sender set the timer?
  - Sender has no idea how far away the receiver is
  - Some TCPs use a default of 3 or 6 seconds

# Tearing Down the Connection




- Closing (each end of) the connection
  - Finish (FIN) to close and receive remaining bytes
  - And other host sends a FIN ACK to acknowledge

# Tearing Down the Connection

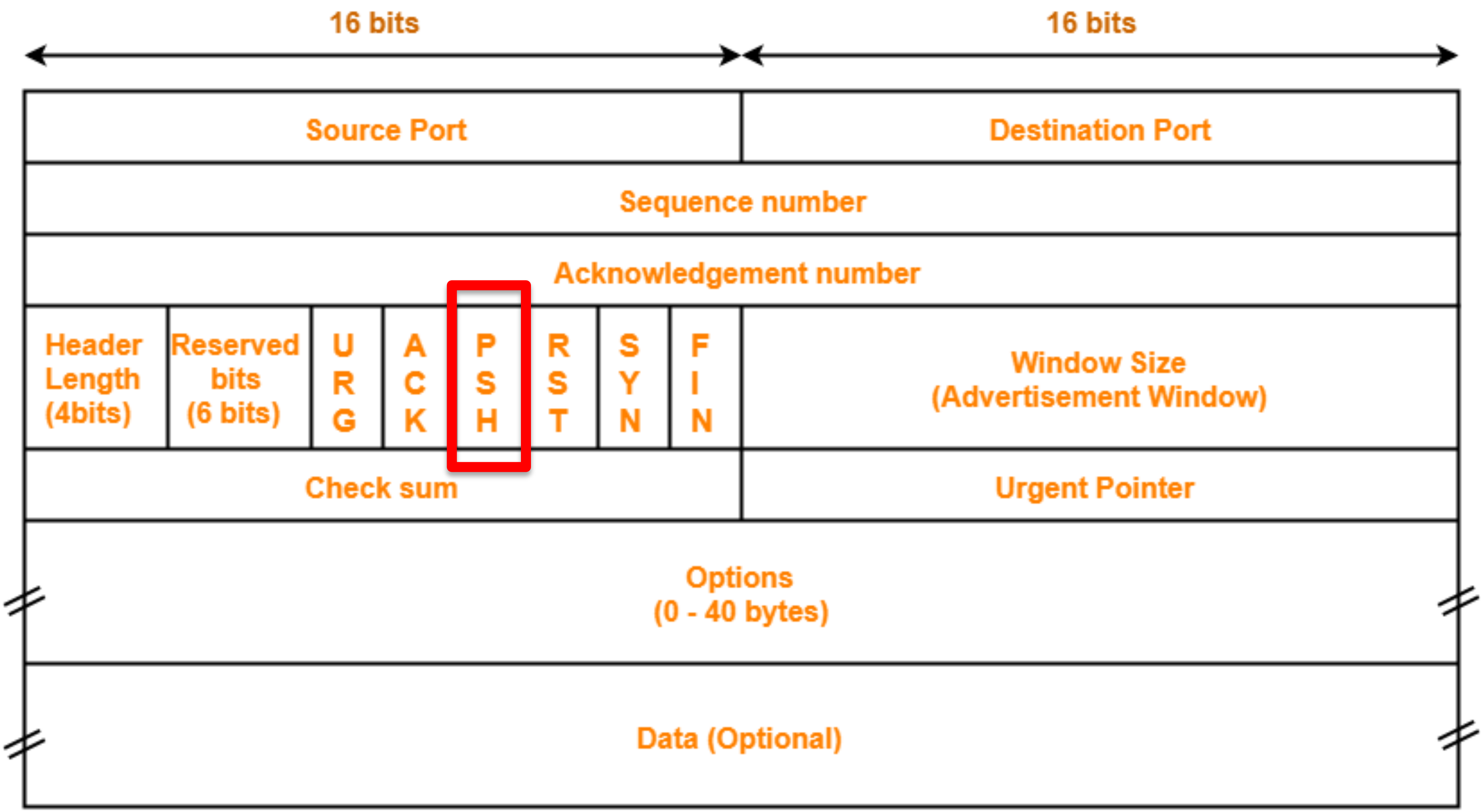



- Closing (each end of) the connection
  - Finish (FIN) to close and receive remaining bytes
  - And other host sends a FIN ACK to acknowledge
  - Reset (RST) to close and not receive remaining bytes

# Sending/Receiving the FIN Packet

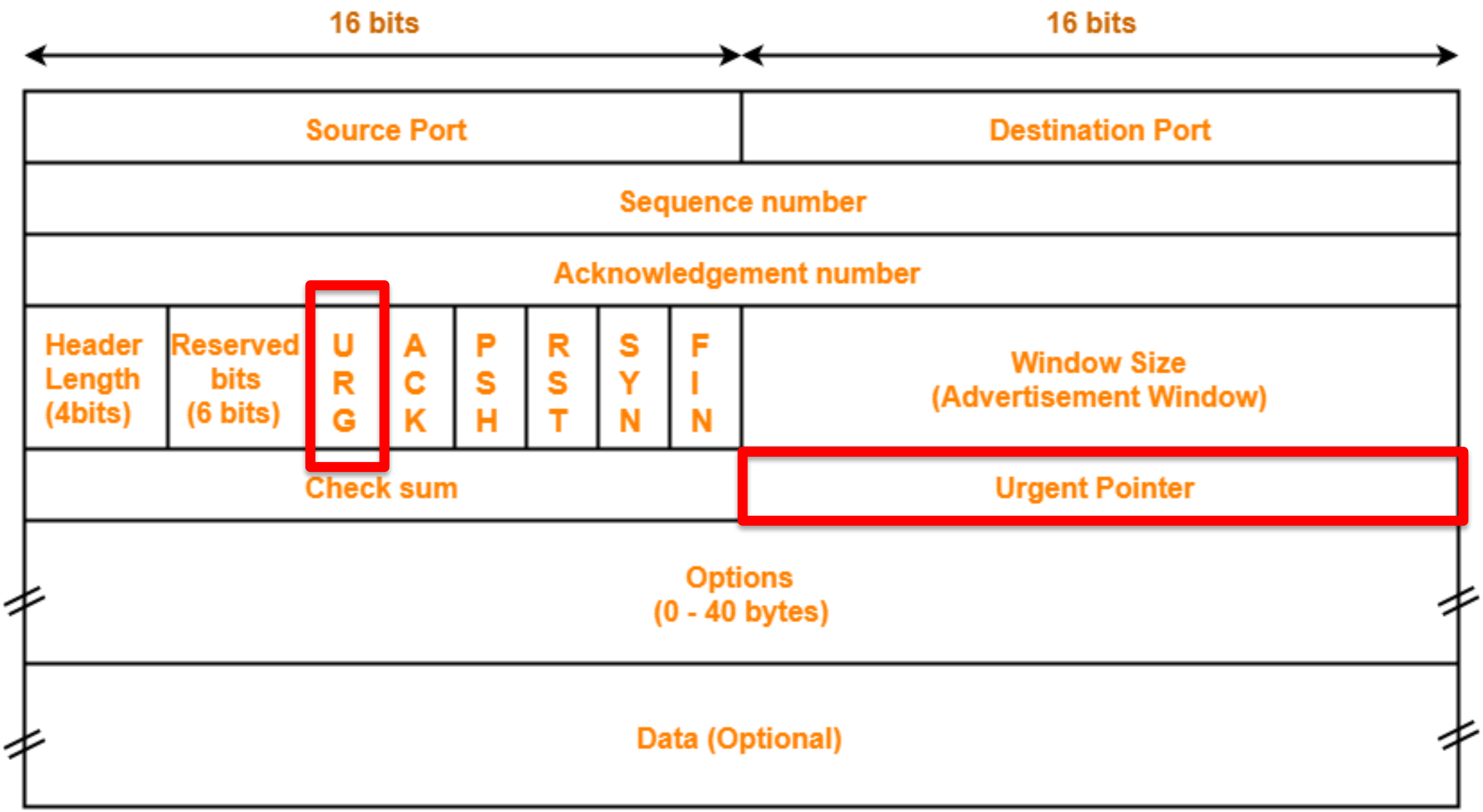
- Sending a FIN: close()
  - Process is done sending data via the socket
  - Process invokes "close()" to close the socket
  - Once TCP has sent all the outstanding bytes…
  - … then TCP sends a FIN
- Receiving a FIN: EOF
  - Process is reading data from the socket
  - Eventually, the attempt to read returns an EOF

# TCP Header: PSH and URG flags



TCP Header

# TCP Header: PSH and URG flags




TCP Header

# Reliable Delivery on a Lossy Channel With Bit Errors

# Challenges of Reliable Data Transfer

- Over a perfectly reliable channel
  - Easy: sender sends, and receiver receives
- Over a channel with bit errors
  - Receiver detects errors and requests retransmission
- Over a lossy channel with bit errors
  - Some data are missing, and others corrupted
  - Receiver cannot always detect loss
- Over a channel that may reorder packets
  - Receiver cannot distinguish loss from out-of-order