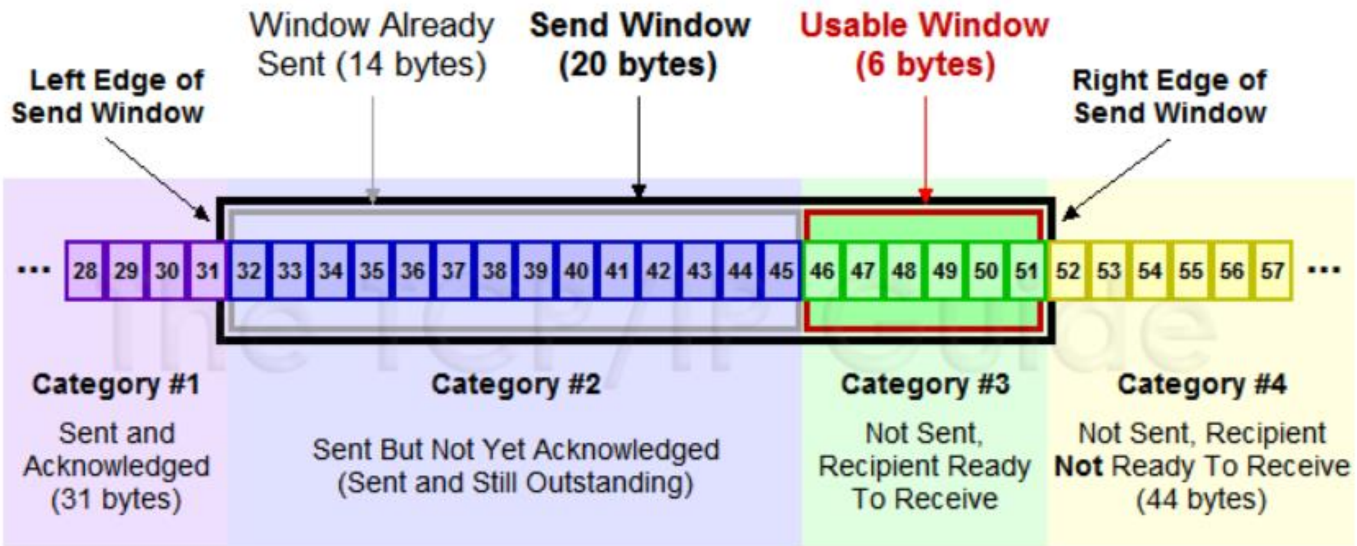
CS 305 Lab Tutorial
Lab 8 TCP Sliding Window & QUIC

Dept. Computer Science and Engineering
Southern University of Science and Technology

SUSTech
Southern University
of Science and Technology

Thanks to Wei WANG & HHQ ZHANG

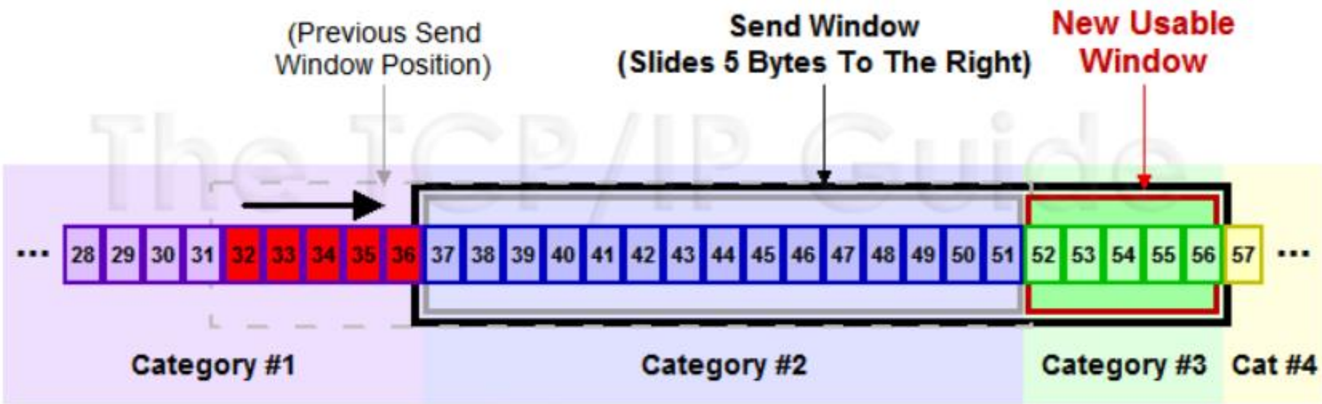
Part A.1 Sliding window system(1)



The ***send window*** is the key to the entire TCP sliding window system: it represents the maximum number of unacknowledged bytes a device is allowed to have outstanding at once.
The **usable window** is the number of bytes that the sender is still allowed to send at any point in time; it is equal to the size of the send window less the number of unacknowledged bytes already transmitted.

http://www.TCPipguide.com/free/t_TCPSlidingWindowAcknowledgmentSystemForDataTranspo-6.htm

Sliding window system(2)



When the sending device **receives new acknowledgment**, it will be able to transfer some of the bytes from Category #2 to Category #1, since they have now been acknowledged. When it does so, something interesting will happen. Since five bytes have been acknowledged, and the window size didn't change, the sender is allowed to send five more bytes. In effect, the window shifts, or ***slides***, over to the right in the timeline.

At the same time five bytes move from Category #2 to Category #1, five bytes move from Category #4 to Category #3, **creating a new usable window for subsequent transmission**.

SUSTech
Southern University
of Science and Technology

http://www.TCPipguide.com/free/t_TCPSlidingWindowAcknowledgmentSystemForDataTranspo-6.htm

ACK number, Sequence number, len

No.	Time	Source	Destination	Protocol	Info
94	8.574280	gaia.cs.umass.…	192.168.88.149	TCP	http(80) → 54861 [ACK] Seq=65701 Ack=333 Win=30336 Len=1460
95	8.576343	gaia.cs.umass.…	192.168.88.149	TCP	http(80) → 54861 [ACK] Seq=67161 Ack=333 Win=30336 Len=1460
96	8.576345	gaia.cs.umass.…	192.168.88.149	TCP	http(80) → 54861 [ACK] Seq=68621 Ack=333 Win=30336 Len=1460
97	8.576345	gaia.cs.umass.…	192.168.88.149	TCP	http(80) → 54861 [ACK] Seq=70081 Ack=333 Win=30336 Len=1460
98	8.576516	192.168.88.149	gaia.cs.umass.…	TCP	54861 → http(80) [ACK] Seq=333 Ack=71541 Win=65536 Len=0

```
Source Port: 54861 (54861)
Destination Port: http (80)
[Stream index: 0]
[TCP Segment Len: 0]
Sequence number: 333    (relative sequence number)
[Next sequence number: 333    (relative sequence number)]
Acknowledgment number: 71541    (relative ack number)
0101 .... = Header Length: 20 bytes (5)
Flags: 0x010 (ACK)
  000. .... .... = Reserved: Not set
  ...0 .... .... = Nonce: Not set
  .... 0... .... = Congestion Window Reduced (CWR): Not set
  .... .0.. .... = ECN-Echo: Not set
  .... ..0. .... = Urgent: Not set
  .... ...1 .... = Acknowledgment: Set
  .... .... 0... = Push: Not set
  .... .... .0.. = Reset: Not set
  .... .... ..0. = Syn: Not set
  .... .... ...0 = Fin: Not set
  [TCP Flags: ·······A····]
```

ack_num（71541）=
seq（70081）+len（1460）

Changes of window

While the size of usable window turn to be 0, it means the sender will not send any segment at this moment. Wireshark mark the segment with “[Tcp Window Full]”

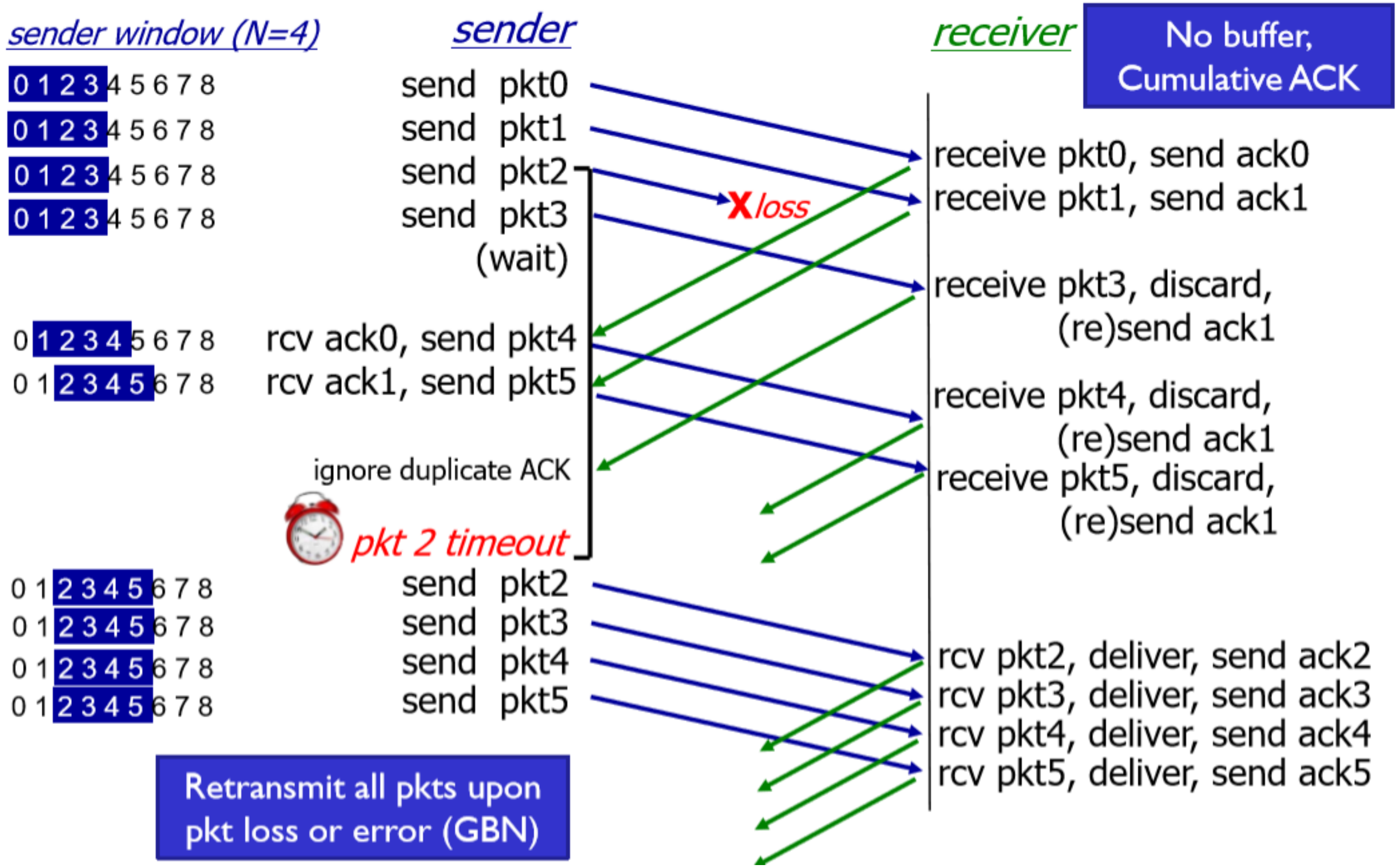
Seq（135781）+ len（1280） – ack（135781） = win（1280）

No.	Time	Source	Destination	Protocol	Info
307	19.117577	LAPTOP-RITC8…	gaia.cs.umass.…	TCP	54861 → http(80) [ACK] Seq=333 Ack=135781 Win=1280 Len=0
309	20.576025	gaia.cs.umas…	LAPTOP-RITC8FU…	TCP	[TCP Window Full] http(80) → 54861 [PSH, ACK] Seq=135781 Ack=333 Win=30336 Len=1280 [TCP…

While the recv window turn to be zero, sender will stop to send packet, it will send “[TCP Keep-Alive]” to keep the TCP connection, waiting for the changing of recv window.

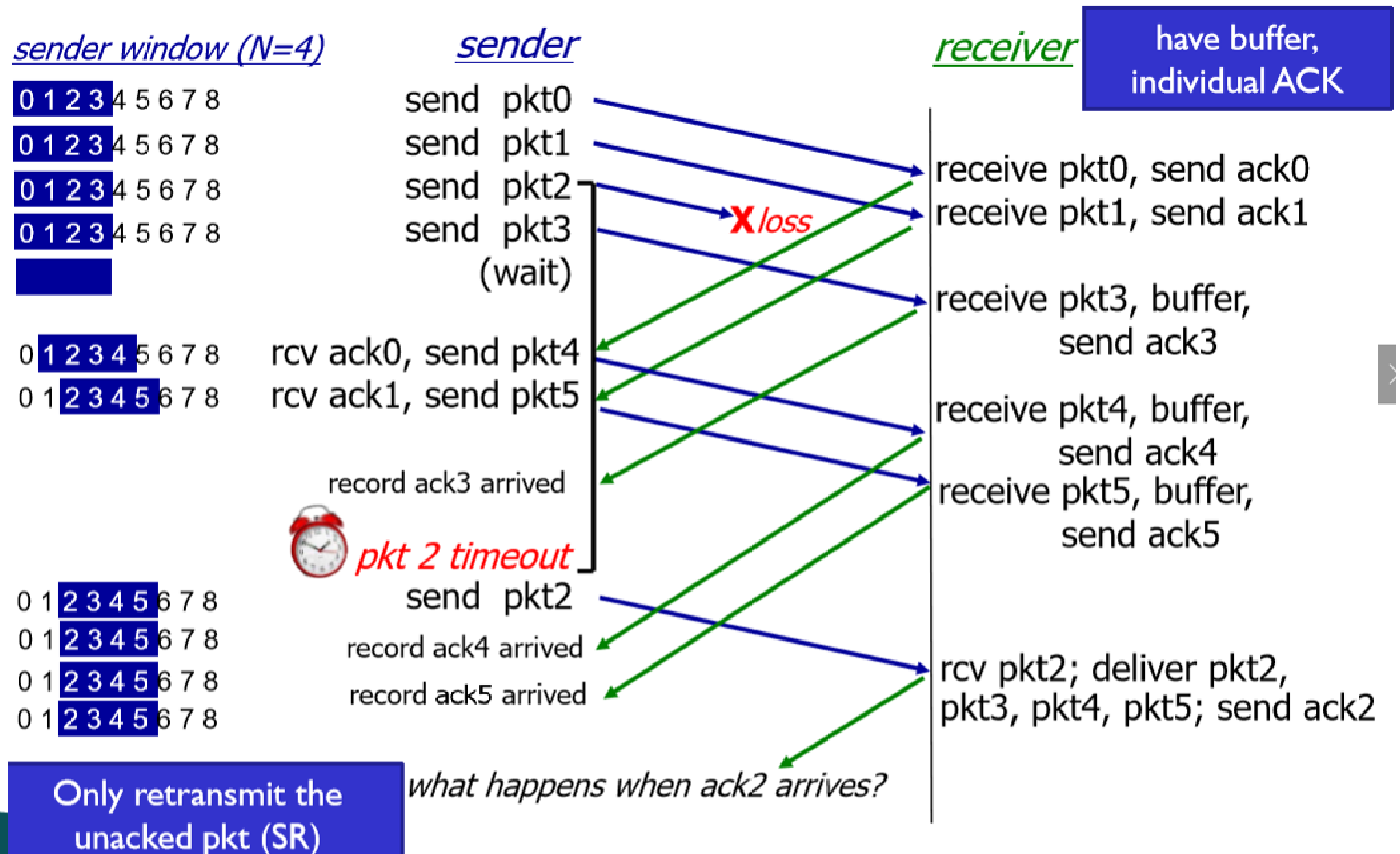
175	19.366403	192.168.88.149	gaia.cs.umass.edu	TCP	[TCP ZeroWindow] 54861 → http(80) [ACK] Seq=333 Ack=137061 Win=0 Len=0
176	20.862900	gaia.cs.umass.edu	192.168.88.149	TCP	[TCP Keep-Alive] http(80) → 54861 [ACK] Seq=137060 Ack=333 Win=30336 Len=0
177	20.862992	192.168.88.149	gaia.cs.umass.edu	TCP	[TCP ZeroWindow] 54861 → http(80) [ACK] Seq=333 Ack=137061 Win=0 Len=0
178	26.701220	gaia.cs.umass.edu	192.168.88.149	TCP	[TCP Keep-Alive] http(80) → 54861 [ACK] Seq=137060 Ack=333 Win=30336 Len=0
179	26.701357	192.168.88.149	gaia.cs.umass.edu	TCP	[TCP ZeroWindow] 54861 → http(80) [ACK] Seq=333 Ack=137061 Win=0 Len=0
180	31.323807	192.168.88.149	gaia.cs.umass.edu	TCP	[TCP Window Update] 54861 → http(80) [ACK] Seq=333 Ack=137061 Win=65536 Len=0

Part A.2 Retransmission : GBN



SUSTech
Southern University
of Science and Technology

SR



sender window (N=4)
sender
receiver
have buffer, individual ACK
0 1 2 3 4 5 6 7 8
0 1 2 3 4 5 6 7 8
0 1 2 3 4 5 6 7 8
0 1 2 3 4 5 6 7 8
send pkt0
send pkt1
send pkt2
send pkt3
(wait)
X loss
receive pkt0, send ack0
receive pkt1, send ack1
receive pkt3, buffer, send ack3
0 1 2 3 4 5 6 7 8
0 1 2 3 4 5 6 7 8
rcv ack0, send pkt4
rcv ack1, send pkt5
receive pkt4, buffer, send ack4
receive pkt5, buffer, send ack5
record ack3 arrived
pkt 2 timeout
0 1 2 3 4 5 6 7 8
0 1 2 3 4 5 6 7 8
0 1 2 3 4 5 6 7 8
0 1 2 3 4 5 6 7 8
send pkt2
record ack4 arrived
record ack5 arrived
rcv pkt2; deliver pkt2, pkt3, pkt4, pkt5; send ack2
what happens when ack2 arrives?
Only retransmit the unacked pkt (SR)

TCP SACK

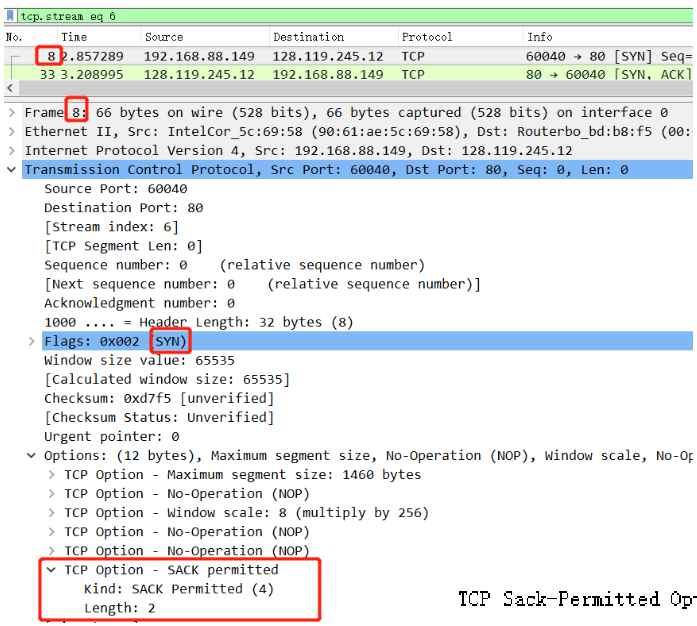
- A Selective Acknowledgment (**SACK**) mechanism, combined with a **selective repeat retransmission policy**, can help the sender retransmit only the missing data segments.
  - The receiving TCP **sends back SACK packets to the sender** informing the sender of data that has been received.
  - With selective acknowledgments, the data receiver can inform the sender about all segments that have arrived successfully, so the sender need **retransmit only the segments that have actually been lost.**

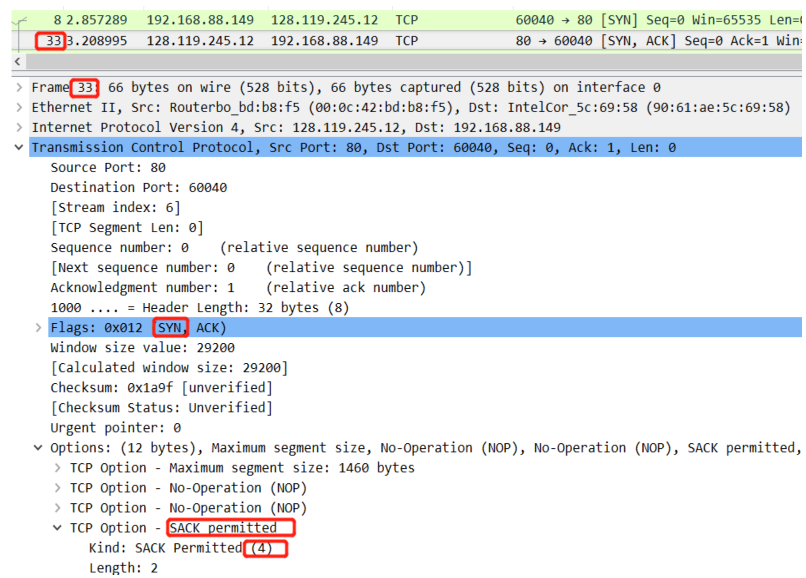
- The selective acknowledgment extension uses two TCP options:
  - **SACK-permitted** option
  - **SACK** option

https://tools.ietf.org/html/rfc2018

SACK-permitted option

- "**SACK-permitted**", which may **be sent in a SYN segment** to indicate that the SACK option can be used once the connection is established.






```
TCP Sack-Permitted Option:

Kind: 4

+---------+---------+
| Kind=4  | Length=2|
+---------+---------+
```

Wireshark tips: TCP.option_kind==4

SUSTech
Southern University
of Science and Technology

SACK option(1)

The **SACK option** is to be **sent by a data receiver to inform the data sender of non-contiguous blocks of data that have been received and queued**.

The data receiver awaits the receipt of data (perhaps by means of retransmissions) to fill the gaps in sequence space between received blocks.

When missing segments are received, **the data receiver acknowledges the data normally** by advancing the left window edge in the Acknowledgement Number Field of the TCP header.

The SACK option does not change the meaning of the Acknowledgement Number field.

This option contains a list of **some of the blocks of contiguous sequence space occupied by data that has been received and queued within the window.**

Each contiguous block of data queued at the data receiver is defined in the SACK option by **two 32-bit unsigned integers** in network byte order:

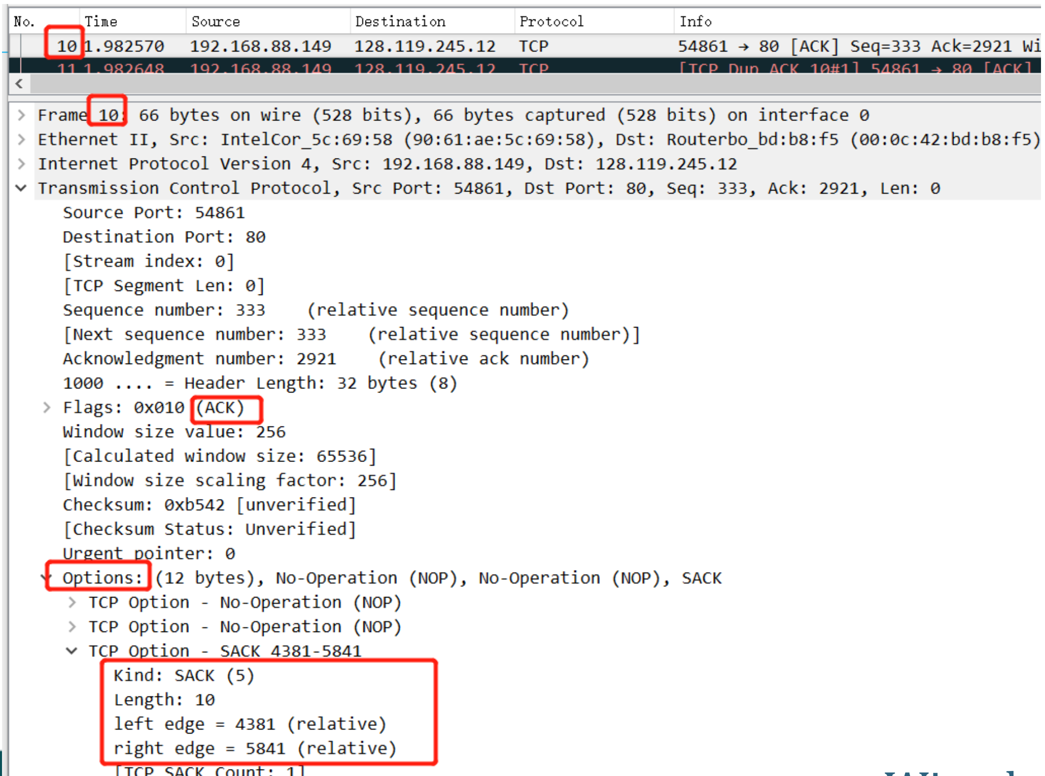
- **Left Edge** of Block This is the first sequence number of this block.
- **Right Edge** of Block This is the sequence number immediately following the last sequence number of this block.

Each block represents received bytes of data that are **contiguous and isolated**; that is, **the bytes just below the block, (Left Edge of Block - 1), and just above the block, (Right Edge of Block), have NOT been received**.

SUSTech
Southern University
of Science and Technology

SACK option(2)

- SACK option , which may be sent over an established connection once permission has been given by SACK-permitted.




```
TCP SACK Option:

Kind: 5

Length: Variable

                        +--------+--------+
                        | Kind=5 | Length |
        +--------+--------+--------+--------+
        |      Left Edge of 1st Block       |
        +--------+--------+--------+--------+
        |      Right Edge of 1st Block      |
        +--------+--------+--------+--------+
        |                                   |
        /            . . .                  /
        |                                   |
        +--------+--------+--------+--------+
        |      Left Edge of nth Block       |
        +--------+--------+--------+--------+
        |      Right Edge of nth Block      |
        +--------+--------+--------+--------+
```

Wireshark tips: TCP.option_kind==5

SUSTech Southern University of Science and Technology

SACK option(3)

No.	Time	Source	Destination	Protocol	Info
199	46.985513	gaia.cs.umass.…	192.168.88.149	TCP	http(80) → 54861 [ACK] Seq=151841 Ack=333 Win=30336 Len=1460 [TCP segment of a reassembled PDU]
200	46.985600	192.168.88.149	gaia.cs.umass.…	TCP	54861 → http(80) [ACK] Seq=333 Ack=153301 Win=65536 Len=0
201	47.595142	gaia.cs.umass.…	192.168.88.149	TCP	[TCP Previous segment not captured] http(80) → 54861 [ACK] Seq=156221 Ack=333 Win=30336 Len=1460 [TCP …
202	47.595144	gaia.cs.umass.…	192.168.88.149	TCP	http(80) → 54861 [ACK] Seq=157681 Ack=333 Win=30336 Len=1460 [TCP segment of a reassembled PDU]
203	47.595274	192.168.88.149	gaia.cs.umass.…	TCP	[TCP Dup ACK 200#1] 54861 → http(80) [ACK] Seq=333 Ack=153301 Win=65536 Len=0 SLE=156221 SRE=157681
204	47.595443	192.168.88.149	gaia.cs.umass.…	TCP	[TCP Dup ACK 200#2] 54861 → http(80) [ACK] Seq=333 Ack=153301 Win=65536 Len=0 SLE=156221 SRE=159141
205	48.207253	gaia.cs.umass.…	192.168.88.149	TCP	[TCP Retransmission] http(80) → 54861 [ACK] Seq=153301 Ack=333 Win=30336 Len=1460
206	48.207367	192.168.88.149	gaia.cs.umass.…	TCP	54861 → http(80) [ACK] Seq=333 Ack=154761 Win=65536 Len=0 SLE=156221 SRE=159141
207	49.742628	gaia.cs.umass.…	192.168.88.149	TCP	[TCP Retransmission] http(80) → 54861 [ACK] Seq=154761 Ack=333 Win=30336 Len=1460
208	49.742765	192.168.88.149	gaia.cs.umass.…	TCP	54861 → http(80) [ACK] Seq=333 Ack=159141 Win=65536 Len=0
209	50.363845	gaia.cs.umass.…	192.168.88.149	TCP	http(80) → 54861 [ACK] Seq=159141 Ack=333 Win=30336 Len=1460 [TCP segment of a reassembled PDU]

#203 and #204 are SACK

#203 tells that 156221~157681 are **contiguous and isolated**

#204 tells that 156221~159141 are **contiguous and isolated**

#200 tells that the block before 153301 are acked

So

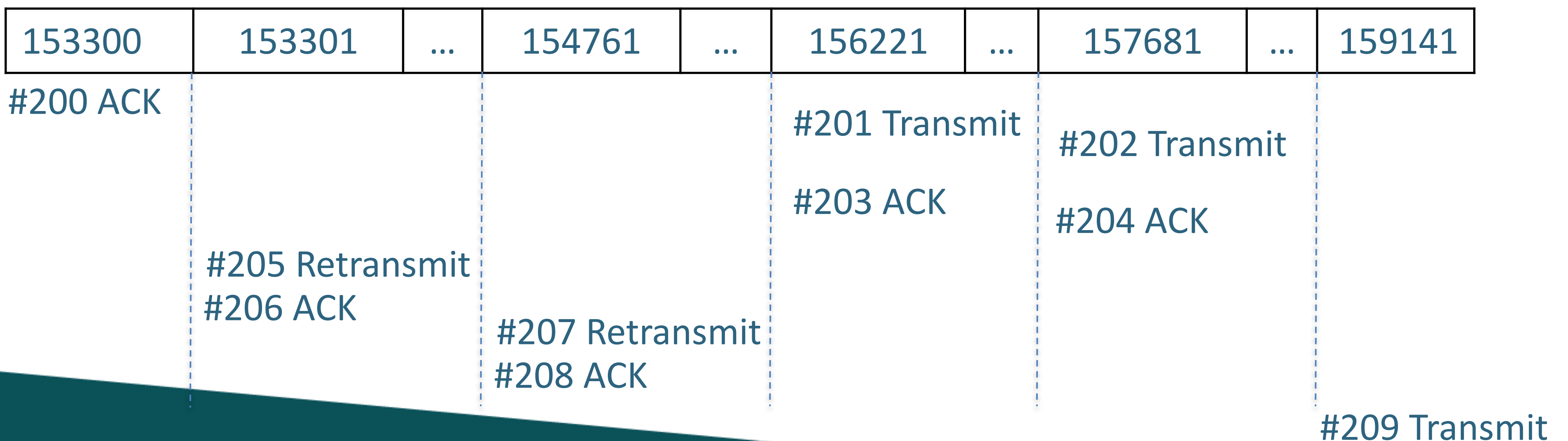
#205 retransmit 153301 ~ 153301+ 1460 -1 ,#206 ack it with 154761

#207 retransmit 154761 ~ 154761+1460 -1

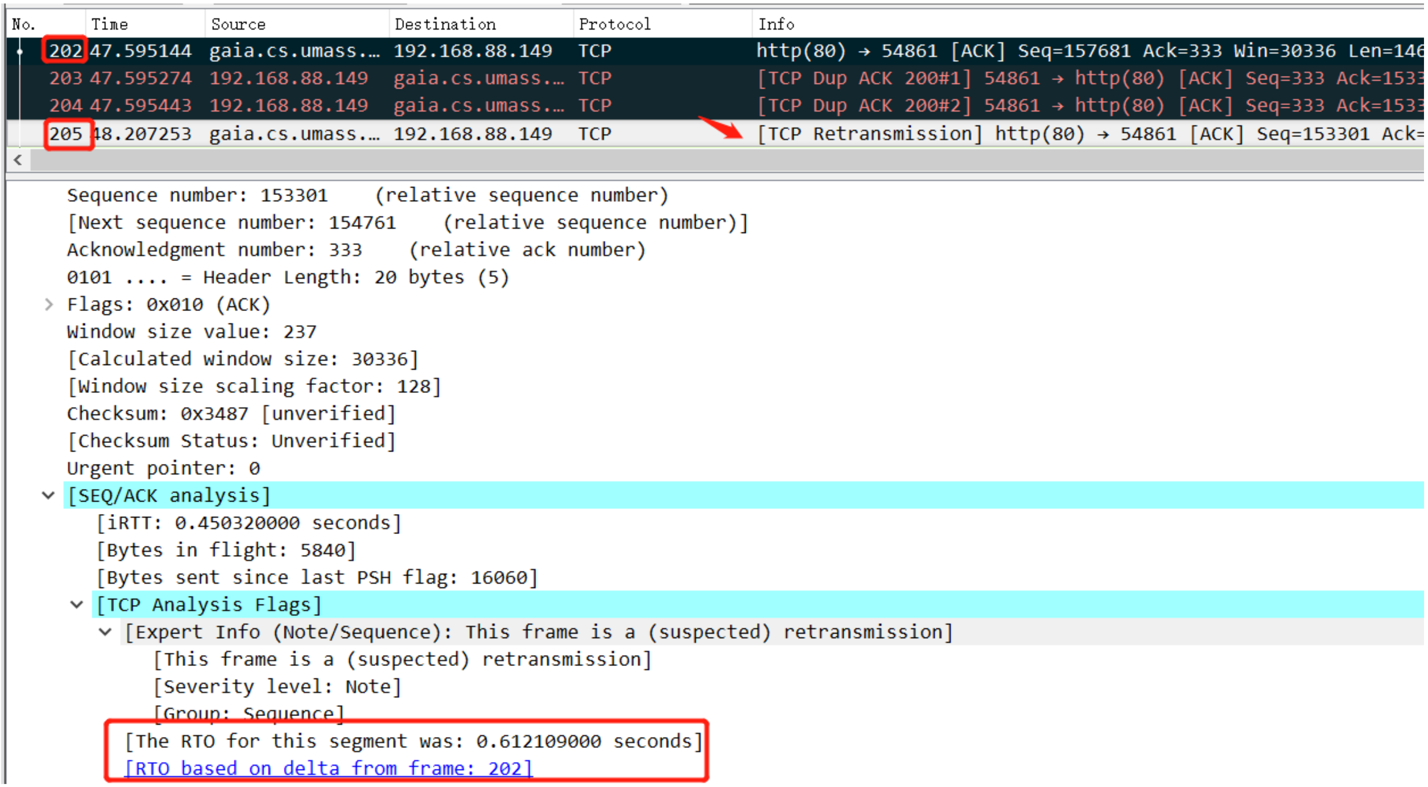
#208 ack it with 159141 (for 157681~159140 are **contiguous but NOT isolated**)

SACK option(4)

No.	Time	Source	Destination	Protocol	Info
199	46.985513	gaia.cs.umass.…	192.168.88.149	TCP	http(80) → 54861 [ACK] Seq=151841 Ack=333 Win=30336 Len=1460 [TCP segment of a reassembled PDU]
200	46.985600	192.168.88.149	gaia.cs.umass.…	TCP	54861 → http(80) [ACK] Seq=333 Ack=153301 Win=65536 Len=0
201	47.595142	gaia.cs.umass.…	192.168.88.149	TCP	[TCP Previous segment not captured] http(80) → 54861 [ACK] Seq=156221 Ack=333 Win=30336 Len=1460 [TCP …
202	47.595144	gaia.cs.umass.…	192.168.88.149	TCP	http(80) → 54861 [ACK] Seq=157681 Ack=333 Win=30336 Len=1460 [TCP segment of a reassembled PDU]
203	47.595274	192.168.88.149	gaia.cs.umass.…	TCP	[TCP Dup ACK 200#1] 54861 → http(80) [ACK] Seq=333 Ack=153301 Win=65536 Len=0 SLE=156221 SRE=157681
204	47.595443	192.168.88.149	gaia.cs.umass.…	TCP	[TCP Dup ACK 200#2] 54861 → http(80) [ACK] Seq=333 Ack=153301 Win=65536 Len=0 SLE=156221 SRE=159141
205	48.207253	gaia.cs.umass.…	192.168.88.149	TCP	[TCP Retransmission] http(80) → 54861 [ACK] Seq=153301 Ack=333 Win=30336 Len=1460
206	48.207367	192.168.88.149	gaia.cs.umass.…	TCP	54861 → http(80) [ACK] Seq=333 Ack=154761 Win=65536 Len=0 SLE=156221 SRE=159141
207	49.742628	gaia.cs.umass.…	192.168.88.149	TCP	[TCP Retransmission] http(80) → 54861 [ACK] Seq=154761 Ack=333 Win=30336 Len=1460
208	49.742765	192.168.88.149	gaia.cs.umass.…	TCP	54861 → http(80) [ACK] Seq=333 Ack=159141 Win=65536 Len=0
209	50.363845	gaia.cs.umass.…	192.168.88.149	TCP	http(80) → 54861 [ACK] Seq=159141 Ack=333 Win=30336 Len=1460 [TCP segment of a reassembled PDU]



Retransmission(1)



While RTO timeout , retransmission is triggered

Retransmission(2)

No.	Time	Source	Destination	Protocol	Info
202	47.595144	gaia.cs.umass.…	192.168.88.149	TCP	http(80) → 54861 [ACK] Seq=157681 Ack=333 Win=30336 Len=146
203	47.595274	192.168.88.149	gaia.cs.umass.…	TCP	[TCP Dup ACK 200#1] 54861 → http(80) [ACK] Seq=333 Ack=1533
204	47.595443	192.168.88.149	gaia.cs.umass.…	TCP	[TCP Dup ACK 200#2] 54861 → http(80) [ACK] Seq=333 Ack=1533
205	48.207253	gaia.cs.umass.…	192.168.88.149	TCP	[TCP Retransmission] http(80) → 54861 [ACK] Seq=153301 Ack=
206	48.207367	192.168.88.149	gaia.cs.umass.…	TCP	54861 → http(80) [ACK] Seq=333 Ack=154761 Win=65536 Len=0 SL
207	49.742628	gaia.cs.umass.…	192.168.88.149	TCP	[TCP Retransmission] http(80) → 54861 [ACK] Seq=154761 Ack=

```
Sequence number: 154761    (relative sequence number)
[Next sequence number: 156221    (relative sequence number)]
Acknowledgment number: 333    (relative ack number)
0101 .... = Header Length: 20 bytes (5)
Flags: 0x010 (ACK)
Window size value: 237
[Calculated window size: 30336]
[Window size scaling factor: 128]
Checksum: 0x595f [unverified]
[Checksum Status: Unverified]
Urgent pointer: 0
[SEQ/ACK analysis]
  [iRTT: 0.450320000 seconds]
  [Bytes in flight: 4380]
  [Bytes sent since last PSH flag: 17520]
  [TCP Analysis Flags]
    [Expert Info (Note/Sequence): This frame is a (suspected) retransmission]
      [This frame is a (suspected) retransmission]
      [Severity level: Note]
      [Group: Sequence]
  [The RTO for this segment was: 2.147484000 seconds]
  [RTO based on delta from frame: 202]
```

SUSTech
Southern University
of Science and Technology

Fast retransmission

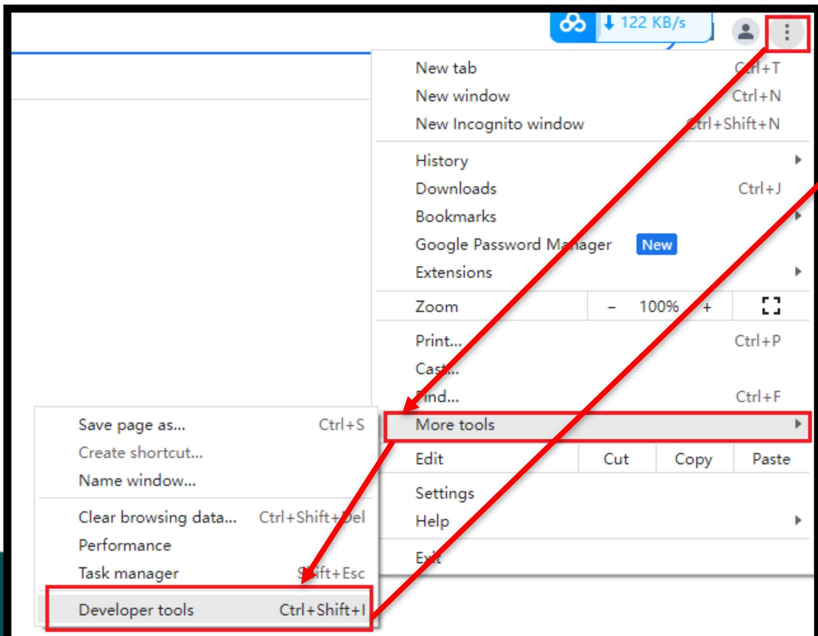
- TCP may generate an immediate acknowledgment (a duplicate ACK) when an out-of-order segment is received. This duplicate ACK should not be delayed. The purpose of this duplicate ACK is to let the other end know that a segment was received out of order, and to tell it what sequence number is expected.
- Since TCP does not know whether a duplicate ACK is caused by a lost segment or just a reordering of segments, it waits for a small number of duplicate ACKs to be received.
  - It is assumed that if there is just a reordering of the segments, there will be only one or two duplicate ACKs before the reordered segment is processed, which will then generate a new ACK.
  - If three or more duplicate ACKs are received in a row, it is a strong indication that a segment has been lost.
- TCP then performs a retransmission of what appears to be the missing segment, without waiting for a retransmission timer to expire.

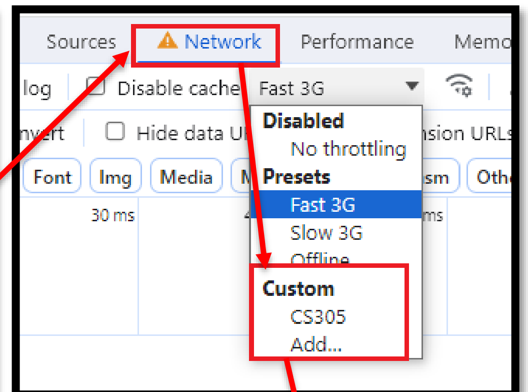
115	10.757197	192.168.88.149	gaia.cs.umass.…	TCP	[TCP Dup ACK 113#1] 54861 → http(80) [ACK] Seq=333 Ack=87601 Win=49408 Len=0 SLE=90521 SRE=91981
116	10.758693	gaia.cs.umass.…	192.168.88.149	TCP	http(80) → 54861 [ACK] Seq=91981 Ack=333 Win=30336 Len=1460 [TCP segment of a reassembled PDU]
117	10.758765	192.168.88.149	gaia.cs.umass.…	TCP	[TCP Dup ACK 113#2] 54861 → http(80) [ACK] Seq=333 Ack=87601 Win=49408 Len=0 SLE=90521 SRE=93441
118	11.340240	gaia.cs.umass.…	192.168.88.149	TCP	http(80) → 54861 [ACK] Seq=93441 Ack=333 Win=30336 Len=1460 [TCP segment of a reassembled PDU]
119	11.340311	192.168.88.149	gaia.cs.umass.…	TCP	[TCP Dup ACK 113#3] 54861 → http(80) [ACK] Seq=333 Ack=87601 Win=49408 Len=0 SLE=90521 SRE=94901
120	11.341761	gaia.cs.umass.…	192.168.88.149	TCP	http(80) → 54861 [ACK] Seq=94901 Ack=333 Win=30336 Len=1460 [TCP segment of a reassembled PDU]
121	11.341762	gaia.cs.umass.…	192.168.88.149	TCP	[TCP Fast Retransmission] http(80) → 54861 [ACK] Seq=87601 Ack=333 Win=30336 Len=1460 [TCP segment of …

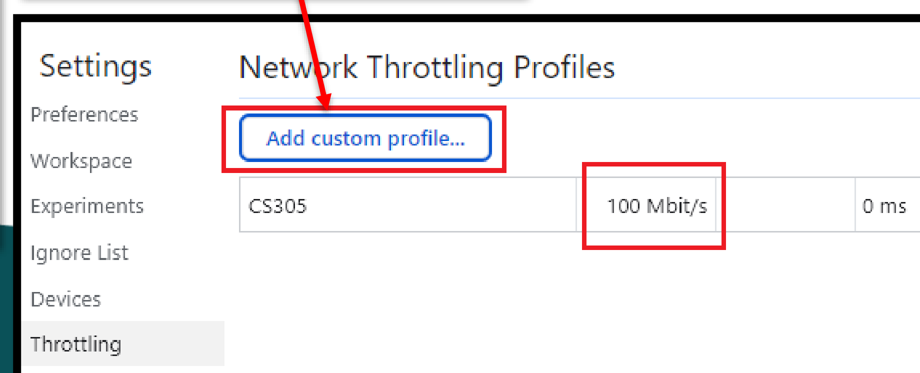
https://tools.ietf.org/html/rfc2001

Tips 1

- How to cause congestion on your network.
  - You can use some tools, such as clumsy-0.2-win64, or Burp Suite, …
  - You can make your browser work in a slow mode.

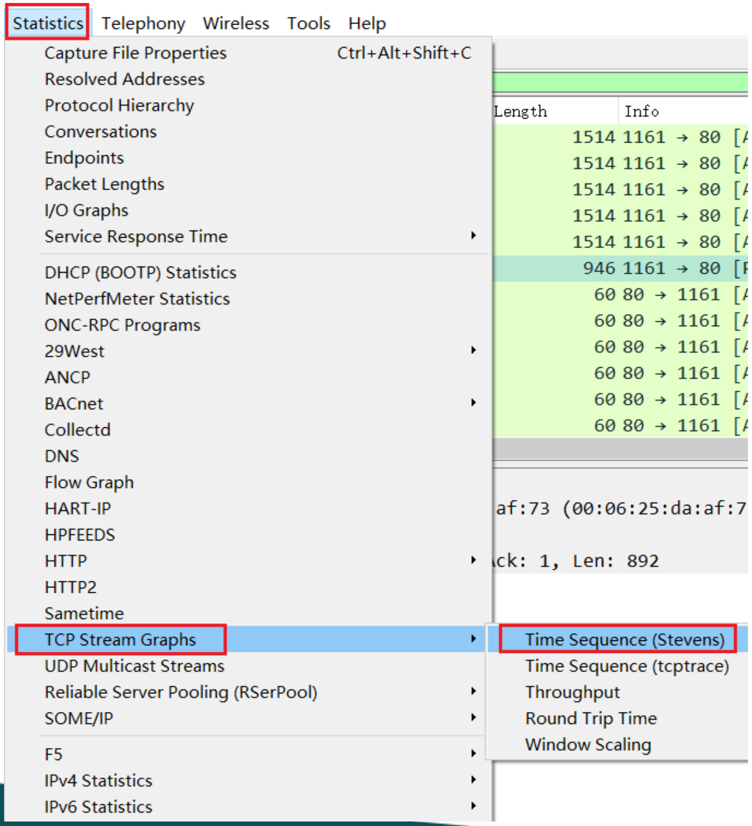


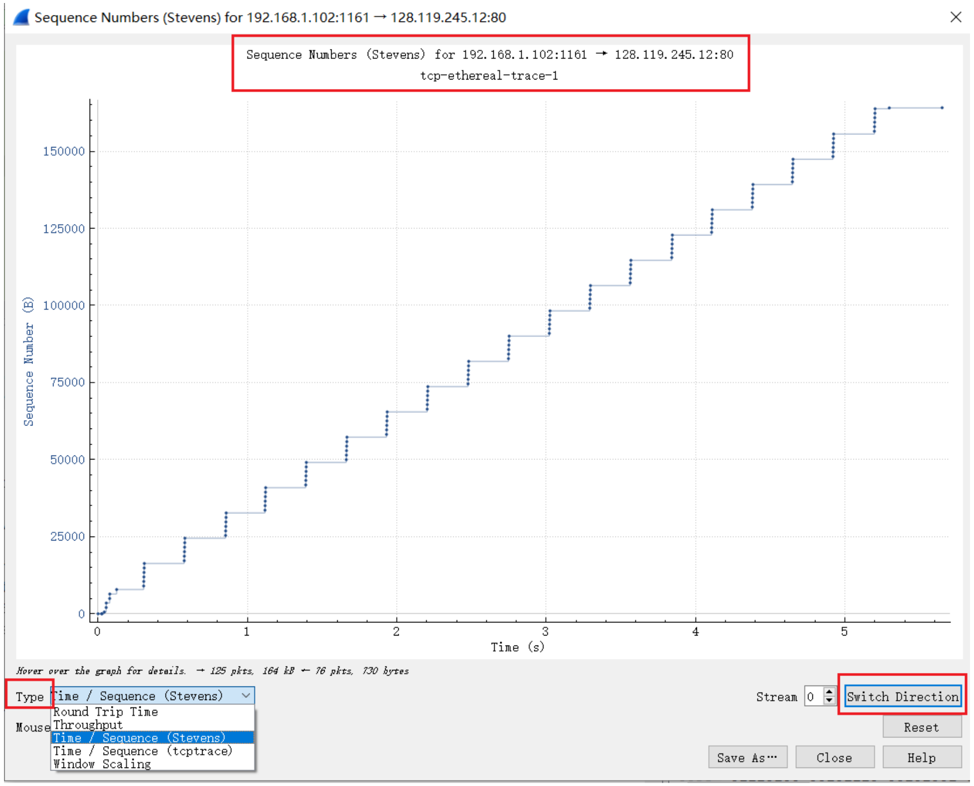




Practice 8.1

- Finish the question 13 of Wireshark_TCP_v8.0.pdf

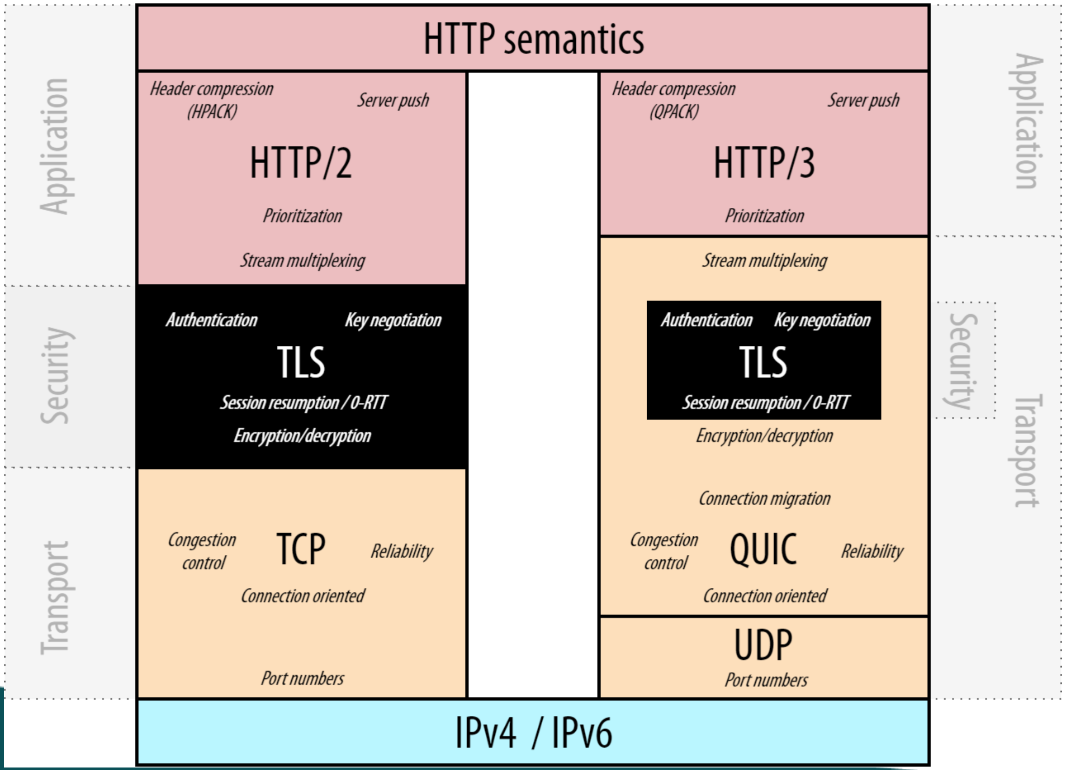




SUSTech
Southern University
of Science and Technology

Part B. QUIC

- QUIC: Quick UDP Internet Connections
- A UDP-Based Multiplexed and Secure Transport

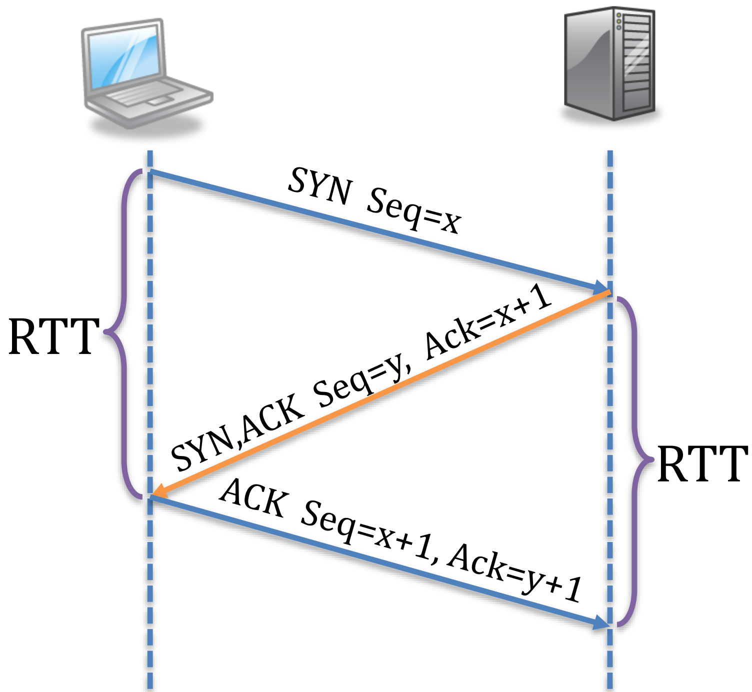


SUSTech
Southern University
of Science and Technology

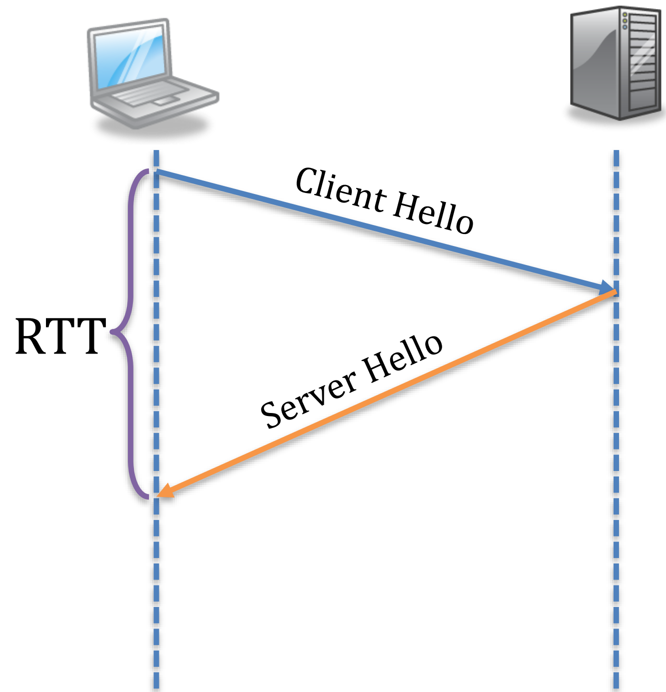
QUIC Working Group: https://quicwg.org/

Part B.1 Connection establishing(1)

- HTTPS: 1-RTT(HTTP) + 1-RTT(TLS) = 2-RTT



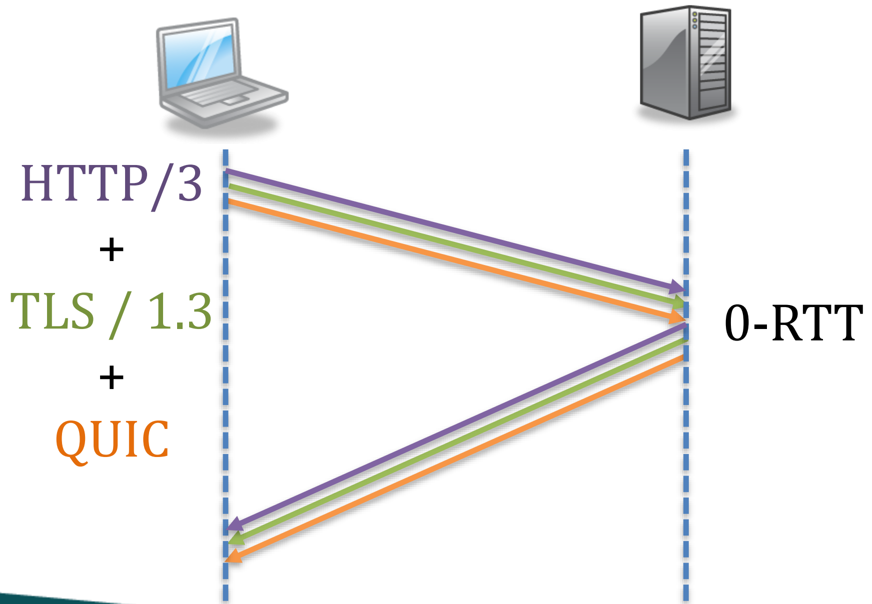
Establishing a TCP connection



TLS handshake

Part B.1 Connection establishing(2)

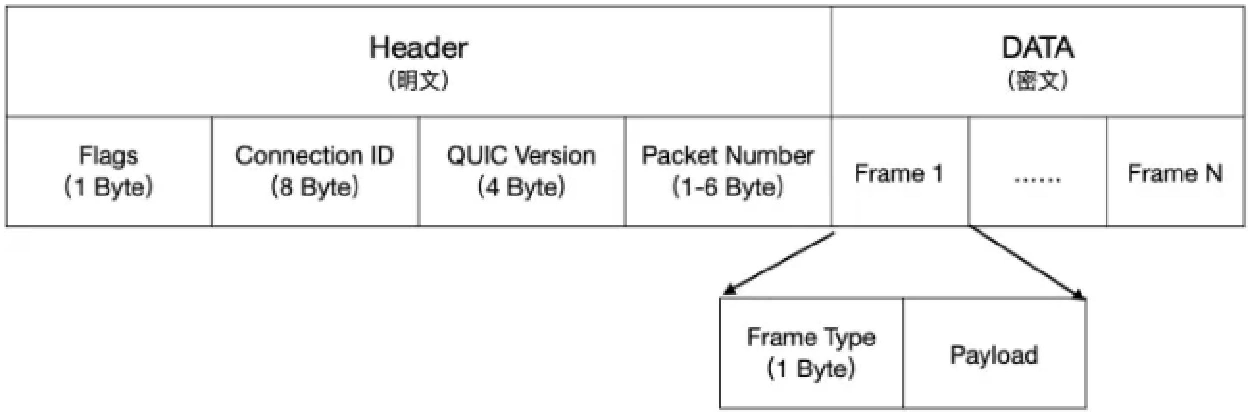
- How does QUIC achieve 0-RTT?
  - The client has cached the ServerConfig (server DH parameter B), and the next time the connection is established, the cached data will be directly used to calculate the communication key.



SUSTech
Southern University
of Science and Technology

Part B.2 QUIC packet structure

- A QUIC data packet consists of two parts: header and data.
  - The header is unencrypted and contains four fields: Flags, Connection ID, QUIC Version, and Packet Number.
  - Data is encrypted and contains one or more frames.
    - Each frame is divided into type and payload, where payload is the application data.



SUSTech
Southern University of Science and Technology

Part B.2 QUIC frame

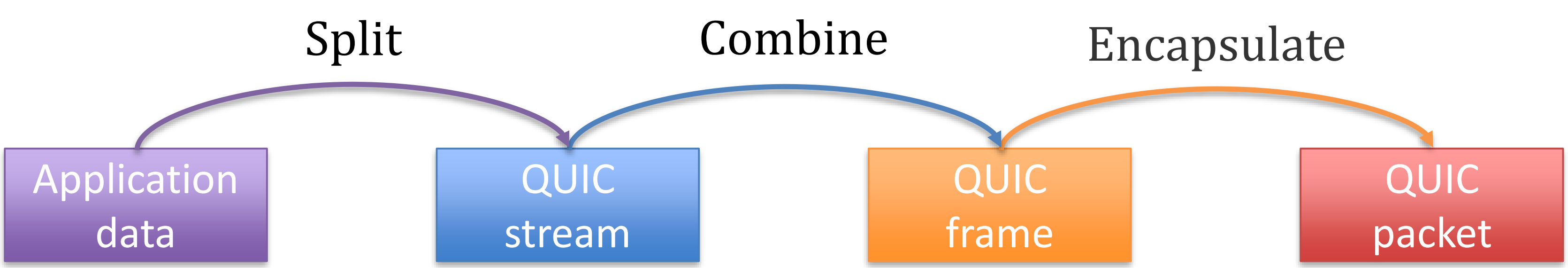
- Frame types of QUIC

Type Value	Frame Type Name	Pkts	Spec
0x00	PADDING	IH01	NP
0x01	PING	IH01	
0x02-0x03	ACK	IH_1	NC
0x04	RESET_STREAM	__01	
0x05	STOP_SENDING	__01	
0x06	CRYPTO	IH_1	
0x07	NEW_TOKEN	___1	
0x08-0x0f	STREAM	__01	F
0x10	MAX_DATA	__01	
0x11	MAX_STREAM_DATA	__01	

Type Value	Frame Type Name	Pkts	Spec
0x12-0x13	MAX_STREAMS	__01	
0x14	DATA_BLOCKED	__01	
0x15	STREAM_DATA_BLOCKED	__01	
0x16-0x17	STREAMS_BLOCKED	__01	
0x18	NEW_CONNECTION_ID	__01	P
0x19	RETIRE_CONNECTION_ID	__01	
0x1a	PATH_CHALLENGE	__01	P
0x1b	PATH_RESPONSE	___1	P
0x1c-0x1d	CONNECTION_CLOSE	ih01	N
0x1e	HANDSHAKE_DONE	___1	

Part B.2 QUIC stream frame

- Streams in QUIC provide a lightweight, ordered byte-stream abstraction to an application.
- QUIC allows for an arbitrary number of streams to operate concurrently and for an arbitrary amount of data to be sent on any stream.



Part B.2 QUIC stream frame format(1)

Frame Type (1 Byte)	Payload			
	Stream ID (1-4 Byte)	offset (0-8 Byte)	Data Length (0-2 Byte)	Data (应用数据)

- The Type field in the STREAM frame takes the form 0b00001XXX(or the set of values from 0x08 to 0x0f).
  - The OFF bit indicates whether there is an Offset field present.
  - The LEN bit indicates whether there is a Length field present.
  - The FIN bit indicates that the frame marks the end of the stream.

					OFF bit	LEN bit	FIN bit
Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0

Part B.2 QUIC stream frame format(2)

Frame Type (1 Byte)	Payload			
	Stream ID (1-4 Byte)	offset (0-8 Byte)	Data Length (0-2 Byte)	Data (应用数据)

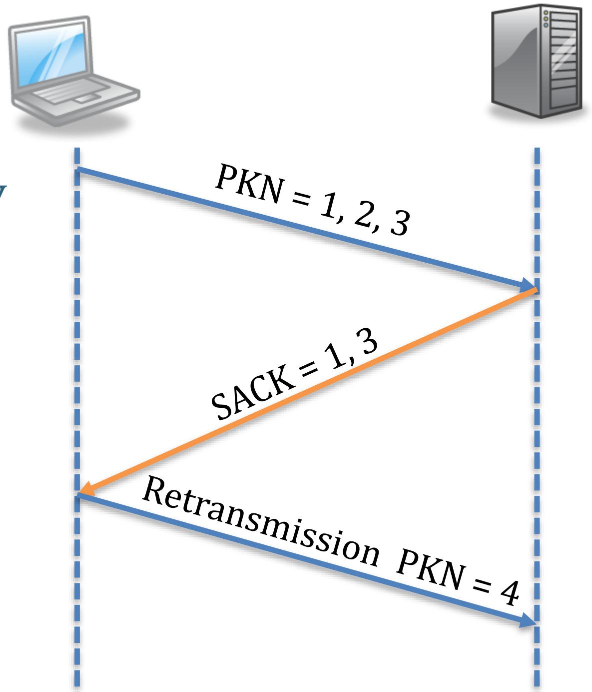
- Stream ID
  - A stream ID is a 62-bit integer (0 to 2^{62}-1) that is unique for all streams on a connection.
  - Stream IDs are encoded as variable-length integers.
- Offset: A variable-length integer specifying the byte offset in the stream for the data in this STREAM frame.
- Data Length: A variable-length integer specifying the length of the Stream Data field in this STREAM frame.
- Data: The bytes from the designated stream to be delivered.

Part B.3 Reliability(1)

- QUIC runs over UDP protocol, and UDP is unreliable. How does QUIC achieve reliable transmission?
- Reliability of QUIC
  - ACK frame
  - Package Number (PKN) and Confirmation Response (SACK)
  - Orderliness of data

Part B.3 Reliability(2)

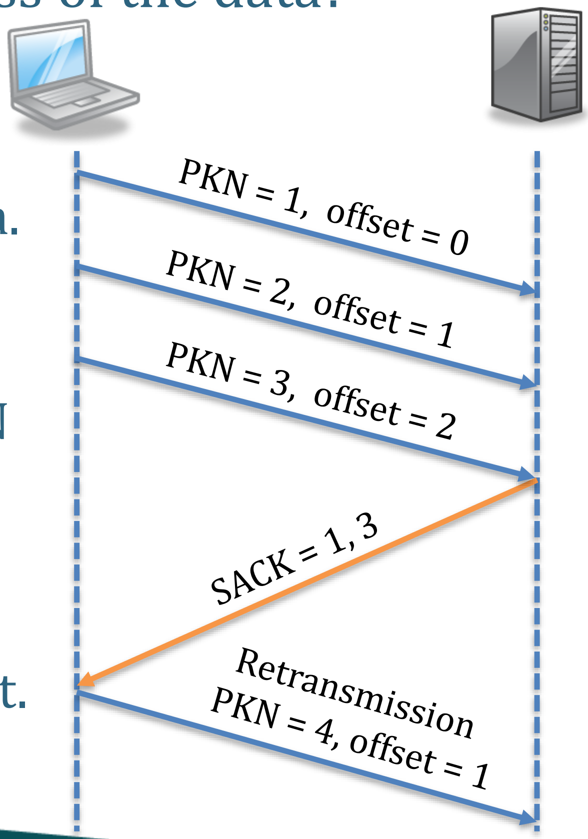
- Q: How does the sending end know if the sent packet has been received by the receiving end?
- A: By Package Number (PKN) and Confirmation Response (SACK)
  - Client: Send 3 data packets to the server (PKN = 1, 2, 3)
  - Server: Notify the client through SACK that they have received 1 and 3, but not 2
  - Client: Retransmit the second packet (PKN=4)
  - It can be seen that the packet numbers of QUIC are monotonically increasing. The previously sent packet (PKN=2) and the retransmitted packet (PKN=4), although the data is the same, they have different packet numbers.



SUSTech
Southern University of Science and Technology

Part B.3 Reliability(3)

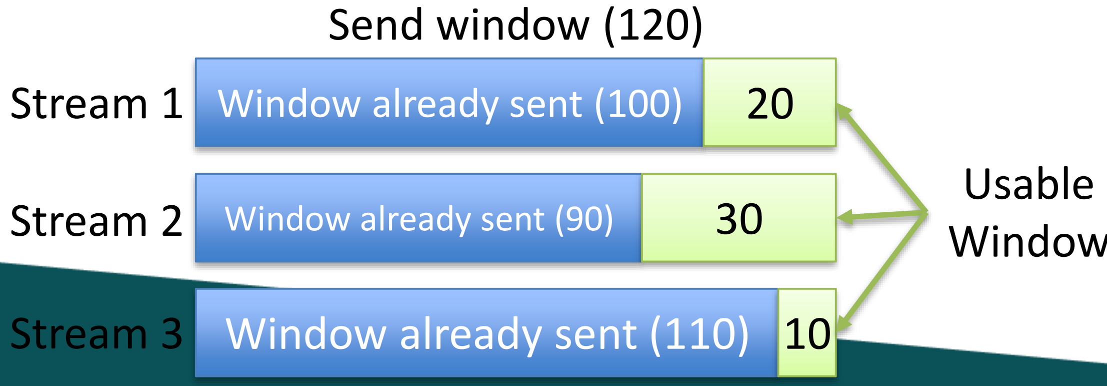
- Q: Since the package number is monotonically increasing, how can the receiving end ensure the orderliness of the data?
- A: By data offset
  - Each packet has an offset field that represents the offset within the entire data.
  - The receiving end can sort the asynchronously arrived packets based on the offset field. Why does QUIC design PKN to be monotonically increasing? Resolve TCP retransmission ambiguity issue.
  - The package numbers of the original and retransmitted packets in QUIC are different.



SUSTech
Southern University
of Science and Technology

Part B.4 Flow control

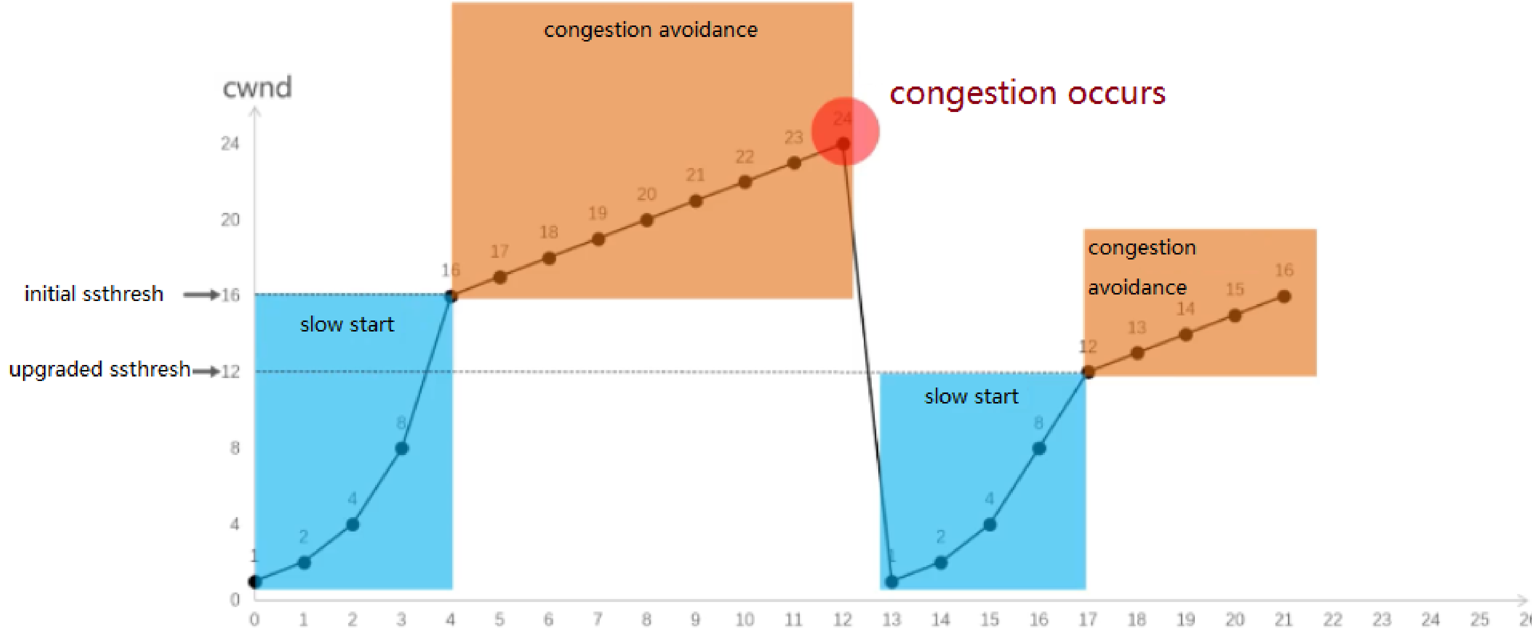
- QUIC also utilizes sliding window mechanism for traffic control as TCP does.
- Unlike TCP, QUIC's sliding window is divided into two levels: Connection and Stream. Connection traffic control specifies the total window size for all data streams, and stream flow control specifies the window size for each stream.
  - Assuming there are three Streams, and the sliding windows are as shown below, The available window size for the entire Connection is: 20+30+10=60.



SUSTech
Southern University of Science and Technology

Part B.5 Congestion control(1)

- Cubic congestion control algorithm is used by default.
- The same mechanism as TCP (such as slow start, congestion avoidance, fast retransmission, and fast recovery strategies)



SUSTech
Southern University
of Science and Technology

Part B.5 Congestion control(2)

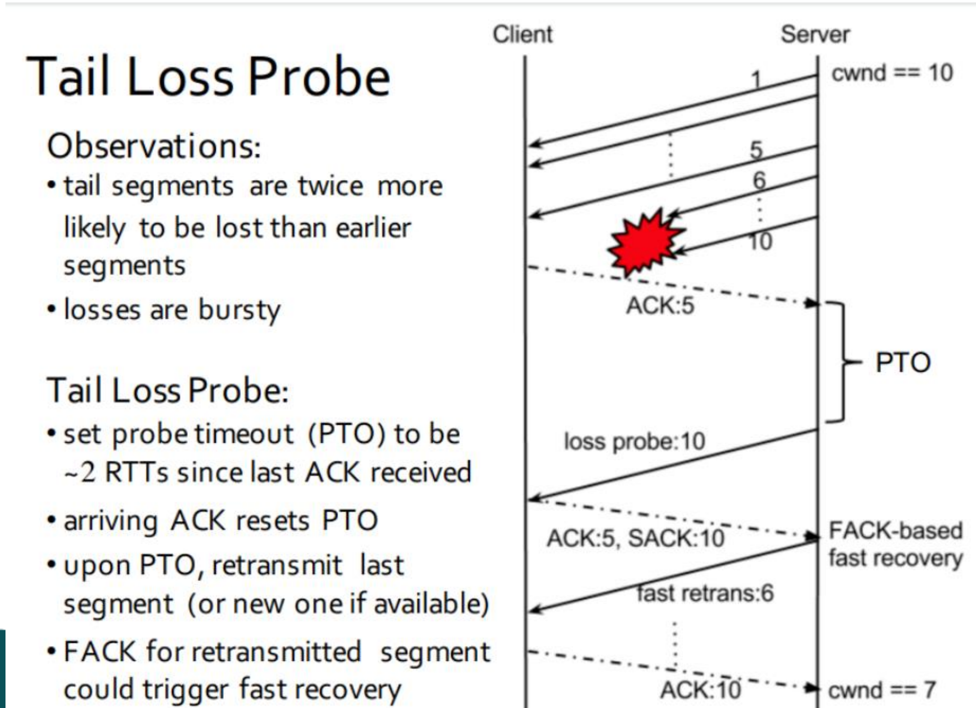
- The TCP protocol is built into the system protocol stack, and if you want to change its congestion control strategy, you need to make modifications at the system level.
- QUIC is based on UDP, you only need to make modifications at the application layer when needed.
- Currently, Google offers a variety of algorithms to choose from and provides a flexible interface to experiment with new congestion control algorithms.
  - New Reno: packet loss detection based
  - CUBIC: packet loss detection based
  - BBR: network bandwidth
- Different control strategies can be set for different applications.

Part B.5 Congestion control(3)

- QUIC provides more detailed and accurate information.
  - For example, the strictly monotonically increasing PKN can easily distinguish whether the packet comes from retransmission or first transmission, avoiding retransmission ambiguity.
  - For example, QUIC carries information about the delay between receiving data packets and sending ACKs, which can calculate RTT time more accurately.
  - In addition, compared to TCP's SACK (Selective Acknowledgements, used to indicate which packets have been), QUIC also provides a larger range (0-256) of NACK (Negative Acknowledgements, used to indicate which packets have not been received) to help the sender quickly retransmit lost packets.

Part B.5 Congestion control(4)

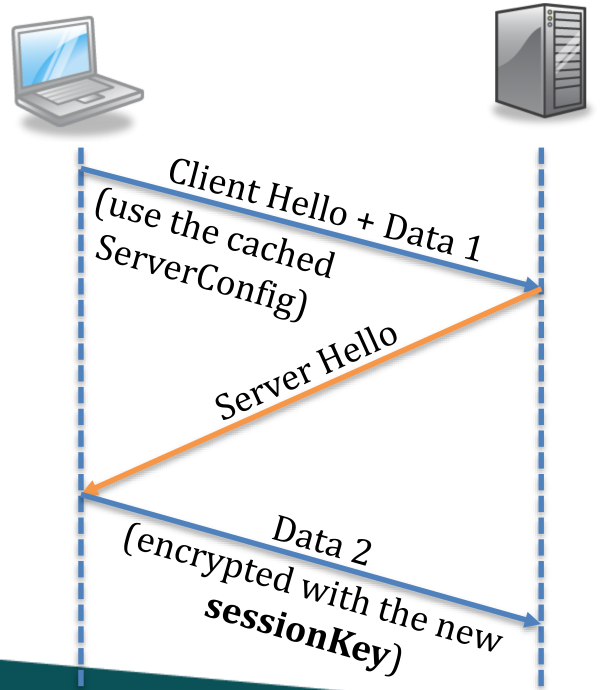
- In order to achieve fast retransmission as much as possible instead of timeout retransmission, QUIC adopts Tail Loss Probes (TLPs) to trigger fast retransmission mechanisms in certain situations.



- The server's segments 6-10 are missing. When the client is waiting for segment 6, the fast retransmission mechanism can not be triggered because they have not received the subsequent sequence.
- After the time reaches the PTO (probe threshold), the TLP algorithm retransmits segments 10(the largest number of the previously sent packages).
- Upon receiving this retransmission sequence, the client can trigger the fast retransmission mechanism.

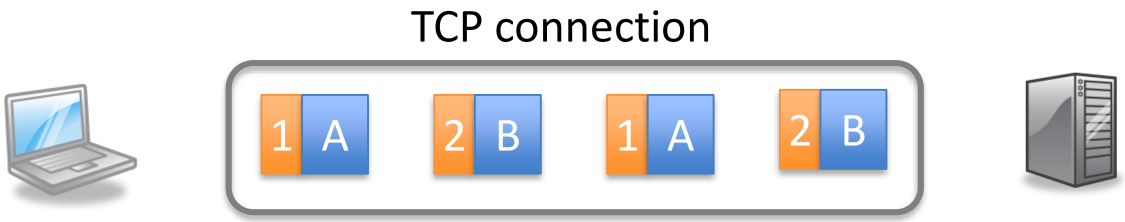
Part B.6 Forward safety

- Forward security: The leakage of long-term keys used to generate session keys, without compromising previous communication content.
  - Client: Generate a random number a, calculate **A**=g^a mod p, send Client Hello and encrypted data packages. It uses the cached ServerConfig (server static configuration) to calculate the initial key initKey: **K**=B^a mod p. (0-RTT)
  - Server: Calculate the initKey **K**=A^b mod p based on the Client Hello message; Generate random number c, calculate **C**=g^c mod p, encrypt C with initKey K, and send Server Hello message.
  - Client: Use initKey K decoding to obtain C, calculate session key **sessionKey**= C^a mod p, encrypt and send application data 2.
  - Server: Calculate session key **sessionKey**= A^c mod p, decrypt to obtain application data 2.



Part B.7 Multiplexing(1)

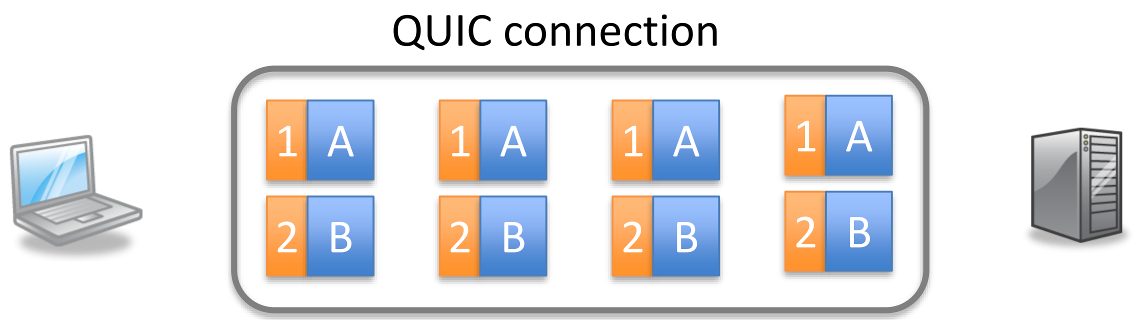
- Multiplexing is one of the main features of HTTP/2.
- The fundamental reason why HTTP/2 can achieve multiplexing is the use of binary frame format data structures.
- A request corresponds to a stream, and the Stream ID can be used to determine which request the data frame belongs to.
  - Assuming there are two requests A and B, with corresponding Stream IDs of 1 and 2, the data transmitted on this TCP connection is approximately as follows:



SUSTech
Southern University
of Science and Technology

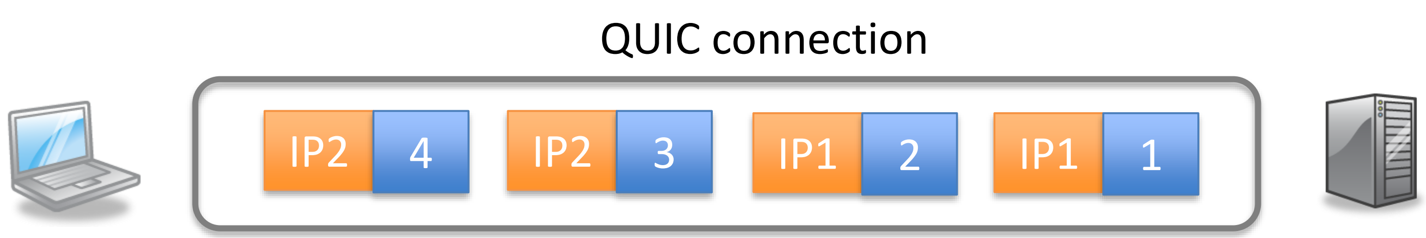
Part B.7 Multiplexing(2)

- Although HTTP/2 solves the Head-of-line blocking of the HTTP layer through multiplexing, there is still Head-of-line blocking of the TCP layer. How does QUIC solve the Head-of-line blocking problem?
- The reason why HTTP/2 has Head-of-line blocking in the TCP layer is that all request flows share the same sliding window. If each request flow is assigned an independent sliding window, this problem can be solved.
  - Packet loss on stream A will not affect data transmission on stream B.



Part B.8 Connection Migration(1)

- When the client switches networks(e.g. switch from wired connection to Wi-Fi), the connection with the server will not be disconnected and communication can still be normal.
- For the TCP protocol, TCP connections are based on 4-tuples: source IP, source port, destination IP, and destination port, as long as one of them changes, a new connection needs to be established.
- The connection of QUIC is based on the Connection ID, and network switching does not affect the change of Connection ID. The connection is still logically connected.



SUSTech
Southern University
of Science and Technology

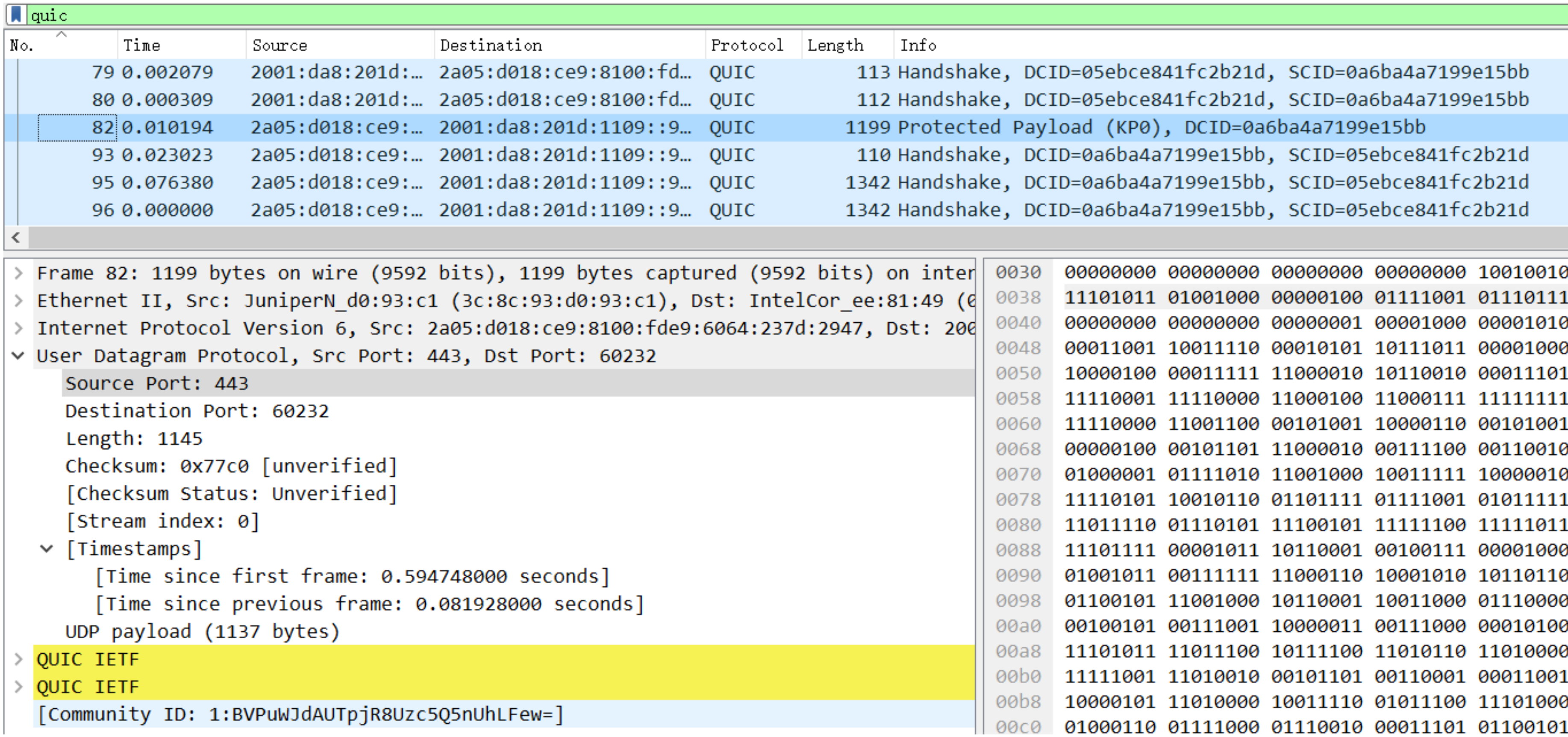
Part B.9 aioquic(1)

- Official documentation: https://aioquic.readthedocs.io/en/latest/
- aioquic is a library for the QUIC network protocol in Python. It features several APIs:
  - a QUIC API following the “bring your own I/O” pattern, suitable for embedding in any framework
  - an HTTP/3 API which also follows the “bring your own I/O” pattern
  - a QUIC convenience API built on top of aioquic, Python’s standard asynchronous I/O framework

Part B.9 aioquic(2)

- Run the aioquic examples.
  - Download examples from Blackboard site or github: https://github.com/aiortc/aioquic
  - Install necessary dependency packages: *aioquic, aiofiles, asgiref, dnslib, httpbin, starlette, wsproto …*
  - Run “python examples/http3_client.py https://quic.aiortc.org/” command on command line under the *aioquic* directory.
- Use Wireshark to capture QUIC packages and make simple analysis.

Part B.9 aioquic(3)



SUSTech
Southern University
of Science and Technology

References

- https://www.rfc-editor.org/rfc/rfc9000.html
- https://cloud.tencent.com/developer/article/1155289
- https://bbs.csdn.net/topics/604347087
- https://bbs.csdn.net/topics/604323396

Practice 8.2 (optional)

- Invoke a QUIC connection by python codes.
- Use Wireshark to capture and analyse the QUIC stream.
- Answer these questions:
  - How many frame types can you observed? What are they? And simply describe their function.
  - What is the Connection ID of packages sent from client? Are they share the same Connection ID or not?
  - What is the Connection ID of packages sent from server? Is it the same as the Connection ID of previous question?

Tips 2

- When running the example codes, if you can observe the output information on running environment, but you can not capture QUIC packages on Wireshark, you can change the configuration of Wireshark as follows:
  - [Edit] -> [Preferences] -> [Protocols] -> [QUIC]
  - change UDP port(s) to 443

```
C:\Users\wq\aioquic>python examples/http3_client.py https://quic.aiortc.org/
2023-04-03 11:31:54,154 INFO quic [c4815b56c18e5329] Duplicate CRYPTO data received for epoch Epoch.HANDSHAKE
2023-04-03 11:31:54,155 INFO quic [c4815b56c18e5329] Duplicate CRYPTO data received for epoch Epoch.HANDSHAKE
2023-04-03 11:31:54,156 INFO quic [c4815b56c18e5329] Duplicate CRYPTO data received for epoch Epoch.HANDSHAKE
2023-04-03 11:31:54,535 INFO quic [c4815b56c18e5329] ALPN negotiated protocol h3
2023-04-03 11:31:54,535 INFO client New session ticket received
2023-04-03 11:31:54,926 INFO client Response received for GET / : 1276 bytes in 0.4 s (0.027 Mbps)
2023-04-03 11:31:54,927 INFO client Push received for GET /style.css : 0 bytes
2023-04-03 11:31:54,928 INFO quic [c4815b56c18e5329] Connection close sent (code 0x100, reason )
```