# COMP90007 Internet Technology

Week10

Yiran (Scott) Ruan

Email: yrrua@unimelb.edu.au

GitHub: https://yiranruan.github.io

# Tutor feedback

- Google ’Casmas tutor feedback’
- Choose COMP90007 -> Yiran Ruan

- Line: https://apps.eng.unimelb.edu.au/casmas/index.php?r=qoct/feedback&subjCode=COMP90007

# Question 1

- Why does UDP exist? Would it not have been enough to just let the user processes send raw IP packets?

# UDP (User Datagram Protocol)

- The Internet protocol suite supports a connectionless transport protocol called UDP (User Datagram Protocol). UDP provides a way for applications to send encapsulated IP datagrams **without having to establish a connection**.
- 8-byte header followed by the payload (segments)

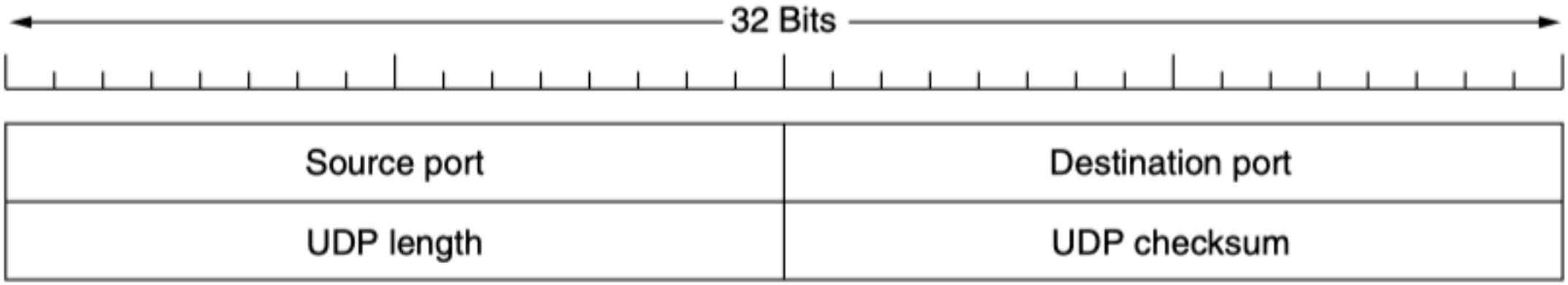


**Figure 6-27.** The UDP header.

# UDP (User Datagram Protocol)

- The two ports serve to identify the endpoints within the source and destination machines. When a UDP packet arrives, its payload is handed to the process attached to the **destination port**.

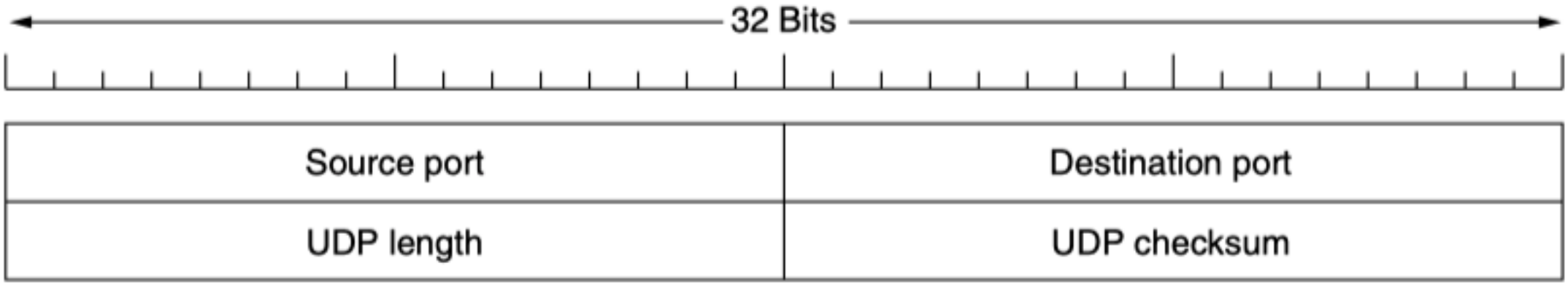


**Figure 6-27.** The UDP header.

# UDP (User Datagram Protocol)

- Without the port fields, the transport layer would not know what to do with each incoming packet. With them, it delivers the embedded segment to the correct application.

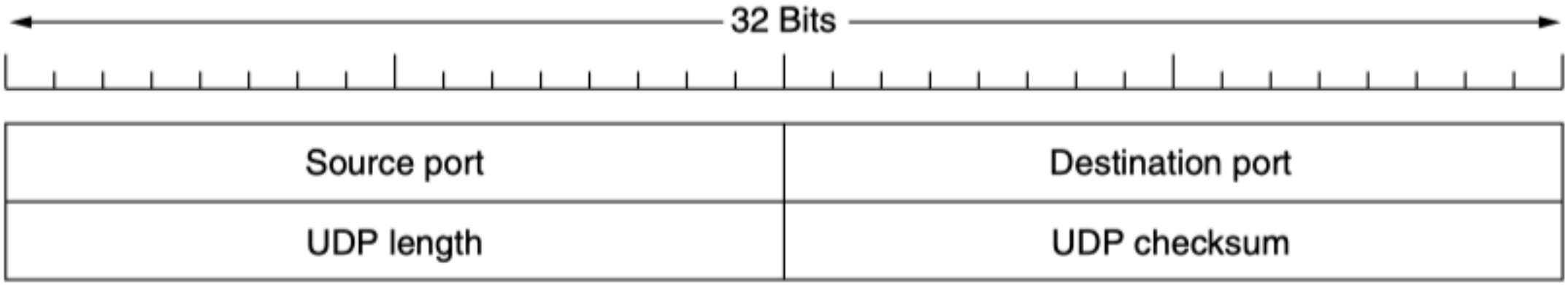


**Figure 6-27.** The UDP header.

# Solution 1

- No. IP packets contain IP addresses, which specify a destination machine. Once such a packet arrived, how would the network handler know which process to give it to? UDP packets contain a destination port. This information is essential so they can be delivered to the correct process.

# Question 2

- Both UDP and TCP use port numbers to identify the destination entity when delivering a message. Discuss possible reasons for why these protocols invented a new abstract ID (port numbers), instead of using process IDs ( which already existed when these protocols were designed?)

# port

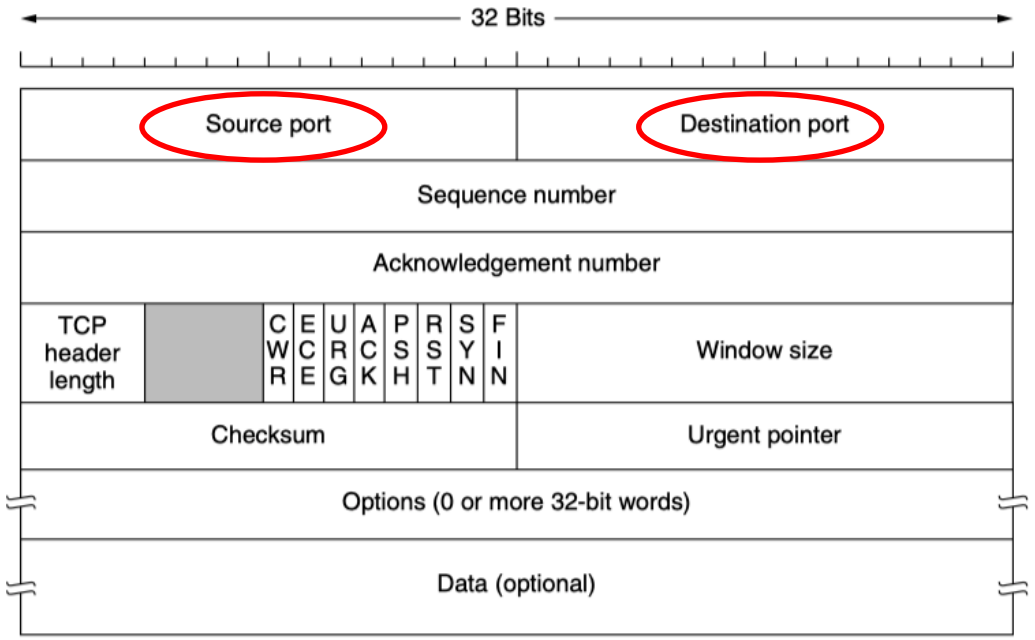


**Figure 6-36.** The TCP header.

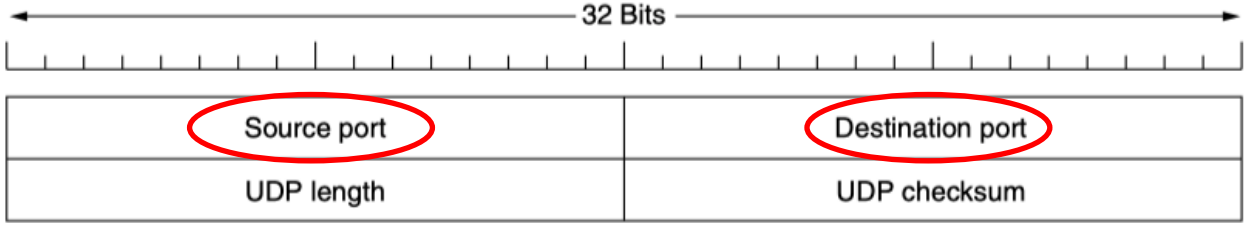


**Figure 6-27.** The UDP header.

# port

- Think of ports as **mailboxes** that applications can rent to receive packets.

# well-known ports

- Port numbers below 1024 are reserved for standard services that can usually only be started by privileged users (e.g., root in UNIX systems). They are called well-known ports.

| Port | Protocol | Use |
|---|---|---|
| 20, 21 | FTP | File transfer |
| 22 | SSH | Remote login, replacement for Telnet |
| 25 | SMTP | Email |
| 80 | HTTP | World Wide Web |
| 110 | POP-3 | Remote email access |
| 143 | IMAP | Remote email access |
| 443 | HTTPS | Secure Web (HTTP over SSL/TLS) |
| 543 | RTSP | Media player control |
| 631 | IPP | Printer sharing |

**Figure 6-34.** Some assigned ports.

# Registered Ports

- Other ports from 1024 through 49151 can be registered with IANA for use by unprivileged users, but applications can and do choose their own ports. For example, the BitTorrent peer-to-peer file-sharing application (unofficially) uses ports 6881–6887, but may run on other ports as well.

# Dynamic and/or Private Ports

- Ports with numbers 49152-65535 are called dynamic and/or private ports.
- The **dynamic port** numbers (also known as the **private port** numbers) are the **port** numbers that are available for use by any application to use in communicating with any other application, using the Internet's Transmission Control Protocol (TCP) or the User Datagram Protocol (UDP).

# Solution 2

- Here are three reasons.
- First, process IDs are OS-specific. Using process IDs would have made these protocols OS-dependent.
- Second, a single process may establish multiple channels of communications. A single process ID (per process) as the destination identifier cannot be used to distinguish between these channels.
- Third, having processes listen on well known ports is easy, but well-known process IDs are impossible.

# Question 3

- What is the key difference between TCP Tahoe and TCP Reno?

# Duplicate acknowledgements

- Because packets can take different paths through the network, they can arrive out of order. This will trigger duplicate acknowledgements even though no packets have been lost. However, this is uncommon in the Internet much of the time.

# Duplicate acknowledgements

- TCP assumes that **three** duplicate acknowledgements imply that a packet has been lost.
- The identity of the lost packet can be inferred from the acknowledgement number as well. It is the very next packet in sequence.
- This packet can then be retransmitted right away, before the retransmission timeout fires.

# Fast Retransmission

- Slow start threshold: set to half the current congestion window, just as with a timeout.
- Slow start can be restarted by setting the congestion window to one packet.
- With this window size, a new packet will be sent after the one round-trip time that it takes to acknowledge the retransmitted packet along with all data that had been sent before the loss was detected.

# TCP Tahoe

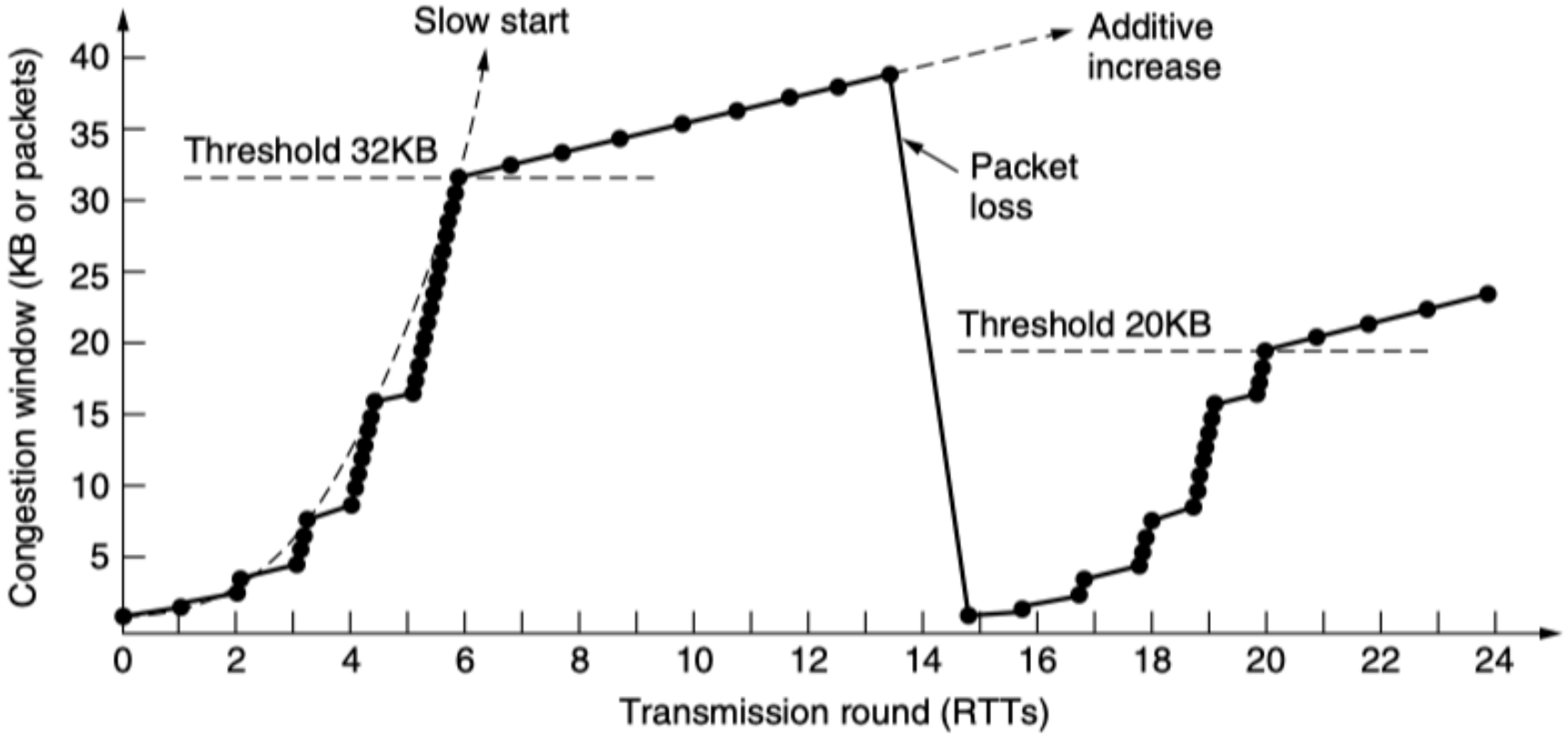


**Figure 6-46.** Slow start followed by additive increase in TCP Tahoe.

# TCP Reno

- After an initial slow start, the congestion window climbs linearly until a packet loss is detected by duplicate acknowledgements. The lost packet is retransmitted and fast recovery is used to keep the ack clock running until the retransmission is acknowledged. At that time, the congestion window is resumed from the new slow start threshold, rather than from 1. This behavior continues indefinitely, and the connection spends most of the time with its congestion window close to the optimum value of the bandwidth-delay product.

# TCP Reno

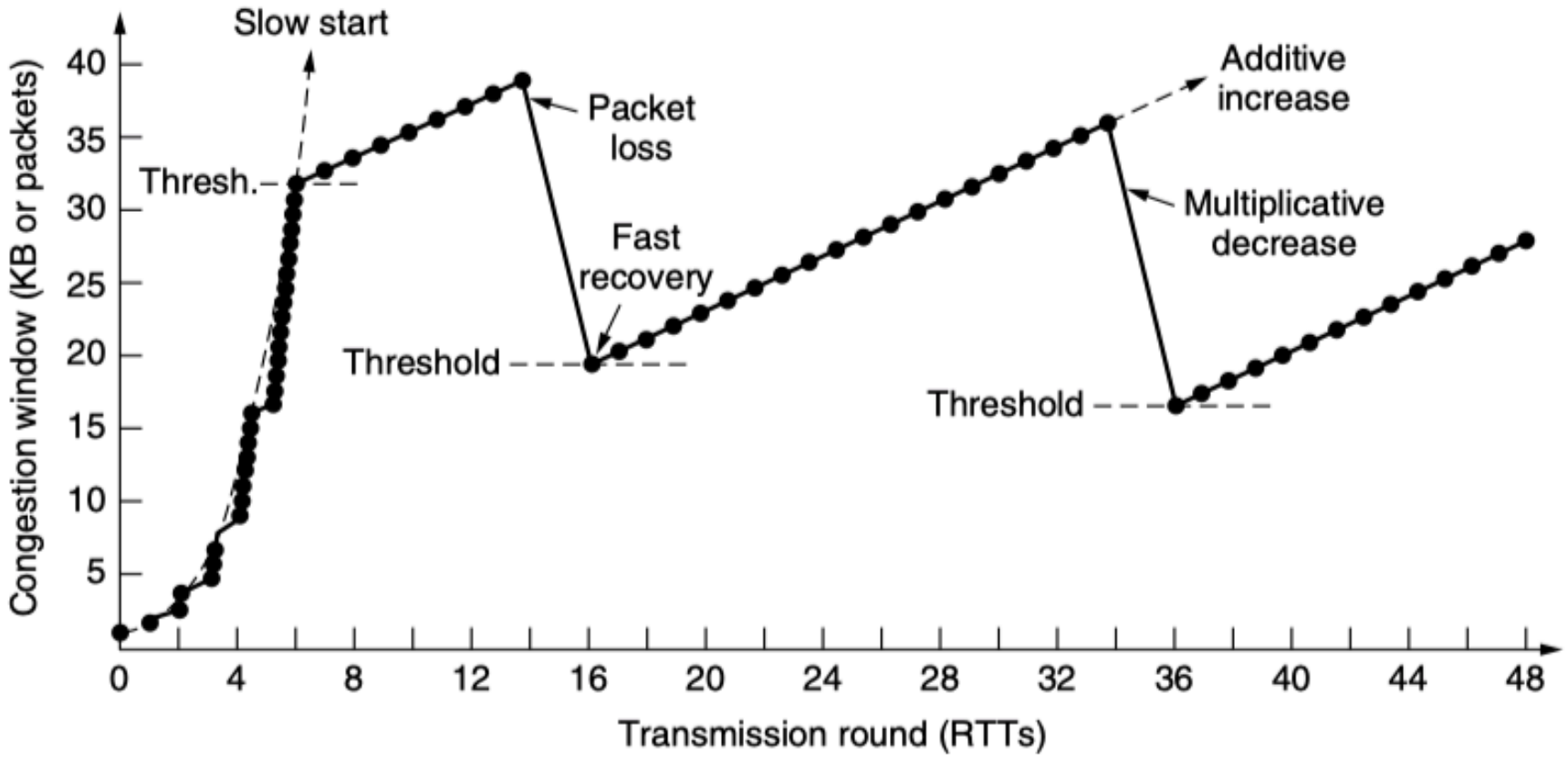


**Figure 6-47.** Fast recovery and the sawtooth pattern of TCP Reno.

# Solution 3

- The key difference, and benefit of Reno, is that Reno avoids Slow start when it can and can do Fast recovery at certain cases.

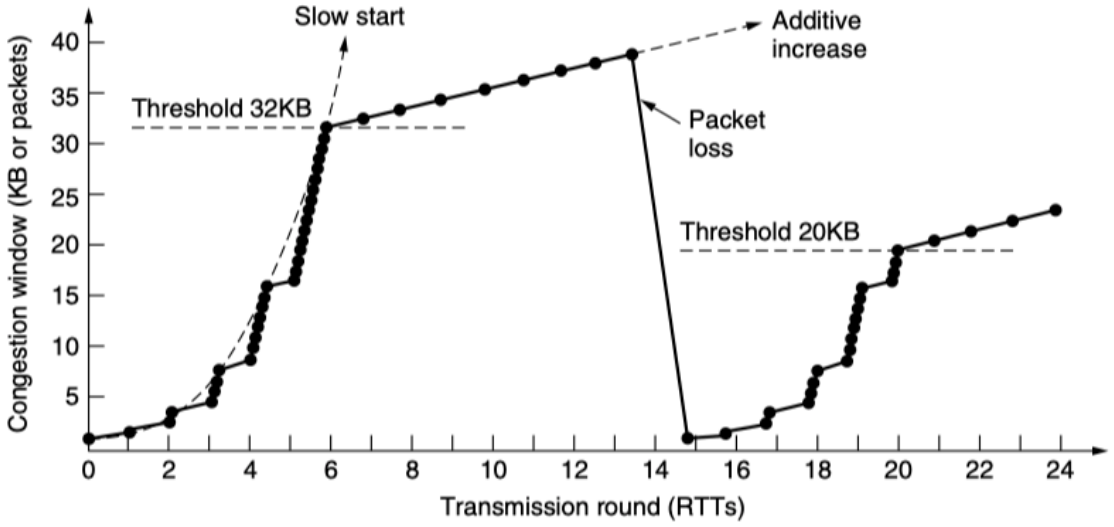


**Figure 6-46.** Slow start followed by additive increase in TCP Tahoe.

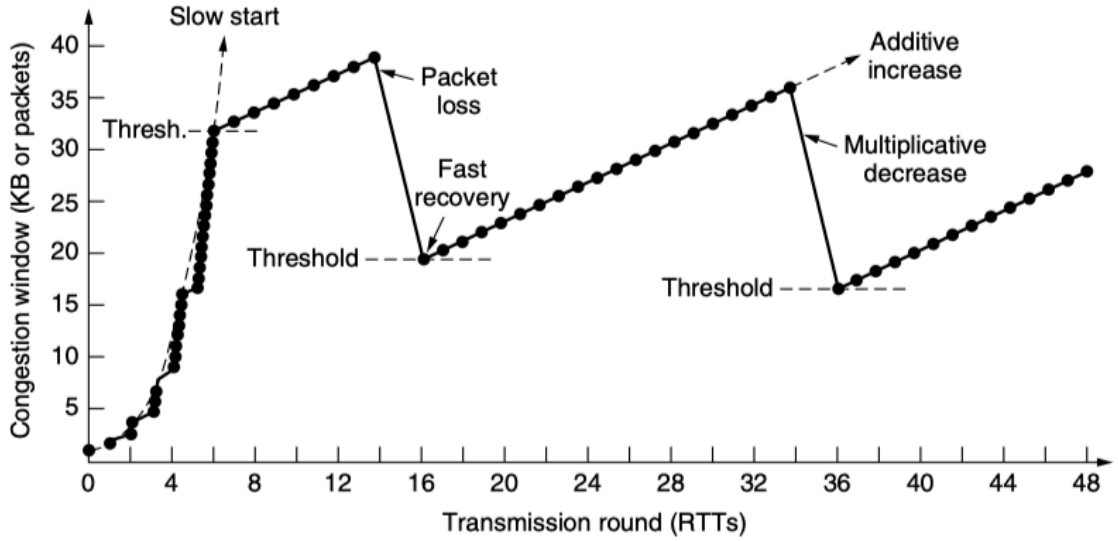


**Figure 6-47.** Fast recovery and the sawtooth pattern of TCP Reno.

# Question 4

- Recall the Leaky Bucket algorithm we saw in class. If a sender has a burst data rate of 20KB/s for 20 seconds as data to send and we have a bucket size of 100KB with a output rate of 10KB/s: Does the Leaky Bucket algorithm achieve its aim of regulating output properly? If so explain how and show your calculations. If not, find the appropriate bucket size and discuss why we need more/less of a bucket size.

# Leaky Bucket

- Large **bursts** of traffic is buffered and smoothed while sending

# Solution 4

- The bucket is too small, we should have had a bucket size of 200KB. The input data is 20x20=400KB, in the same time the bucket can empty only 20x10=200KB and can hold another 100KB. So another 100KB bucket size is needed to deal with 400KB in total.

# Question 5

- TCP relies on timers for resending in case some ACKs are missing. If we set such timers to a fixed value of say 100ms, discuss what would be the advantages and disadvantages of such a static protocol design.

# Solution 5

- A fixed 100ms timer is easy to implement and does not require monitoring network status and load. If conditions are stable and 100ms is set accordingly then all should work fine with little overhead. However, in most cases, network conditions change. Thus 100ms could lead to prematurely timers going off, which means many redundant resends, especially when network is busy... When network conditions indicate fast traffic, then the opposite would be true, i.e., static timers will go off too late for missing segments and applications would wait extra for no good reason. In case of dynamic conditions in the network, timers should be set based on measurements rather than to a static value.