University of Glasgow

Internet Protocols Laboratory

Reliability and Data Transfer

Networked Systems (H)

Lecture 5

Colin Perkins | https://csperkins.org/ | Copyright © 2026 University of Glasgow | This work is licensed under the Creative Commons Attribution-NoDerivatives 4.0 International License.
To view a copy of this license, visit http://creativecommons.org/licenses/by-nd/4.0/ or send a letter to Creative Commons, PO Box 1866, Mountain View, CA 94042, USA.

Lecture Outline

- Packet loss in the Internet
- Unreliable data using UDP
- Reliable data transfer with TCP
- Reliable data transfer with QUIC

Colin Perkins | https://csperkins.org/ | Copyright © 2026 University of Glasgow

University of Glasgow

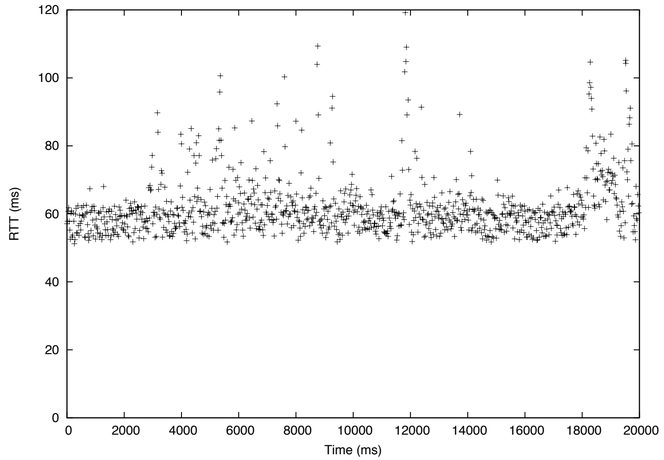
Packet Loss in the Internet

- Best effort traffic
- The end-to-end argument
- Timeliness vs. reliability trade-off

Colin Perkins | https://csperkins.org/ | Copyright © 2026 University of Glasgow

Packet Loss in the Internet

- The Internet is an **unreliable** best effort packet delivery network
  - IP packets may be lost, delayed, reordered, or corrupted in transit
  - How often this happens varies significantly
    - Wireless links are less reliable than wired links
    - Countries with well-developed infrastructure tend to have reliable Internet links; countries with less robust or lower capacity infrastructure see more issues
  - Some **packet loss is inevitable**
- The **transport layer** must adapt the quality of service provided by the network to match application needs
  - TCP and QUIC intentionally try to push links to capacity, causing temporary overload as they try to find the limit



Colin Perkins | https://csperkins.org/ | Copyright © 2026 University of Glasgow

The End-to-End Argument

- Is it better to place functionality within the network or at the end points?
  - Only put functionality that is absolutely necessary in the network, leave everything else to end systems
    - **Example**: let the network provide best effort packet delivery, rather than try to detect and retransmit lost packets
    - If the network is not guaranteed to be **100%** reliable, always, end systems must check for lost packets anyway
    - Since 100% reliability can never be guaranteed, no point in complicating the network trying to make it reliable
- One of the defining principles of the Internet



End-To-End Arguments in System Design

J. H. SALTZER, D. P. REED, and D. D. CLARK
Massachusetts Institute of Technology Laboratory for Computer Science

This paper presents a design principle that helps guide placement of functions among the modules of a distributed computer system. The principle, called the end-to-end argument, suggests that functions placed at low levels of a system may be redundant or of little value when compared with the cost of providing them at that low level. Examples discussed in the paper include bit-error recovery, security using encryption, duplicate message suppression, recovery from system crashes, and delivery acknowledgment. Low-level mechanisms to support these functions are justified only as performance enhancements.

CR Categories and Subject Descriptors: C.0 [General] Computer System Organization—*system architectures*; C.2.2 [Computer-Communication Networks]: Network Protocols—*protocol architecture*; C.2.4 [Computer-Communication Networks]: Distributed Systems; D.4.7 [Operating Systems]: Organization and Design—*distributed systems*

General Terms: Design

Additional Key Words and Phrases: Data communication, protocol design, design principles

1. INTRODUCTION

Choosing the proper boundaries between functions is perhaps the primary activity of the computer system designer. Design principles that provide guidance in this choice of function placement are among the most important tools of a system designer. This paper discusses one class of function placement argument that has been used for many years with neither explicit recognition nor much conviction. However, the emergence of the data communication network as a computer system component has sharpened this line of function placement argument by making more apparent the situations in which and the reasons why it applies. This paper articulates the argument explicitly, so as to examine its nature and to see how general it really is. The argument appeals to application requirements and provides a rationale for moving a function upward in a layered system closer to the application that uses the function. We begin by considering the communication network version of the argument.

This is a revised version of a paper adapted from End-to-End Arguments in System Design by J. H. Saltzer, D.P. Reed, and D.D. Clark from the 2nd International Conference on Distributed Systems (Paris, France, April 8–10) 1981, pp. 509–512. © IEEE 1981
This research was supported in part by the Advanced Research Projects Agency of the U.S. Department of Defense and monitored by the Office of Naval Research under contract N00014-75-C-0661.
Authors' address: J. H. Saltzer and D. D. Clark, M.I.T. Laboratory for Computer Science, 545 Technology Square, Cambridge, MA 02139. D. P. Reed, Software Arts, Inc., 27 Mica Lane, Wellesley, MA 02181.
Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the ACM copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Association for Computing Machinery. To copy otherwise, or to republish, requires a fee and/or specific permission.
© 1984 ACM 0734-2071/84/1100-0277 $00.75
ACM Transactions on Computer Systems, Vol. 2, No. 4, November 1984, Pages 277–288.

J. Saltzer, D. Reed, and D. Clark, ” End-to-end arguments in system design”, ACM Transactions on Computer Systems, November 1984. http://dx.doi.org/10.1145/357401.357402

Colin Perkins | https://csperkins.org/ | Copyright © 2026 University of Glasgow

Timeliness vs Reliability Trade-off (1/2)



- Repairing or retransmitting lost packets takes time
- Fundamental trade-off:
  - If a connection is to be reliable, it **cannot** guarantee timeliness
  - If a connection is to be timely, it **cannot** guarantee reliability

Colin Perkins | https://csperkins.org/ | Copyright © 2026 University of Glasgow

Timeliness vs Reliability Trade-off (2/2)



- Different applications make different timeliness vs. reliability trade-offs:
  - Web, email, etc. → data must be delivered in order sent; no strong timeliness requirement
  - Telephony and video conferencing → tolerates some data loss, but requires timeliness
- Implication for network architecture:
  - Network layer should provide a timely but unreliable service
  - Transport layer protocols can add reliability, if needed

Colin Perkins | https://csperkins.org/ | Copyright © 2026 University of Glasgow

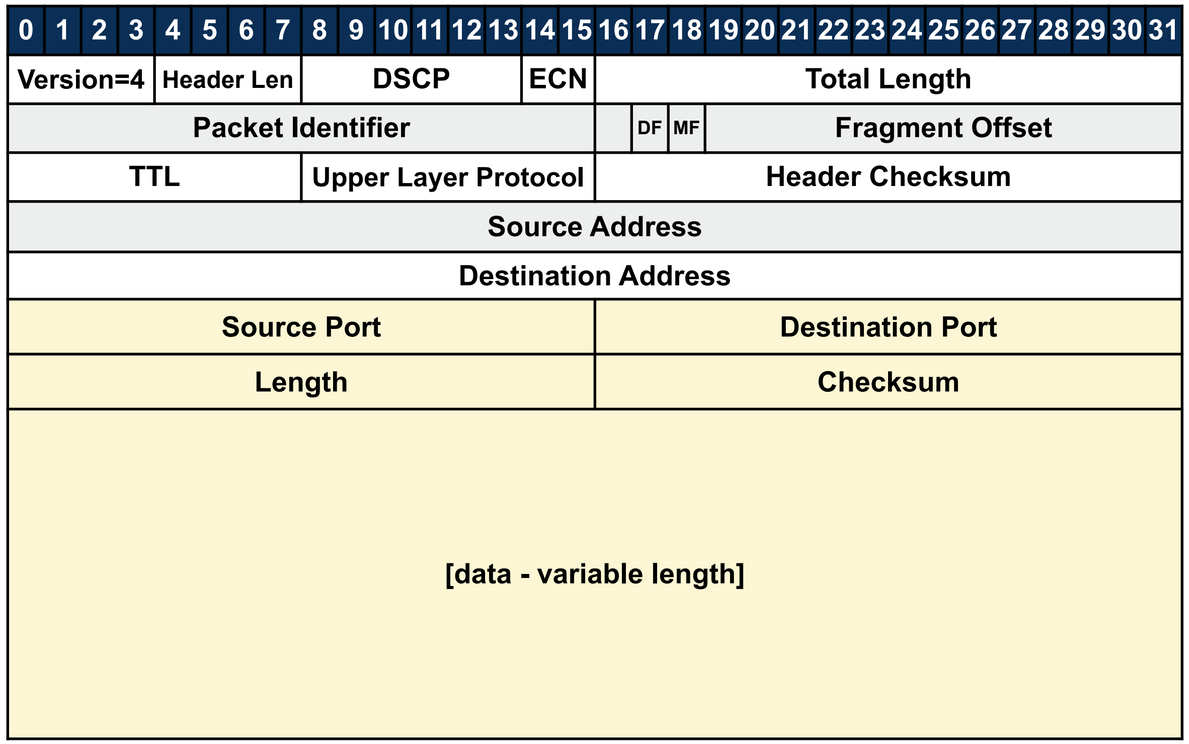
University of Glasgow

Unreliable Data Using UDP

- UDP service model
- Sending and receiving data
- Higher-layer protocols

CC BY ND

Packet Loss and UDP

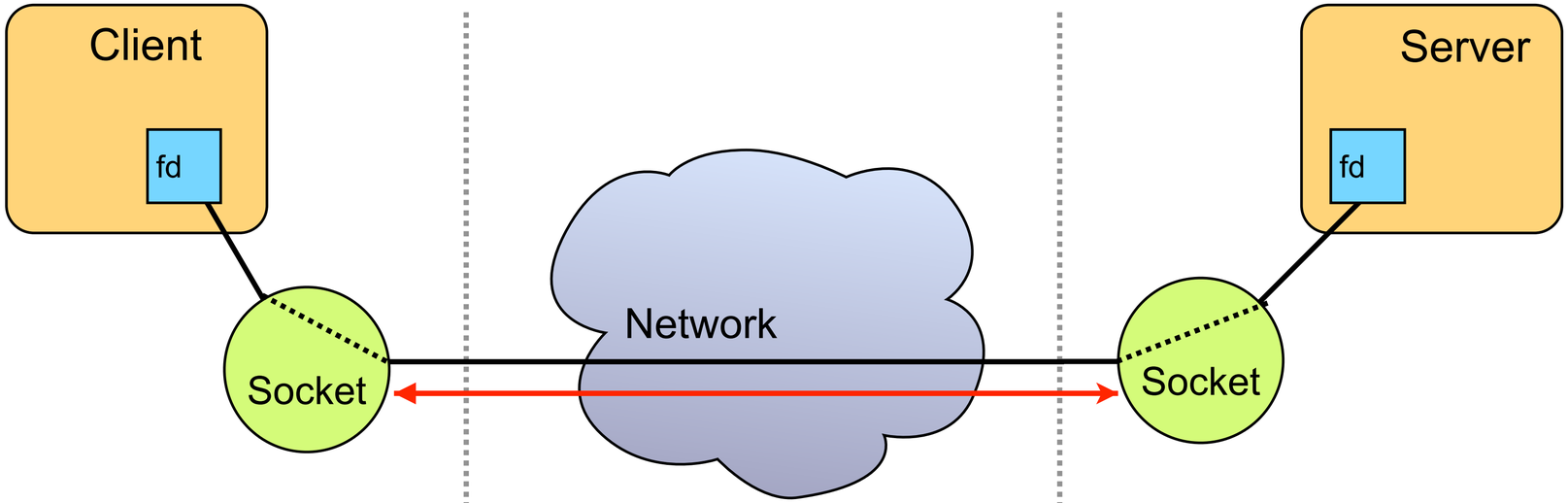


- UDP → **unreliable**, connectionless, datagram service
  - Adds port numbers and a checksum to IP
  - Does not provide reliability or congestion control; only **best effort packet delivery**
- A **substrate for building new transport protocols**
  - QUIC → Lecture 4
  - RTP → Lecture 7
  - DNS → Lecture 8

Well-known UDP port numbers: https://www.iana.org/assignments/service-names-port-numbers/service-names-port-numbers.xhtml

Colin Perkins | https://csperkins.org/ | Copyright © 2026 University of Glasgow

Using UDP Datagrams




```
int fd = socket(...)

bind(fd, ..., ...)

sendto(fd, data, datalen, addr, addrlen)

recvfrom(fd, buffer, buflen, flags, addr, addrlen)

close(fd)
```

Colin Perkins | https://csperkins.org/ | Copyright © 2026 University of Glasgow

Sending UDP Datagrams

- The **`sendto()`** call sends a single datagram
- Each call to **`sendto()`** can send to a different address, even though they use the same socket.

```
int                 fd;
char                buffer[...];
int                 buflen = sizeof(buffer);
struct sockaddr_in  addr;
...
if (sendto(fd, buffer, buflen, (struct sockaddr *) addr, sizeof(addr)) < 0) {
    // Error...
}
```

- Alternatively, **`connect()`** to an address, then use **`send()`** to send the data
  - There is no connection made by the UDP layer, a **`connect()`** call only sets the destination address for future packets.

Colin Perkins | https://csperkins.org/ | Copyright © 2026 University of Glasgow

Receiving UDP Datagrams

- The **`recv()`** call may be used to read a single datagram, but doesn't provide the source address of the datagram.
- Most code uses **`recvfrom()`** instead – this fills in the source address of the received datagram:

```
int                 fd;
char                buffer[...];
int                 buflen = sizeof(buffer);
struct sockaddr     addr;
socklen_t           addr_len = sizeof(addr);
int                 rlen;
...
rlen = recvfrom(fd, buffer, buflen, 0, &addr, &addrlen);
if (rlen < 0) {
    // Error...
}
```

- Packets can be lost, delayed, or reordered in transit – UDP **does not** attempt to recover from this

Colin Perkins | https://csperkins.org/ | Copyright © 2026 University of Glasgow

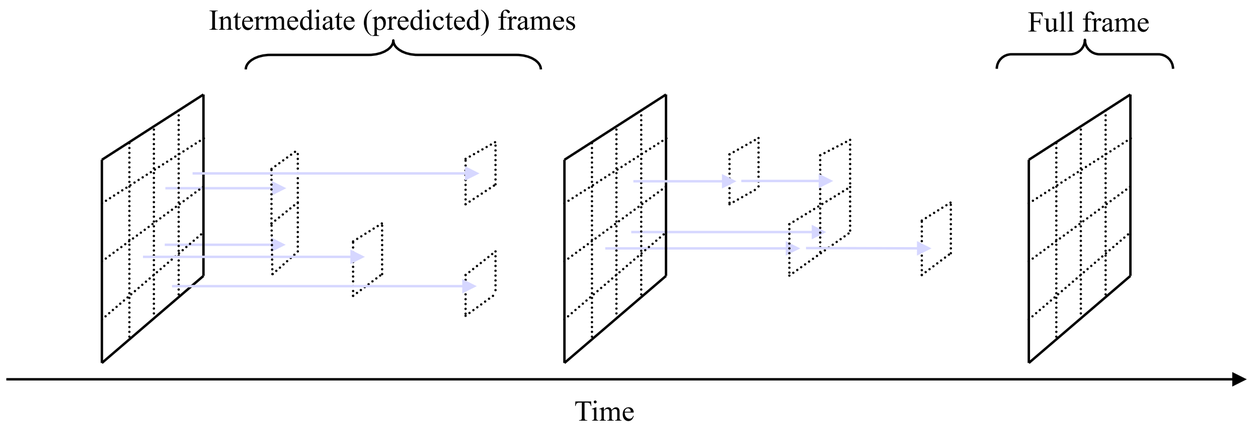
UDP Framing and Reliability (1/2)

- Each **`sendto()`** calls generates exactly one UDP datagram
  - Each **`recvfrom()`** corresponds to a single **`sendto()`**
- But, UDP is unreliable:
  - Datagrams may be lost, delayed, reordered, or duplicated in transit
  - Data sent with **`sendto()`** might never arrive, might arrive more than once, or data sent in consecutive calls to **`sendto()`** might arrive out-of-order
  - UDP **does not** attempt to correct this
- The **protocol built on UDP** is responsible for correcting the order, detecting duplicates, and repairing loss – if necessary
  - The protocol running within UDP generally includes a sequence number to support this
  - e.g., RTP and QUIC both include a packet sequence number inside UDP packets

Colin Perkins | https://csperkins.org/ | Copyright © 2026 University of Glasgow

UDP Framing and Reliability (2/2)

- UDP provides framing – data is delivered a packet at a time – but is unreliable
- Application must organise the data so it's useful if some datagrams lost
  - e.g. video conferencing with I- and P-frames



See also: D. D. Clark and D. L. Tennenhouse, "Architectural considerations for a new generation of protocols", ACM SIGCOMM Conference, Philadelphia, PA, USA, September 1990.http://dx.doi.org/10.1145/99517.99553

Colin Perkins | https://csperkins.org/ | Copyright © 2026 University of Glasgow

UDP Sequencing and Reliability

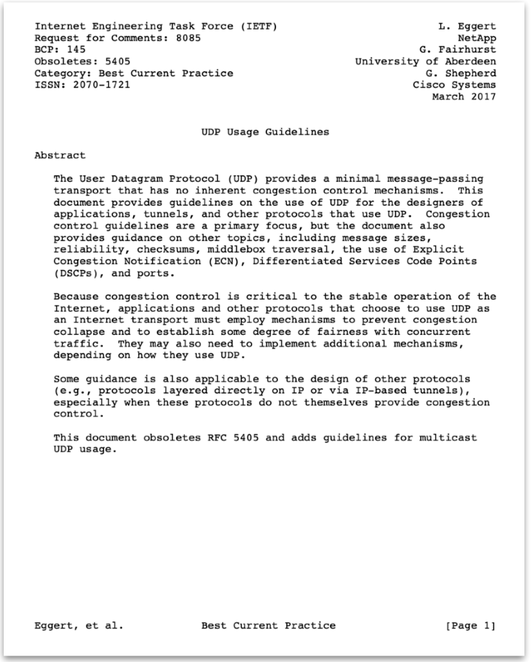
- UDP does not attempt to provide sequencing, reliability, or timing recovery
  - It provides a best-effort datagram delivery service, and identifies applications according to port numbers – substrate on which application protocols can be implemented
  - The syntax and semantics of the datagrams depends on the application layer protocol running within UDP
    - This might provide sequence numbers and acknowledgements
    - This might provide retransmission and/or forward error correction
    - This might provide timestamps to recover timing
    - This might provide payload identification, or other information

Colin Perkins | https://csperkins.org/ | Copyright © 2026 University of Glasgow

Guidelines for writing UDP applications

- IETF provides guidelines for writing UDP-based applications
  - RFC 8085 – https://datatracker.ietf.org/doc/rfc8085/

- **Read this before trying to write UDP-based applications**



Internet Engineering Task Force (IETF) — L. Eggert, NetApp
Request for Comments: 8085 — G. Fairhurst, University of Aberdeen
BCP: 145 — G. Shepherd, Cisco Systems
Obsoletes: 5405 — March 2017
Category: Best Current Practice
ISSN: 2070-1721

UDP Usage Guidelines

Abstract

The User Datagram Protocol (UDP) provides a minimal message-passing transport that has no inherent congestion control mechanisms. This document provides guidelines on the use of UDP for the designers of applications, tunnels, and other protocols that use UDP. Congestion control guidelines are a primary focus, but the document also provides guidance on other topics, including message sizes, reliability, checksums, middlebox traversal, the use of Explicit Congestion Notification (ECN), Differentiated Services Code Points (DSCPs), and ports.

Because congestion control is critical to the stable operation of the Internet, applications and other protocols that choose to use UDP as an Internet transport must employ mechanisms to prevent congestion collapse and to establish some degree of fairness with concurrent traffic. They may also need to implement additional mechanisms, depending on how they use UDP.

Some guidance is also applicable to the design of other protocols (e.g., protocols layered directly on IP or via IP-based tunnels), especially when these protocols do not themselves provide congestion control.

This document obsoletes RFC 5405 and adds guidelines for multicast UDP usage.

Eggert, et al. — Best Current Practice — [Page 1]

Colin Perkins | https://csperkins.org/ | Copyright © 2026 University of Glasgow

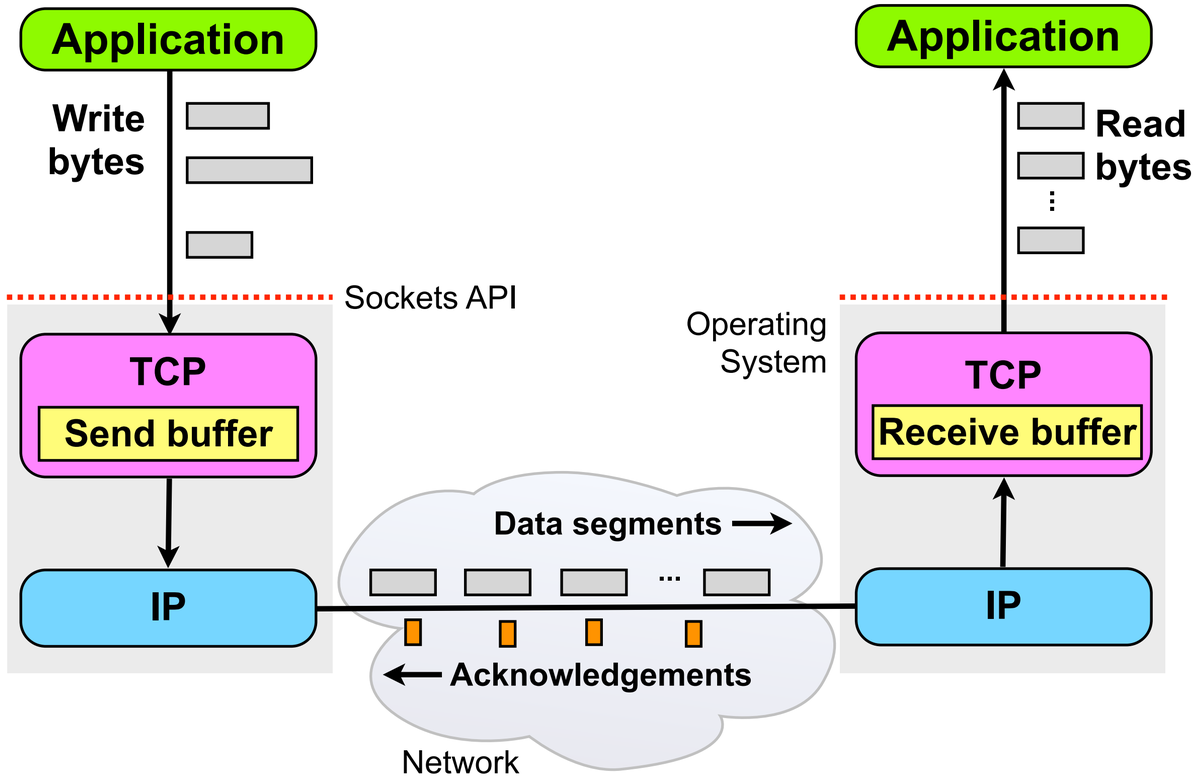
University of Glasgow

Reliable Data With TCP

- TCP service model
- Sequence numbers and ACKs
- Packet loss detection and recovery

CC BY ND

TCP Service Model

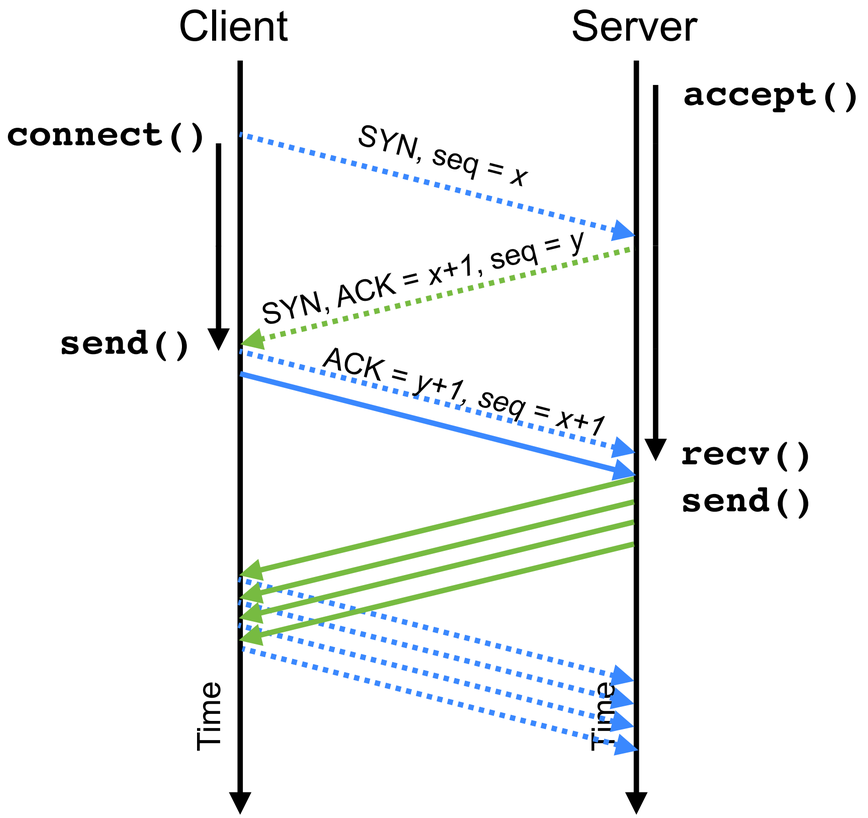


- Reliable, ordered, byte stream delivery service running over IP

Colin Perkins | https://csperkins.org/ | Copyright © 2026 University of Glasgow

Reliable Data Transfer with TCP

- TCP delivers an ordered, reliable, byte stream
- After connection established, client and server can **`send()`** or **`recv()`** data
  - Data can flow in either direction within a TCP connection
  - No requirement that data transfers follow a request-response pattern; client and server can send in any order
- TCP ensures data delivered reliably and in order
  - TCP sends acknowledgements for segments as they are received; retransmits lost data
  - TCP will recover original transmission order if segments are delayed and arrive out of order



Colin Perkins | https://csperkins.org/ | Copyright © 2026 University of Glasgow

Reading and Writing Data with TCP (1/2)

```
char data[]  = “Hello, world!”;
int  datalen = strlen(data);
...
int sent = send(fd, data, datalen, 0);
if (sent == -1) {
    // Error has occurred
    ...
} else if (sent < datalen) {
    // Couldn’t send it all, retry unsent
    …
}
```

- The **`send()`** function transmits data
  - Blocks until the data can be written
    - Might not be able to send all the data, if the connection is congested
  - Returns actual amount of data sent
    - Returns -1 if error occurs, sets **`errno`**

Colin Perkins | https://csperkins.org/ | Copyright © 2026 University of Glasgow

Reading and Writing Data with TCP (2/2)

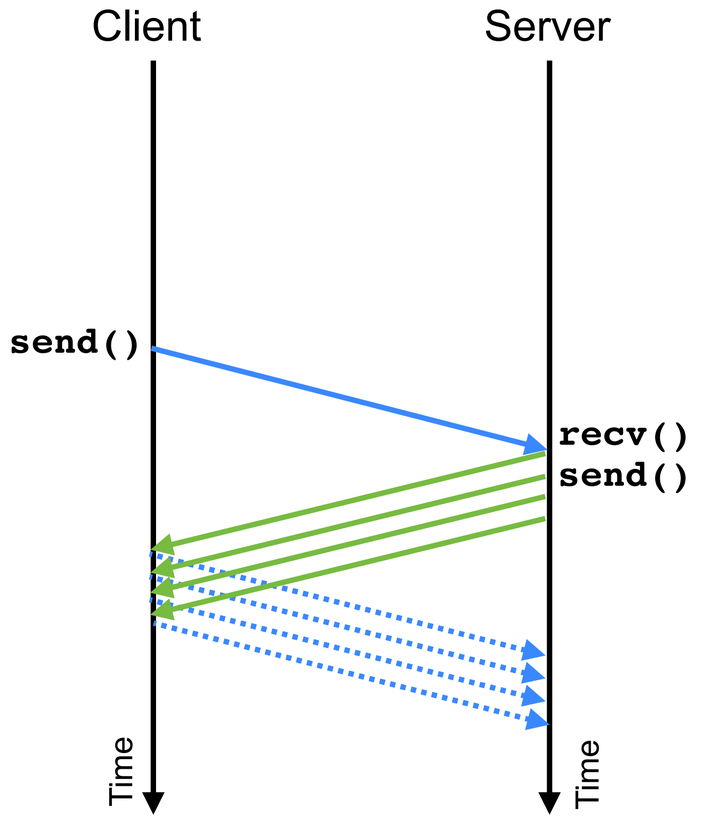
- The **`recv()`** function receives data
  - Blocks until data available or connection closed; reads up to BUFLEN bytes
  - Returns number of bytes read
    - If connection closed, returns 0
    - If error occurs, returns -1 and sets **`errno`**
  - Received data is **not** null terminated – **this is a significant security risk**

```
#define BUFLEN 1500
...
ssize_t i;
ssize_t rcount;
char    buf[BUFLEN];
...
rcount = recv(fd, buf, BUFLEN, 0);
if (rcount == 0) {
    // Sender closed the connection
    ...
} else if (rcount == -1) {
    // Error has occurred
    ...
} else {
    // Successfully read rcount bytes
    for (i = 0; i < rcount; i++) {
        printf("%c", buf[i]);
    }
    printf("\n");
}
```

Colin Perkins | https://csperkins.org/ | Copyright © 2026 University of Glasgow

TCP Segments and Sequence Numbers (1/2)

- The **`send()`** call enqueues data for transmission
- Data is split into **segments**, each placed in a **TCP packet**,
- TCP packets sent in IP packets, when allowed by congestion control algorithm
  - Each segment has a sequence number
  - Sequence numbers start from the value sent in the TCP handshake
    - Initial sequence number is sent in first packet sent from client→server (the packet that has the **SYN** bit set to 1); chosen randomly by the client
    - First data packet has sequence number one higher than the initial packet
    - Subsequent data packets increase sequence number by number of bytes sent
  - Separate sequence number spaces in each direction



Colin Perkins | https://csperkins.org/ | Copyright © 2026 University of Glasgow

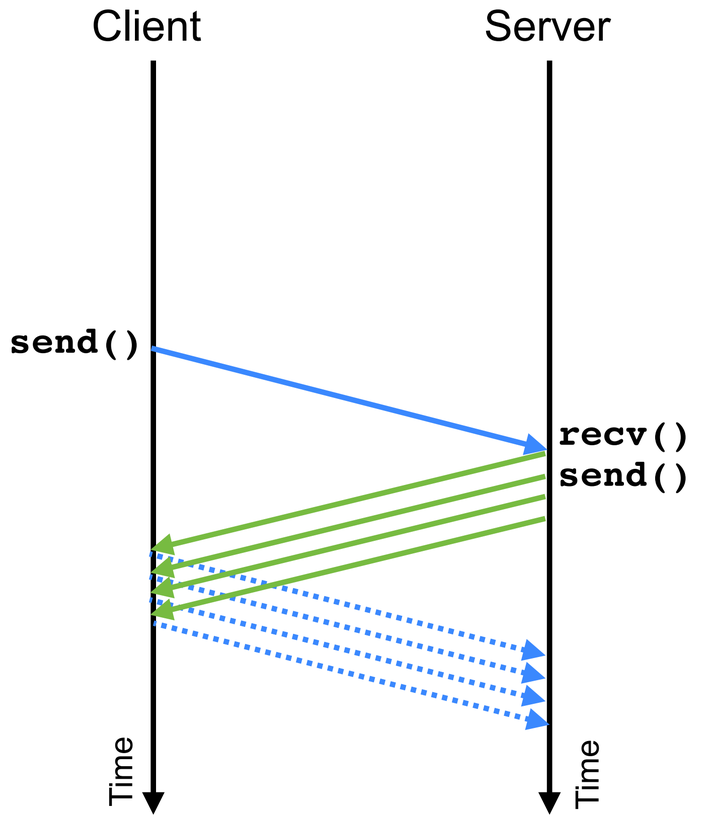
TCP Segments and Sequence Numbers (2/2)

- Calls to **`send()`** don't directly map to TCP segments
  - If data given to **`send()`** is too large to fit in one segment, TCP will split it across several segments
  - If data given to **`send()`** is small, TCP might wait to send the data, combining it with later data into a single larger segment
    - Known as **Nagle's algorithm** – improves efficiency, but adds some delay
    - Can be disabled with the **`TCP_NODELAY`** socket option:

```
int param = 1;

if (setsockopt(fd, SOL_TCP, TCP_NODELAY, &param, sizeof(param)) < 0) {
  // Error occurred
} else {
  // Successfully disabled Nagle's algorithm
}
```

- Implication: data returned by **`recv()`** does not always correspond to a single **`send()`** call



Colin Perkins | https://csperkins.org/ | Copyright © 2026 University of Glasgow

TCP Does Not Preserve Message Boundaries

HTTP Headers

```
HTTP/1.1 200 OK
Date: Mon, 19 Jan 2009 22:25:40 GMT
Server: Apache/2.0.46 (Scientific Linux)
Last-Modified: Mon, 17 Nov 2003 08:06:50 GMT
ETag: "57c0cd-e3e-17901a80"
Accept-Ranges: bytes
Content-Length: 3646
Connection: close
Content-Type: text/html; charset=UTF-8
```

Body

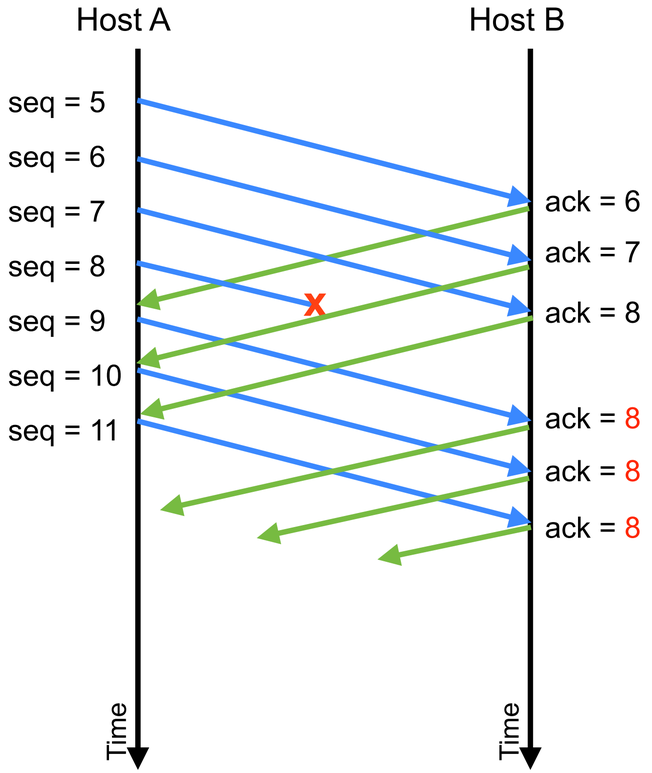
```
<HTML>
<HEAD>
<TITLE>Computing Science, University of Glasgow </TITLE>
...
</BODY>
</HTML>
```

- The recv() call returns data reliably, and in the order sent, but doesn't frame data
  - Desirable if one recv() call would fetch all the HTTP headers, then another the entire body
  - TCP not does guarantee this – might split the response arbitrarily, e.g., matching the colours
  - Complicates code using TCP
- Implication: you must parse the data to know how much remains to be read

Colin Perkins | https://csperkins.org/ | Copyright © 2026 University of Glasgow

TCP Acknowledgements

- TCP receiver acknowledges each segment received
  - Each segment has a **sequence number** and **acknowledgement number**
  - Segments sent to acknowledge each received segment – contains acknowledgment number indicating sequence number of the **next contiguous byte expected**
    - Includes data if any ready to send, or can be empty apart from acknowledgement
  - If a packet is lost, subsequent packets will generate duplicate acknowledgements
    - In example, segment with sequence number 8 is lost
    - When segment with sequence number 9 arrives, the receiver acknowledges 8 again – since that's still the next contiguous sequence number expected
  - Can send **delayed acknowledgements**
    - e.g., acknowledge every second packet if there is no data to send in reverse direction

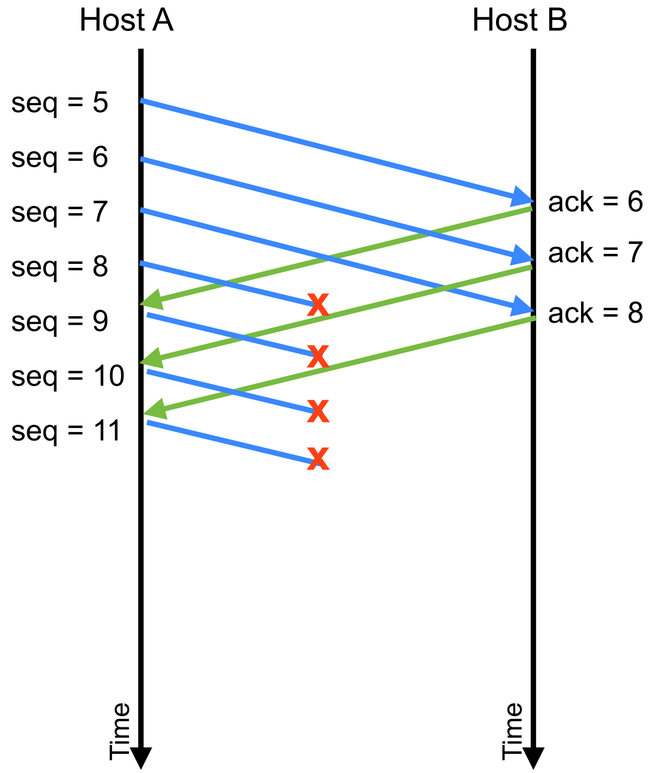


TCP connection progress, showing sequence numbers and acknowledgements; in this example, each segment carries one byte of data; the segment with sequence number 8 is lost.

Colin Perkins | https://csperkins.org/ | Copyright © 2026 University of Glasgow

TCP Acknowledgements: Loss Detection (1/3)

- TCP uses acknowledgements to detect lost segments
  - If data is being sent, but no acknowledgements return, then either receiver or the network failed
  - TCP treats **a timeout** as an indication of packet loss and retransmits unacknowledged segments

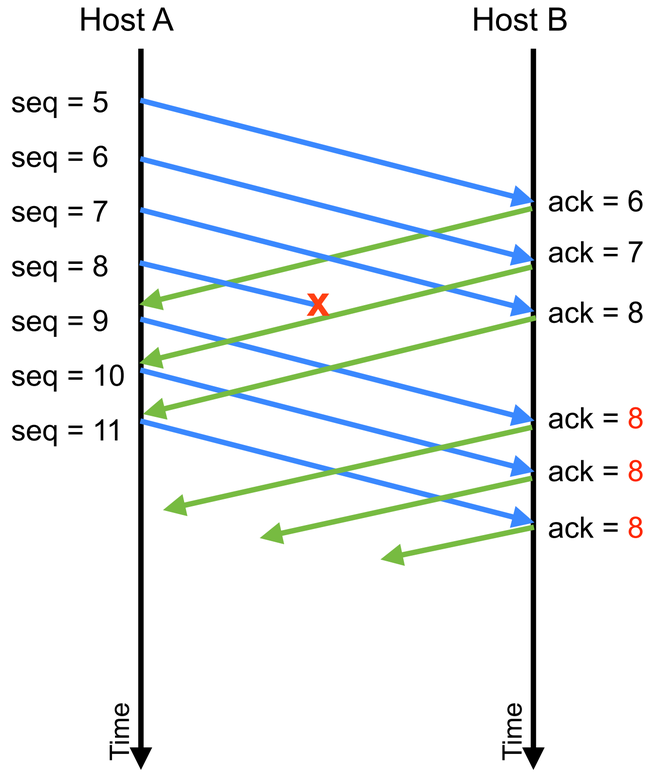


TCP connection progress, showing sequence numbers and acknowledgements; in this example, each segment carries one byte of data; the segment with sequence number 8 is lost.

Colin Perkins | https://csperkins.org/ | Copyright © 2026 University of Glasgow

TCP Acknowledgements: Loss Detection (2/3)

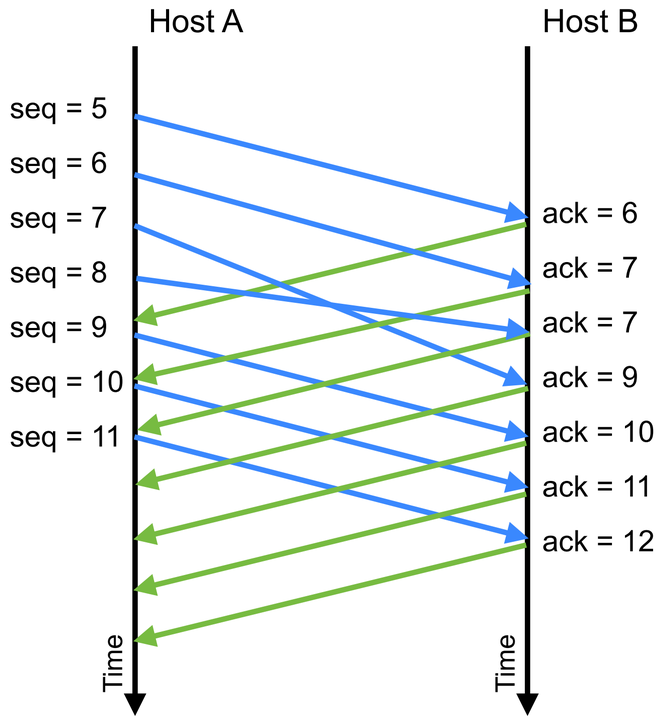
- TCP uses acknowledgements to detect lost segments
  - Duplicate acknowledgements received → some data lost, but later segments arrived
  - TCP treats a **triple duplicate acknowledgement** – four consecutive acknowledgements for the same sequence number – as an indication of packet loss
  - Lost segments are retransmitted



TCP connection progress, showing sequence numbers and acknowledgements; in this example, each segment carries one byte of data; the segment with sequence number 8 is lost.

Colin Perkins | https://csperkins.org/ | Copyright © 2026 University of Glasgow

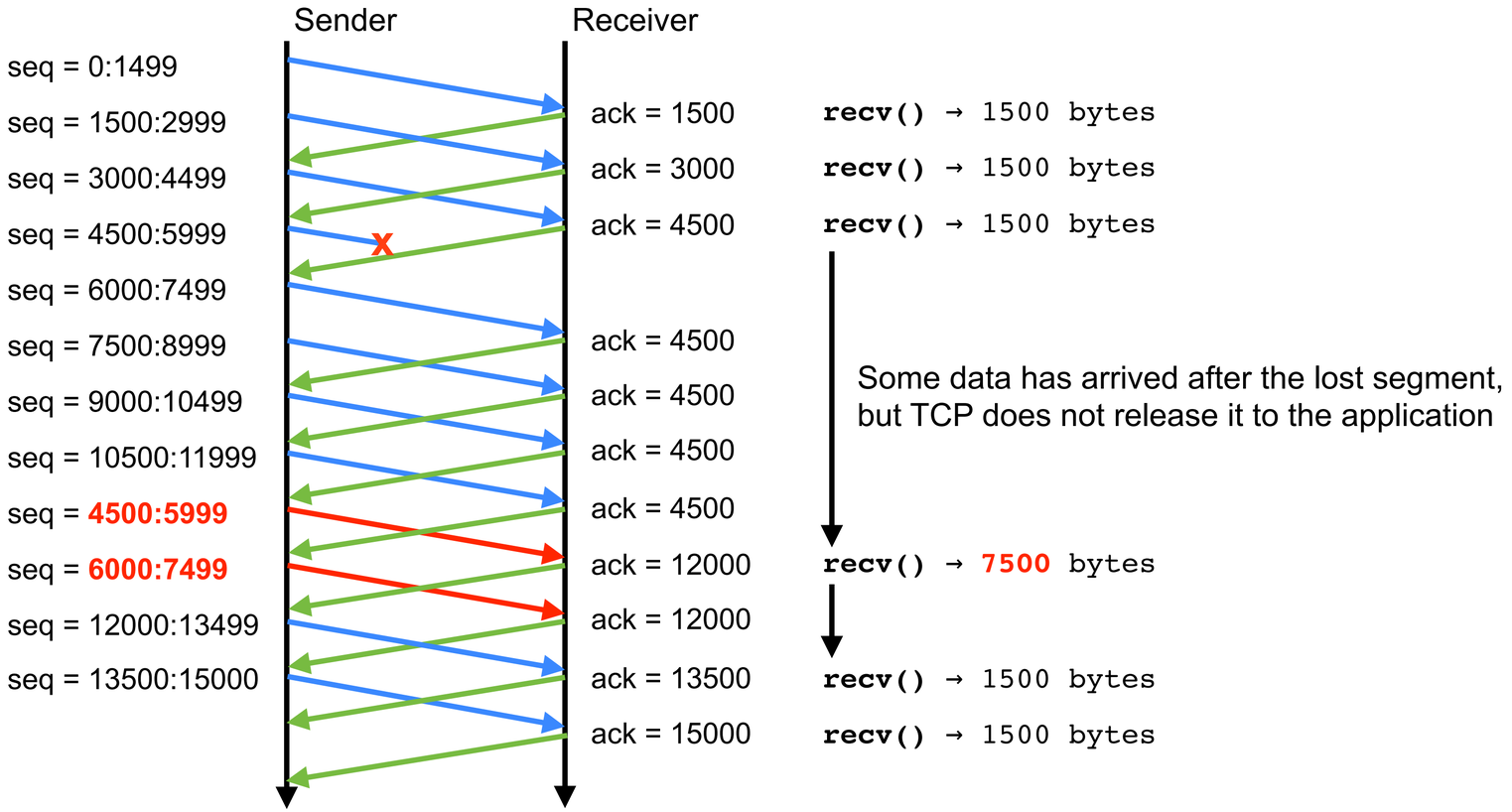
TCP Acknowledgements: Loss Detection (3/3)



- Why a **triple duplicate acknowledgement**?
  - Packet delay leading to reordering will also cause duplicate acknowledgement to be generated
  - Gives appearance of loss, when the segments are merely delayed
  - Use of triple duplicate ACK to indicate packet loss stops reordered packets from being unnecessarily retransmitted
    - Packets delayed enough to cause one or two duplicate acknowledgements is relatively common
    - Packets being delayed enough to cause three or more duplicates is rare
    - Balance speed of loss detection vs. likelihood of retransmitting a packet that was merely delayed

Colin Perkins | https://csperkins.org/ | Copyright © 2026 University of Glasgow

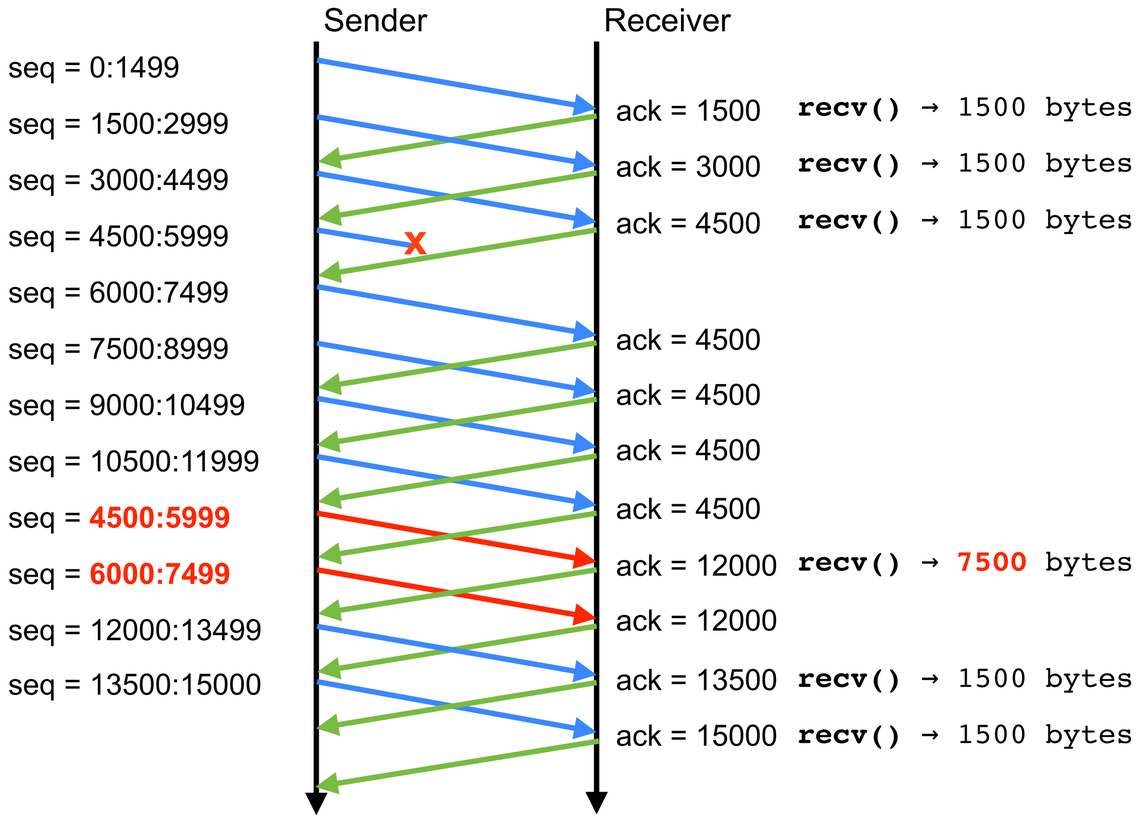
Head-of-line Blocking in TCP (1/3)

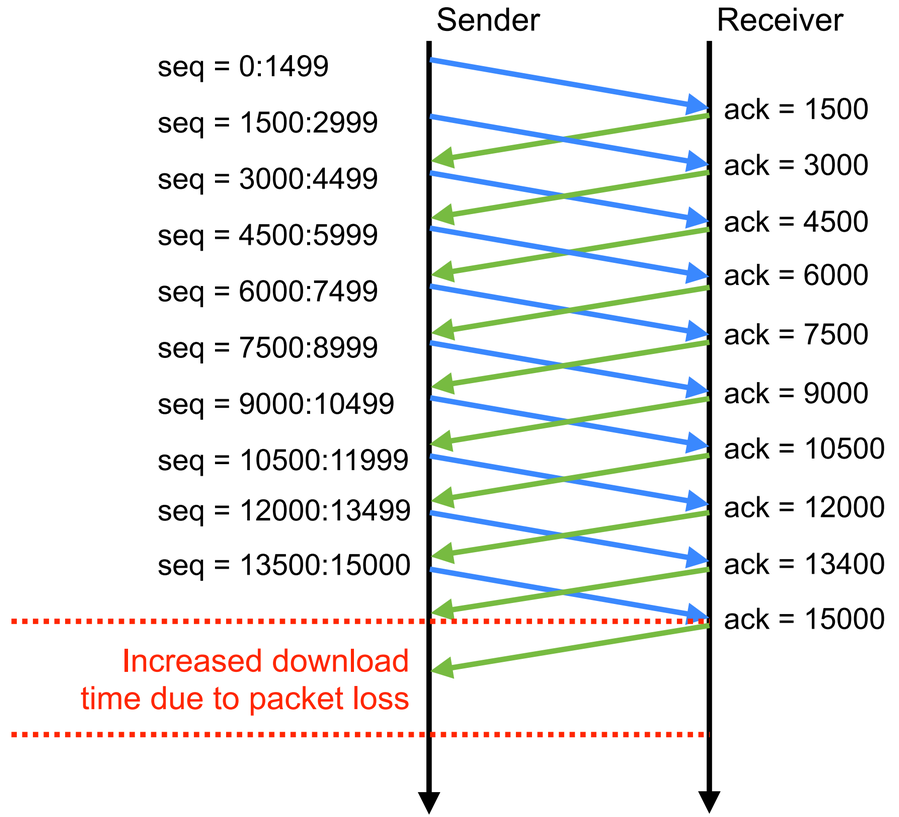


- A TCP receiver will wait for missing data to be retransmitted – calls to **`recv()`** block until missing data arrives; TCP **always** delivers data to the application in-order

Colin Perkins | https://csperkins.org/ | Copyright © 2026 University of Glasgow

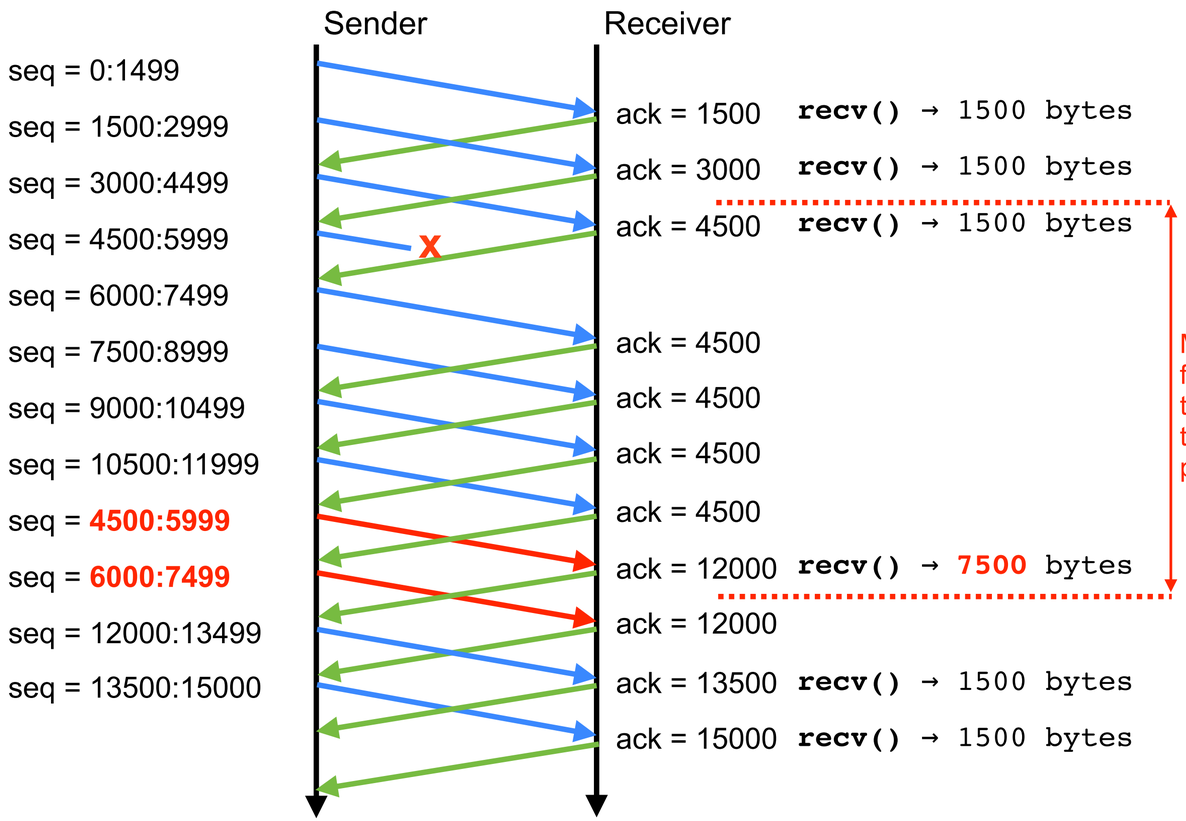
Head-of-line Blocking in TCP (2/3)

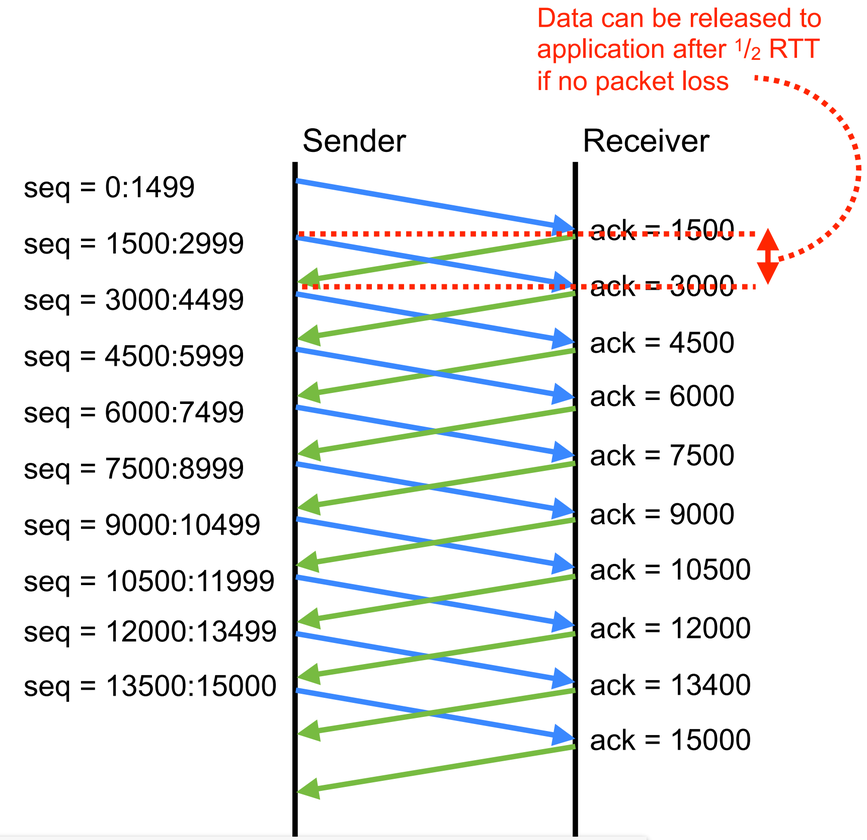




Colin Perkins | https://csperkins.org/ | Copyright © 2026 University of Glasgow

Head-of-line Blocking in TCP (3/3)





- Head-of-line blocking increases latency for progressive and real-time applications
- **Significant latency increase** if file is being played out as it downloads

Colin Perkins | https://csperkins.org/ | Copyright © 2026 University of Glasgow

Reliable Data Transfer with TCP

- **TCP provides an ordered, reliable, byte stream service**
  - Service model is simple to understand – it's like reading from a file
  - Head of line blocking can significantly impact progressive and real-time uses
    - MPEG DASH ("Dynamic Adaptive Streaming over HTTP") → Lecture 7
  - Lack of framing can complicate application design

Colin Perkins | https://csperkins.org/ | Copyright © 2026 University of Glasgow

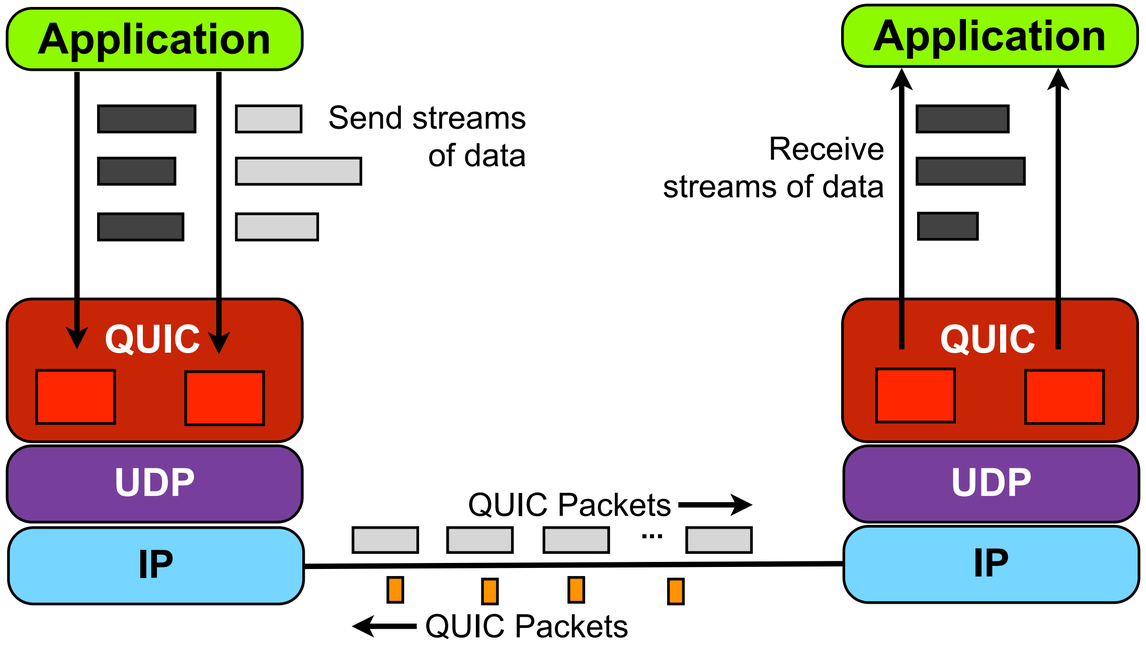
University of Glasgow

Reliable Data Transfer with QUIC

- QUIC service model
- Multi-streaming
- Limited head-of-line blocking

CC BY ND

QUIC Service Model

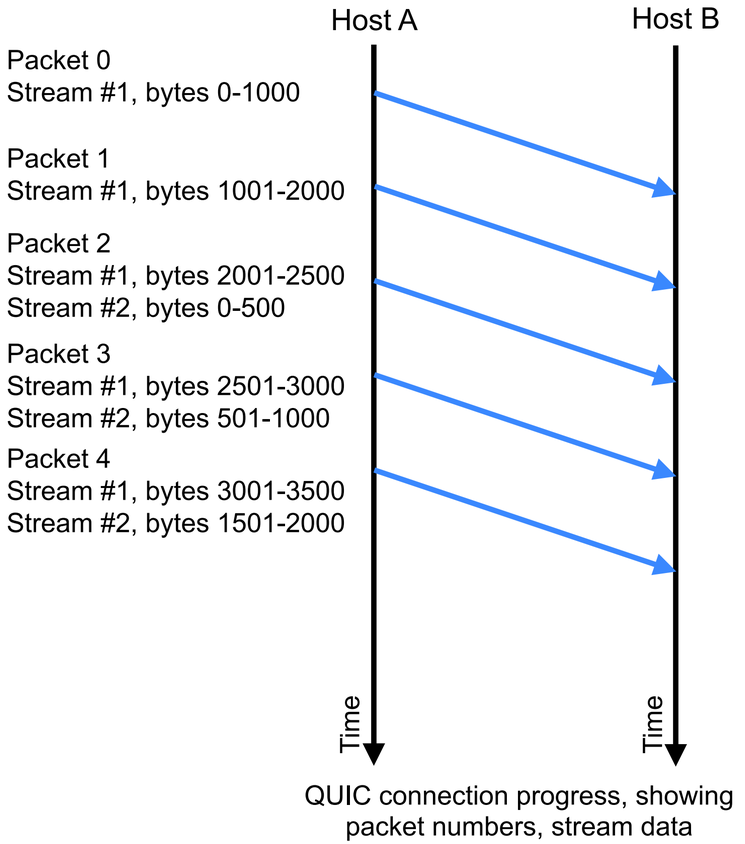


- TCP delivers reliable, ordered, byte stream
- QUIC delivers **several** ordered reliable byte streams within a single connection
  - Each stream is delivered reliably and in-order
  - Order is not preserved between streams within a QUIC connection

Colin Perkins | https://csperkins.org/ | Copyright © 2026 University of Glasgow

QUIC Packets and Sequence Numbers (1/2)

- Each QUIC packet has a **Packet Number**
  - Two packet number spaces
    - Packets sent during initial handshake
    - Packets sent to carry data
  - Within each space, packet number starts at zero, increases by one for each packet sent
    - Counts number of packets sent
    - **Different to TCP**, where the sequence number records the offset within the byte stream
- Each QUIC packet contains one or more **frames**
  - STREAM frames carry data
  - Each has a stream identifier, and carries the length of the data and its offset from start of stream – closer to TCP sequence number

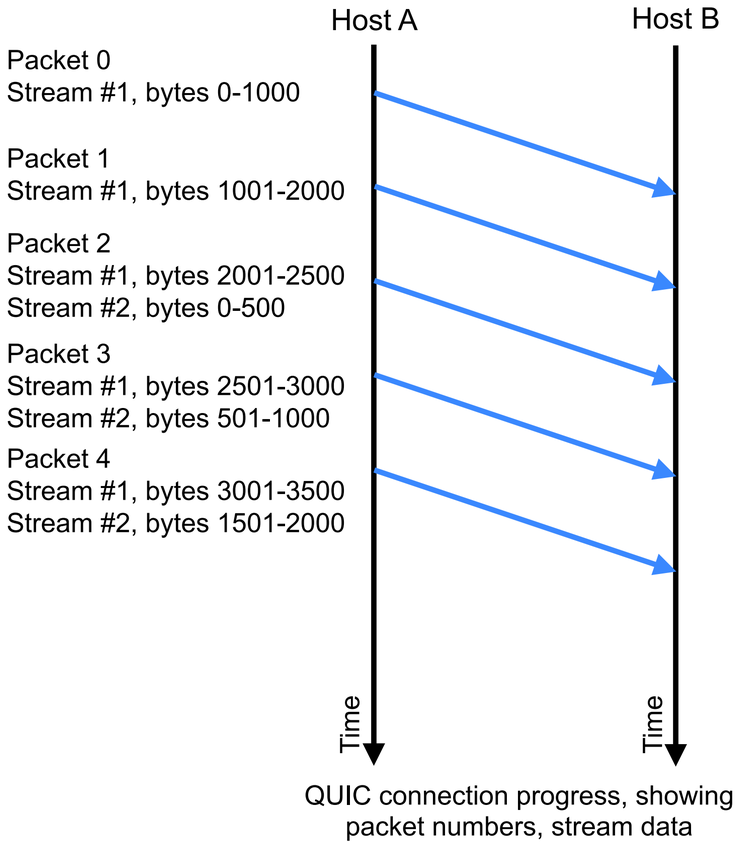


QUIC connection progress, showing packet numbers, stream data

Colin Perkins | https://csperkins.org/ | Copyright © 2026 University of Glasgow

QUIC Packets and Sequence Numbers (2/2)

- QUIC doesn’t preserve message boundaries in streams
  - If data written to stream is too big for a packet, it will be split across multiple packets
  - If data written to stream is too small for a packet, it may be delayed and sent with other data to fill complete packet
  - Data from more than one stream might be sent in a single packet
  - If >1 stream has data to send, the QUIC sender can choose how to prioritise and deliver frames from each stream

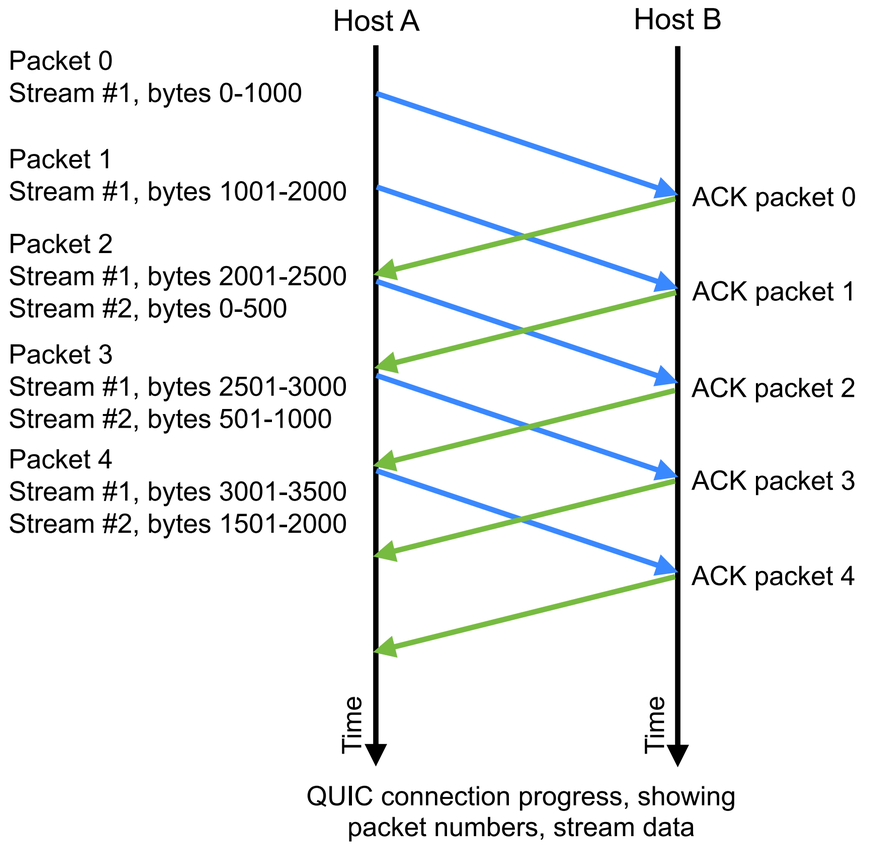


QUIC connection progress, showing packet numbers, stream data

Colin Perkins | https://csperkins.org/ | Copyright © 2026 University of Glasgow

QUIC Acknowledgements (1/2)

- A QUIC receiver acknowledges the **packets** it receives
  - Unlike TCP, it **does not** acknowledge sequence numbers within each stream
  - The sender needs to remember what data from each stream was in each packet, to know what parts of each stream have been received



QUIC connection progress, showing packet numbers, stream data

Colin Perkins | https://csperkins.org/ | Copyright © 2026 University of Glasgow

QUIC Acknowledgements (2/2)

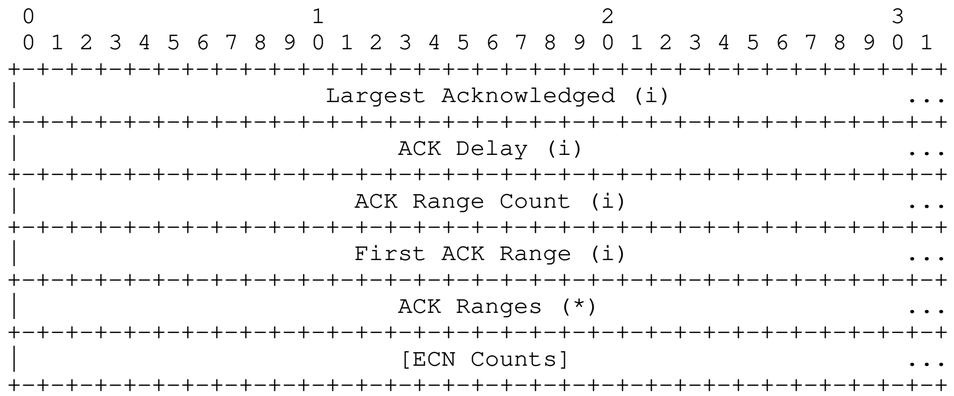
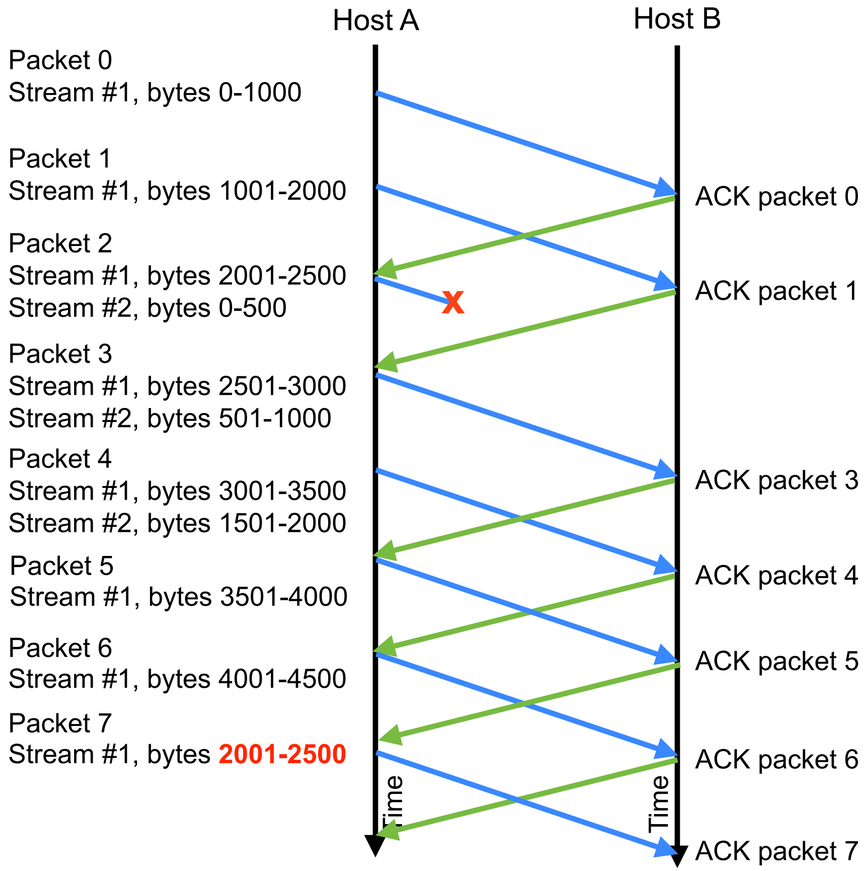


Figure 19: ACK Frame Format

- Acknowledgements sent as reverse path frames
- Acknowledgements can be delayed
  - e.g., send acknowledgement after every second packet is received
  - The **`ACK Delay`** field measures this delay to allow accurate network RTT calculation
- Acknowledgements contain ranges
  - e.g., a single ACK Frame can indicate that packets 1-4 and 6-8 were received
  - Like TCP Selective Acknowledgement extension

Colin Perkins | https://csperkins.org/ | Copyright © 2026 University of Glasgow

QUIC Retransmissions and Loss Recovery



QUIC connection progress, showing packet numbers, stream data, and acknowledgements

- QUIC does **not** retransmit lost packets
  - Every packet has a unique packet number and is only transmitted once
  - Packets declared lost when at least three later packets have been acknowledged – equivalent to TCP triple duplicate ACK
  - Packets declared lost after timeout – like TCP
- **QUIC retransmits lost data in new packets**
  - Packetisation of STREAM frames into packets may differ when data retransmitted
  - Data ordering between streams not preserved

Colin Perkins | https://csperkins.org/ | Copyright © 2026 University of Glasgow

Head-of-Line Blocking in QUIC

- QUIC can deliver several streams within a single connection
  - Data for each stream is delivered reliability and in-order
    - Whether packet loss affects one or more streams depends how the sender chooses to distribute stream data between packets
    - Depending on loss patterns, each stream can independently suffer head-of-line blocking – this affects performance **of that stream** in the same way head-of-line blocking affects TCP
  - Order is not preserved between streams within a QUIC connection
    - One stream might be blocked waiting for a retransmission, while other streams continue delivering data
    - Data from different streams is sent and received independently → careful use of streams can limit duration of head-of-line blocking

Colin Perkins | https://csperkins.org/ | Copyright © 2026 University of Glasgow

Reliable Data Transfer with QUIC

- QUIC delivers **several** ordered reliable byte streams within a single connection
  - Can be treated like several parallel TCP connections → instead of opening several TCP connections to a server, open one QUIC connection and send multiple streams within it
  - Can be treated as a framing device → each stream represents a single object, with end-of-stream signalling the object is complete
  - Order is not preserved between streams within a QUIC connection

- Best practices for use of multiple streams are still evolving

Colin Perkins | https://csperkins.org/ | Copyright © 2026 University of Glasgow

University of Glasgow

Summary

- Internet → best effort packet delivery
- Data transfer in Internet transports
  - UDP – unreliable but timely; substrate
  - TCP – reliable, ordered, stream
  - QUIC – many reliable, ordered, streams
- Different services for different needs

CC BY ND