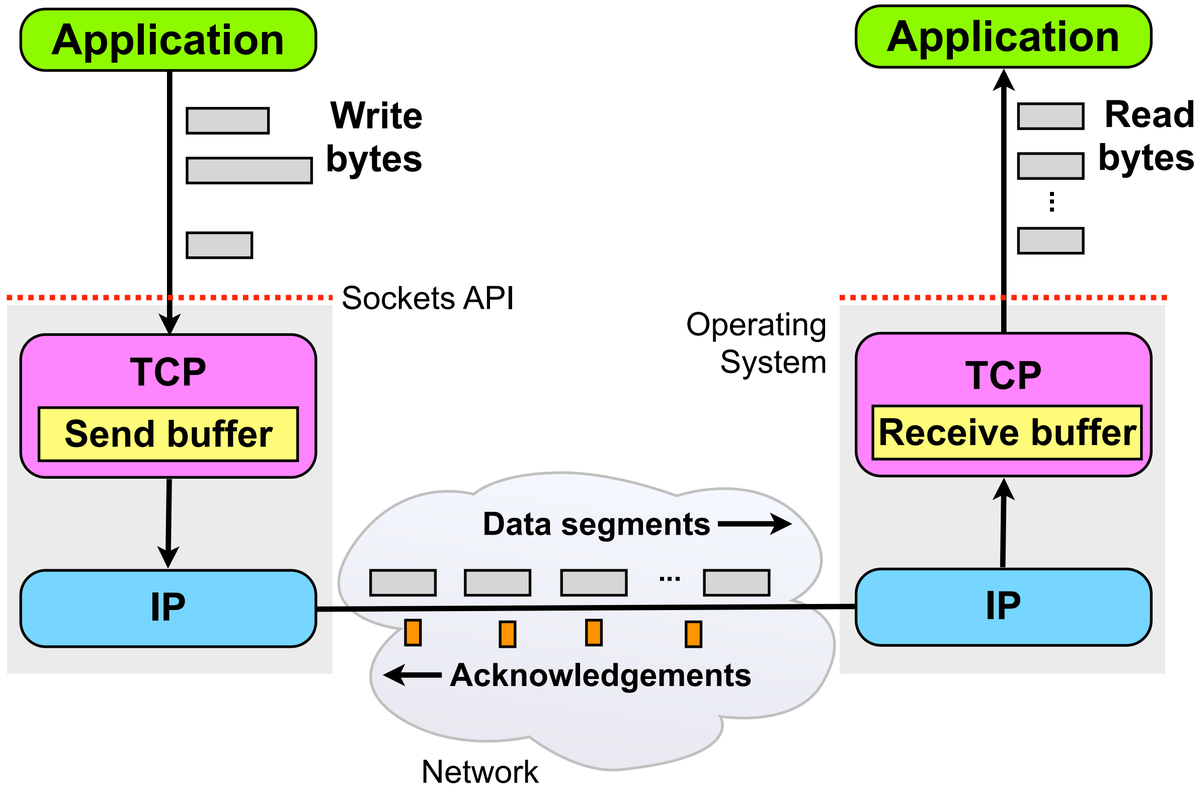
Reliable Data With TCP

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

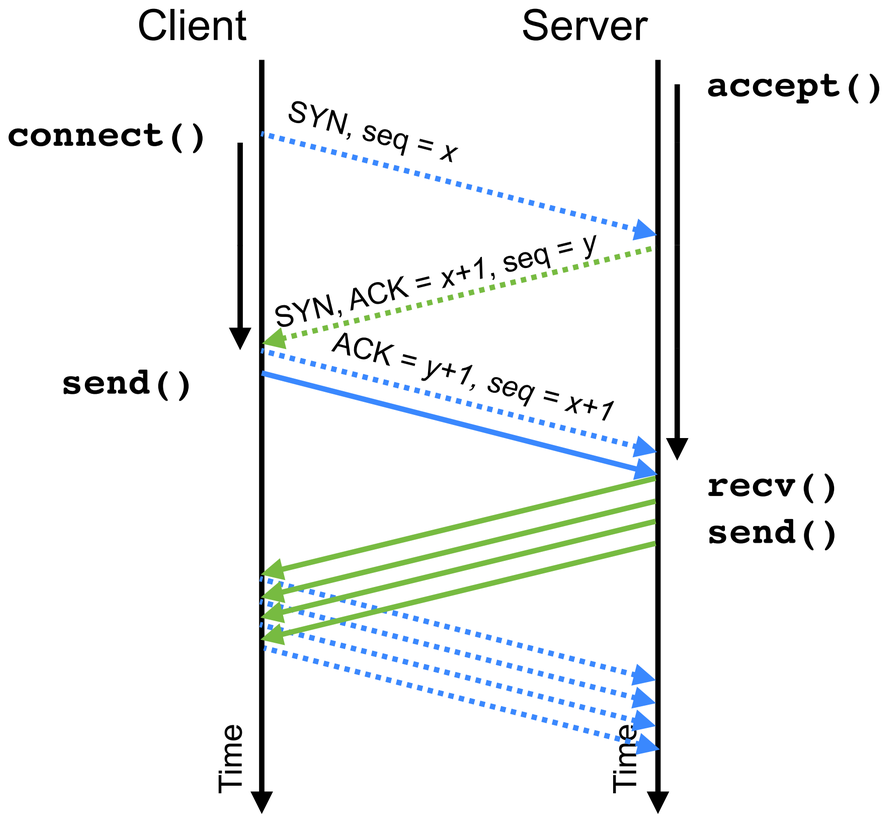
TCP Service Model



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

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



Reading and Writing Data on a TCP Connection (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`**

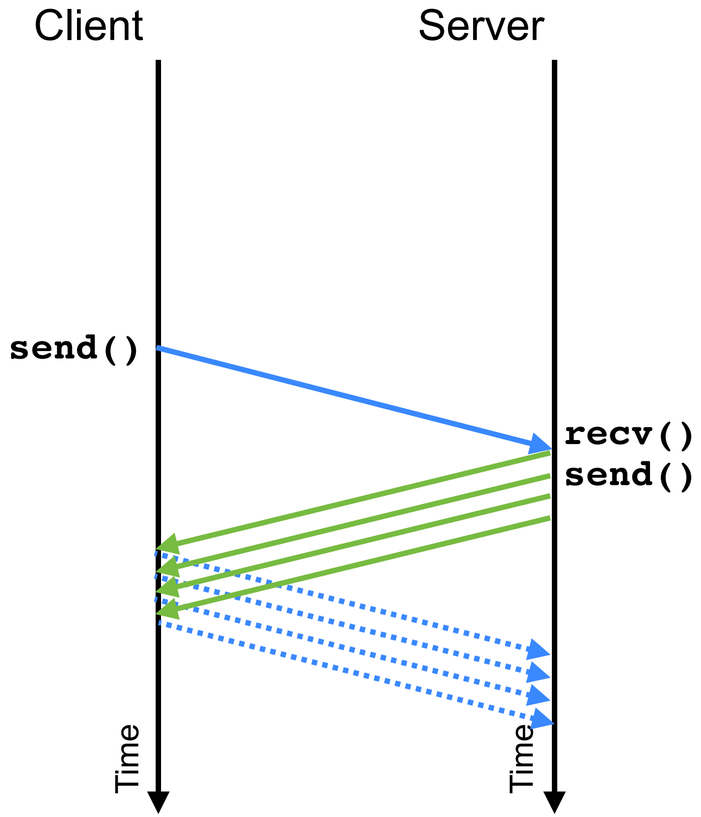
Reading and Writing Data on a TCP Connection (2/2)

- 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");
}
```

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



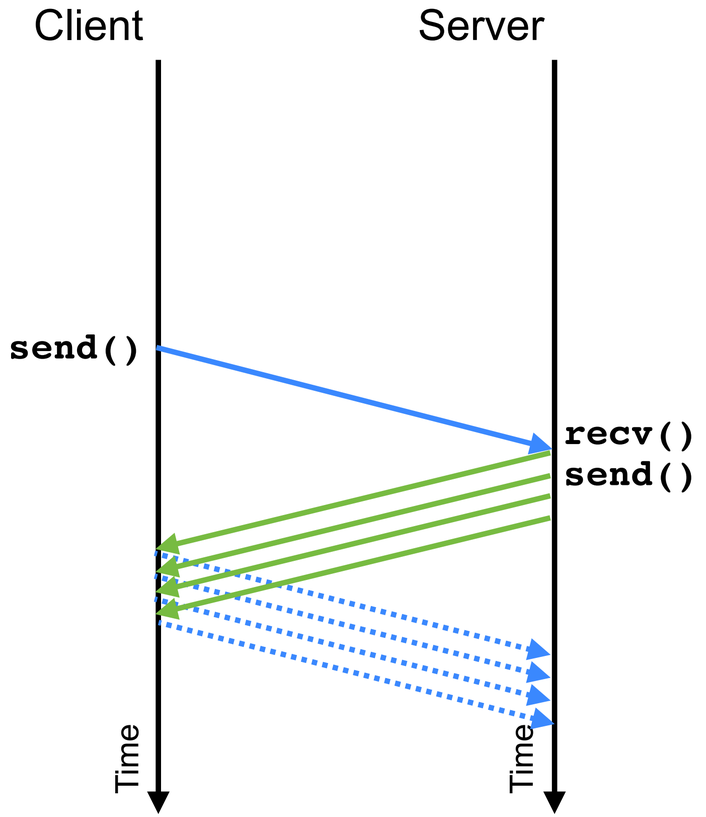
TCP Segments and Sequence Numbers (2/2)

- 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



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
```

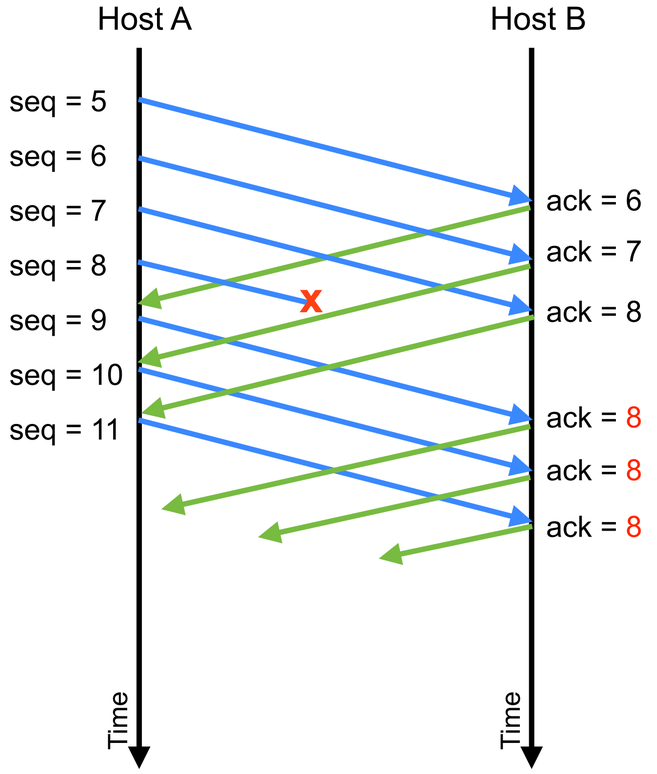
Body

```
<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

TCP Acknowledgements

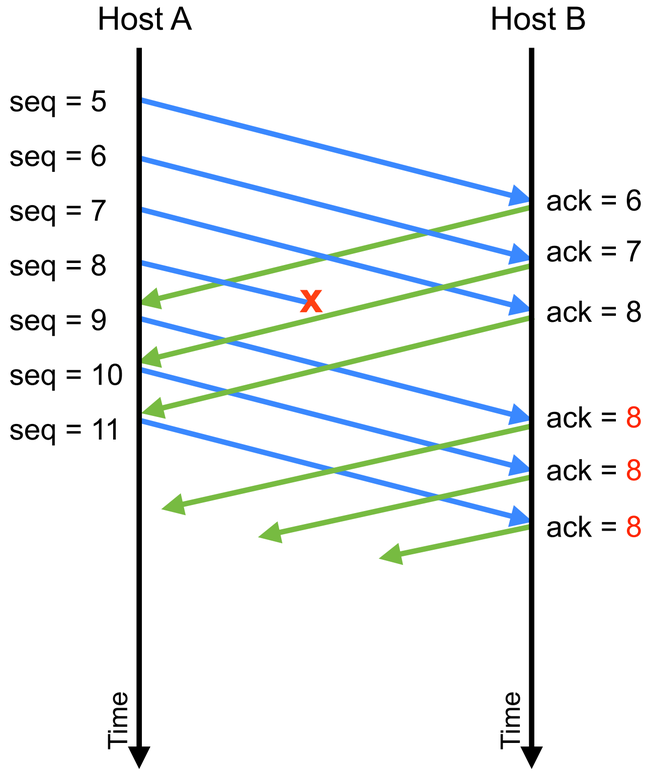
- TCP receiver acknowledges each segment received
  - Each TCP segment has **sequence number** and **acknowledgement number**
  - Segments sent to acknowledge each received segment – contains acknowledgment number indicating sequence number of the **next contiguous byte expected**
    - Includes data 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.

TCP Acknowledgements: Loss Detection (1/3)

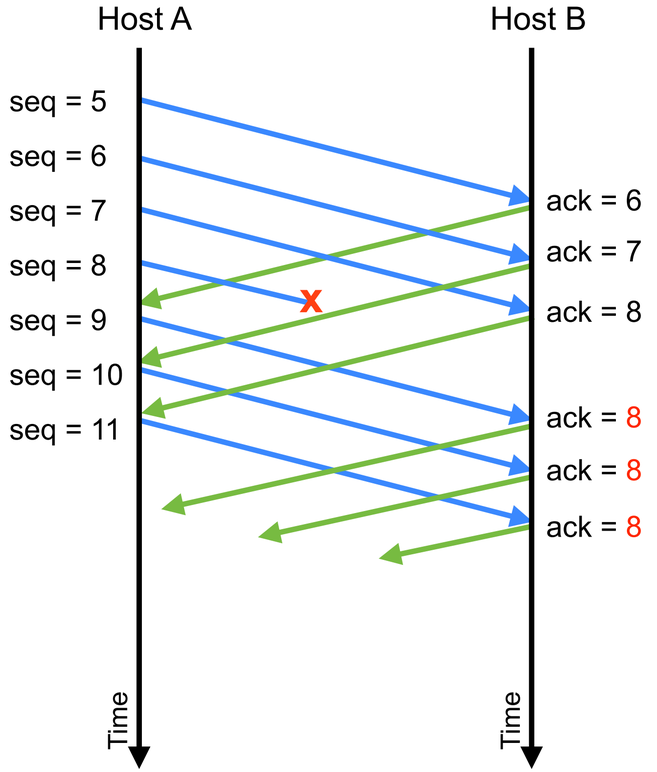
- TCP uses acknowledgements to detect lost segments
  - If data is being sent, but no acknowledgements return, then either receiver or the network failed
  - TCP treats **a timeout** as an indication of packet loss and retransmits unacknowledged segments



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.

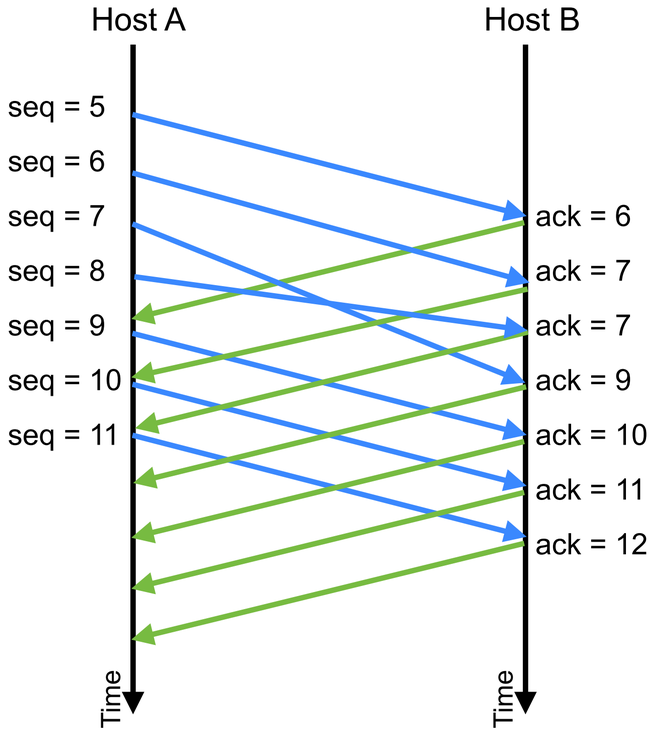
TCP Acknowledgements: Loss Detection (2/3)

- 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



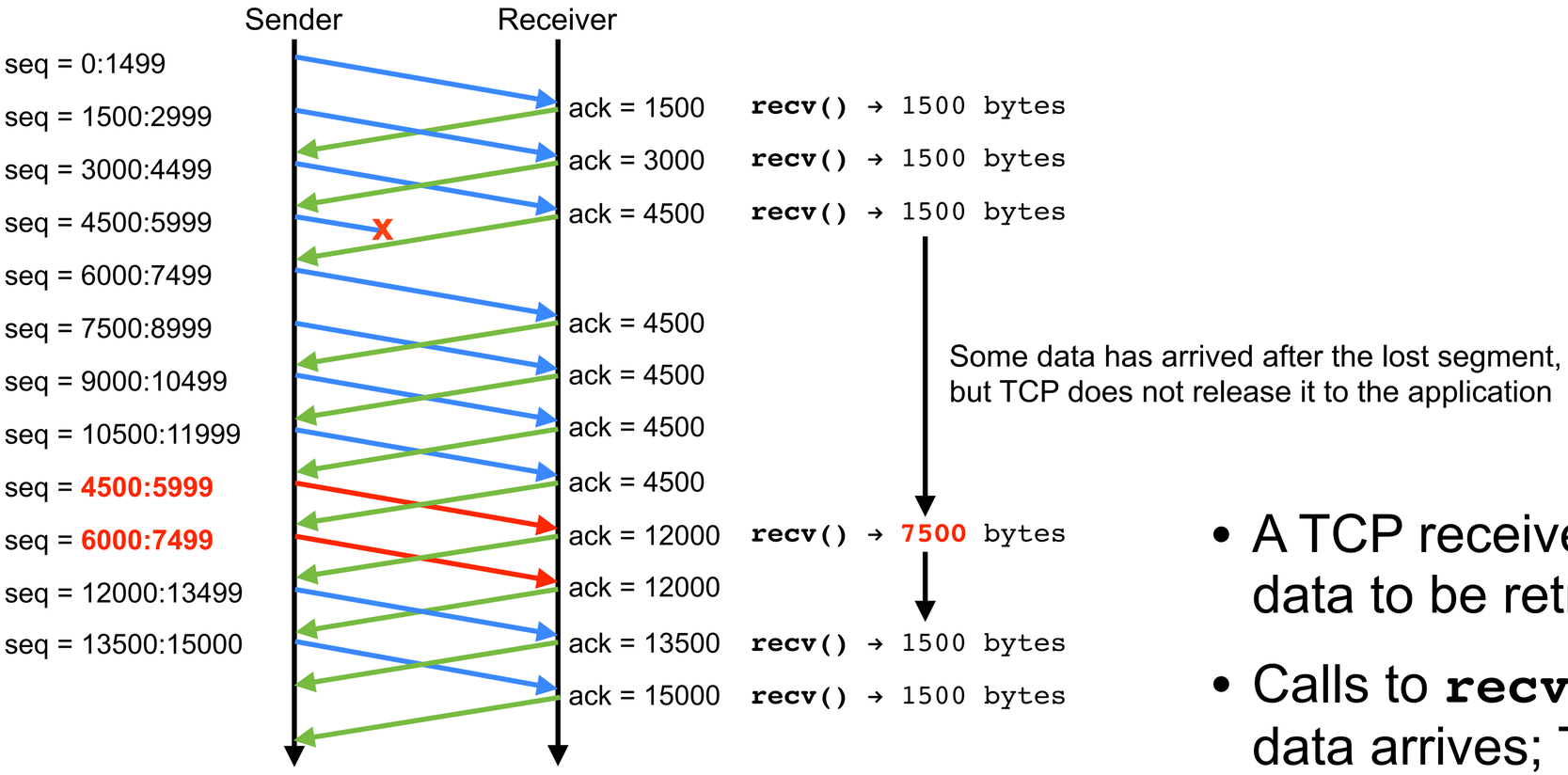
TCP connection progress, showing sequence numbers and acknowledgements; in this example, each segment carries one byte of data; the segment with sequence number 8 is lost.

TCP Acknowledgements: Loss Detection (3/3)



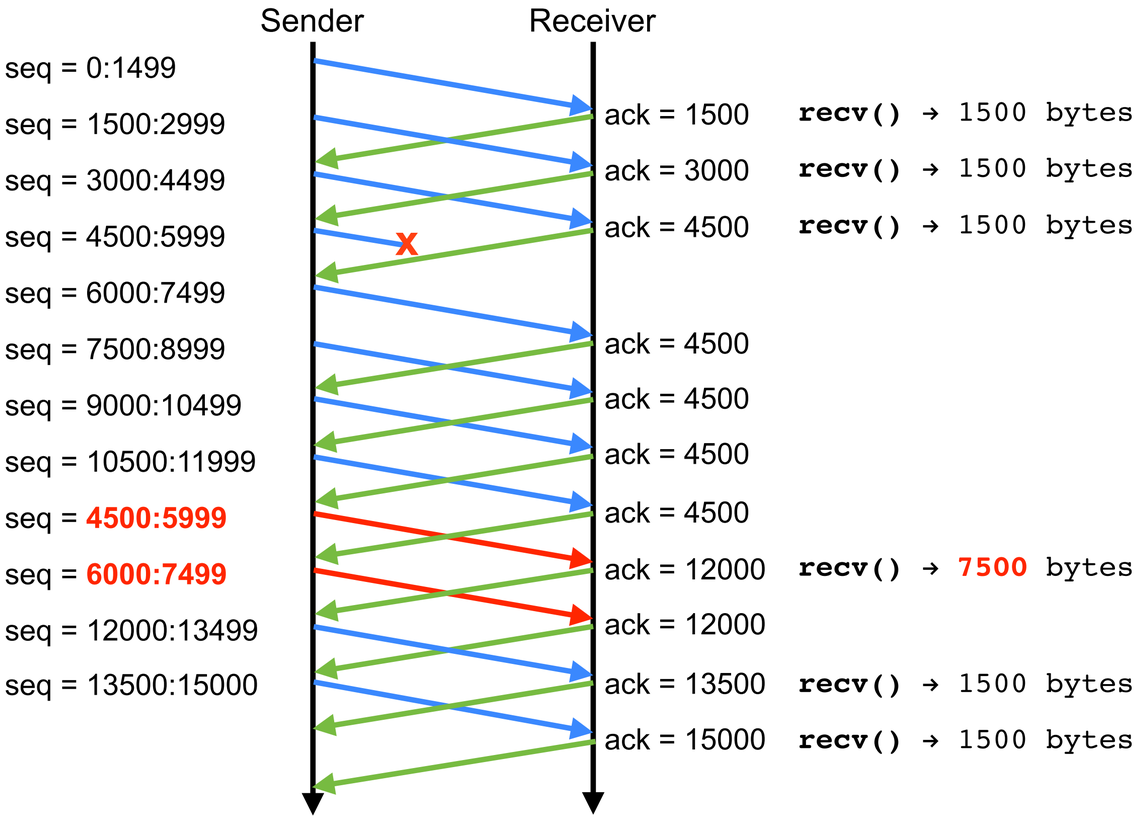
- Why a **triple duplicate acknowledgement**?
  - Packet delay leading to reordering will also cause duplicate acknowledgement to be generated
  - Gives appearance of loss, when the segments are merely delayed
  - Use of triple duplicate ACK to indicate packet loss stops reordered packets from being 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

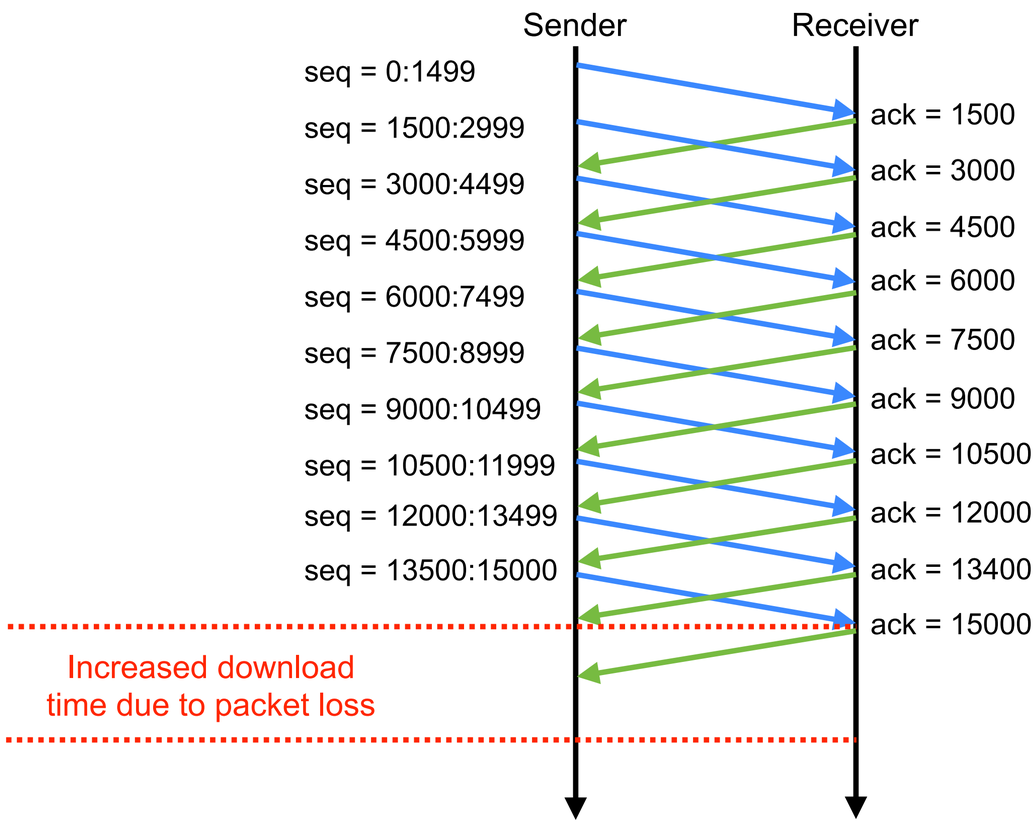
Head-of-line Blocking in TCP (1/3)



- 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 a contiguous, ordered, sequence

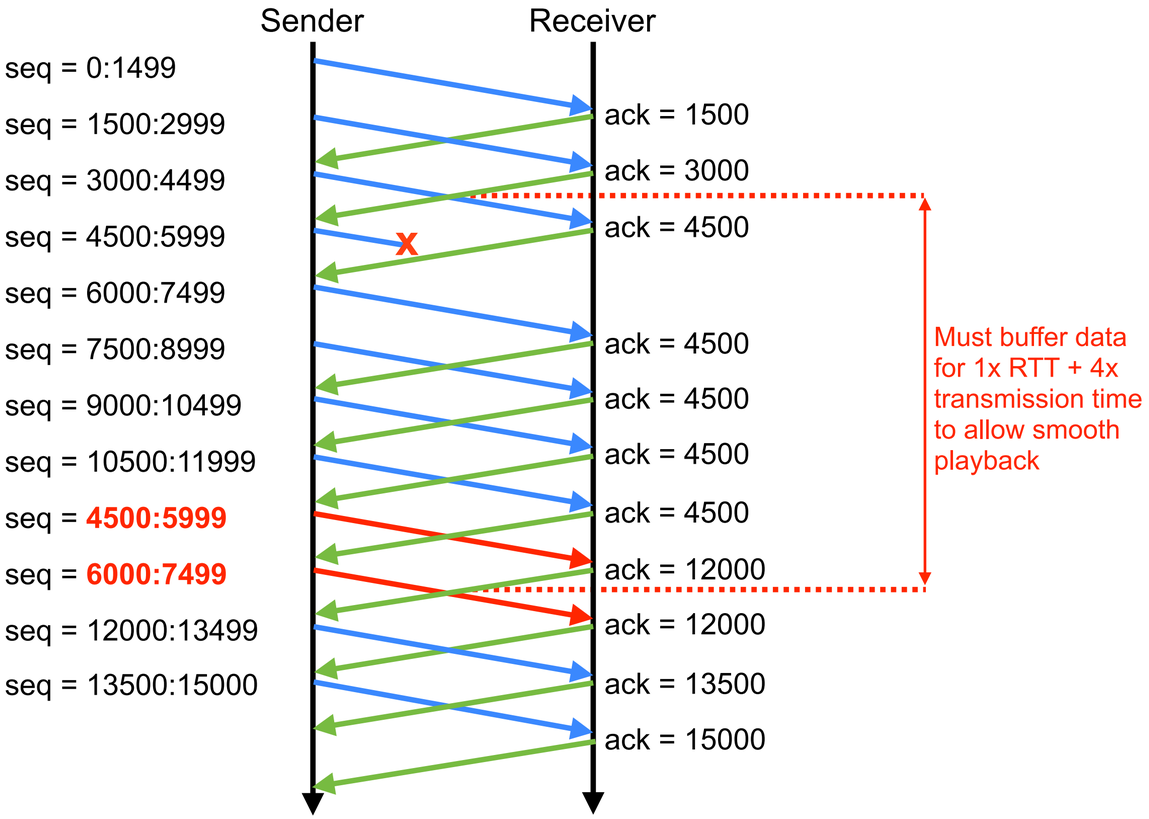
Head-of-line Blocking in TCP (2/3)

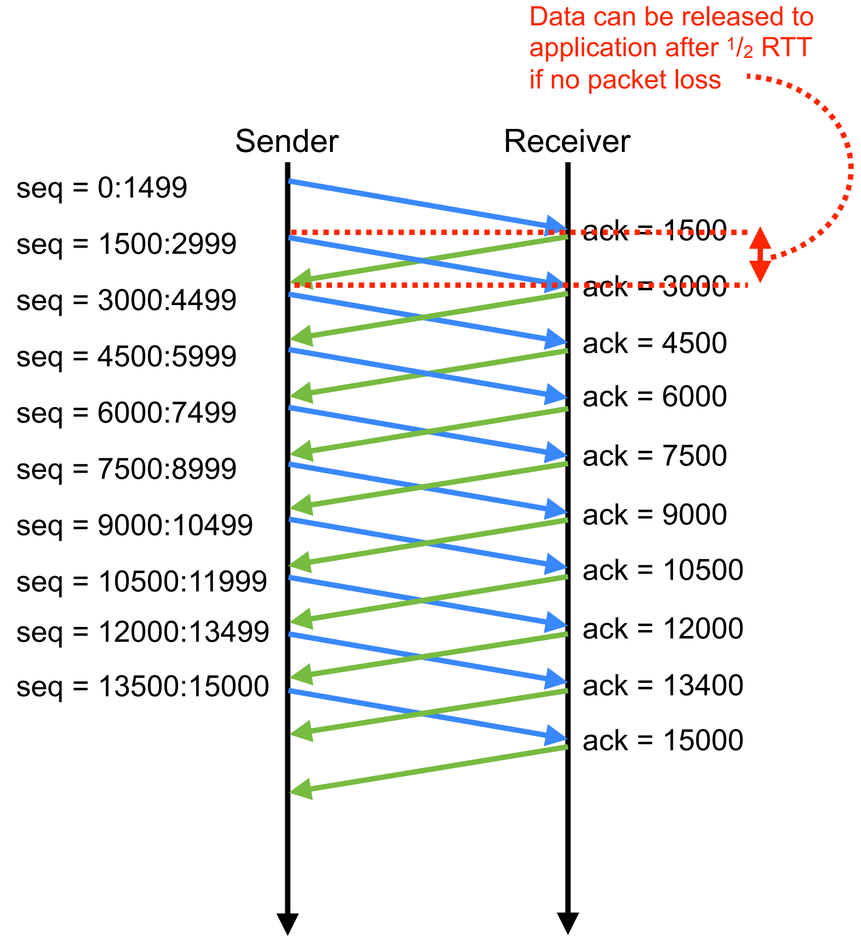




- Head-of-line blocking increases total download time
- Delay depends on relative values of RTT and packet serialisation delay

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

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

Reliable Data With TCP

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