University of Glasgow

School of Computing Science

TCP and Congestion Control

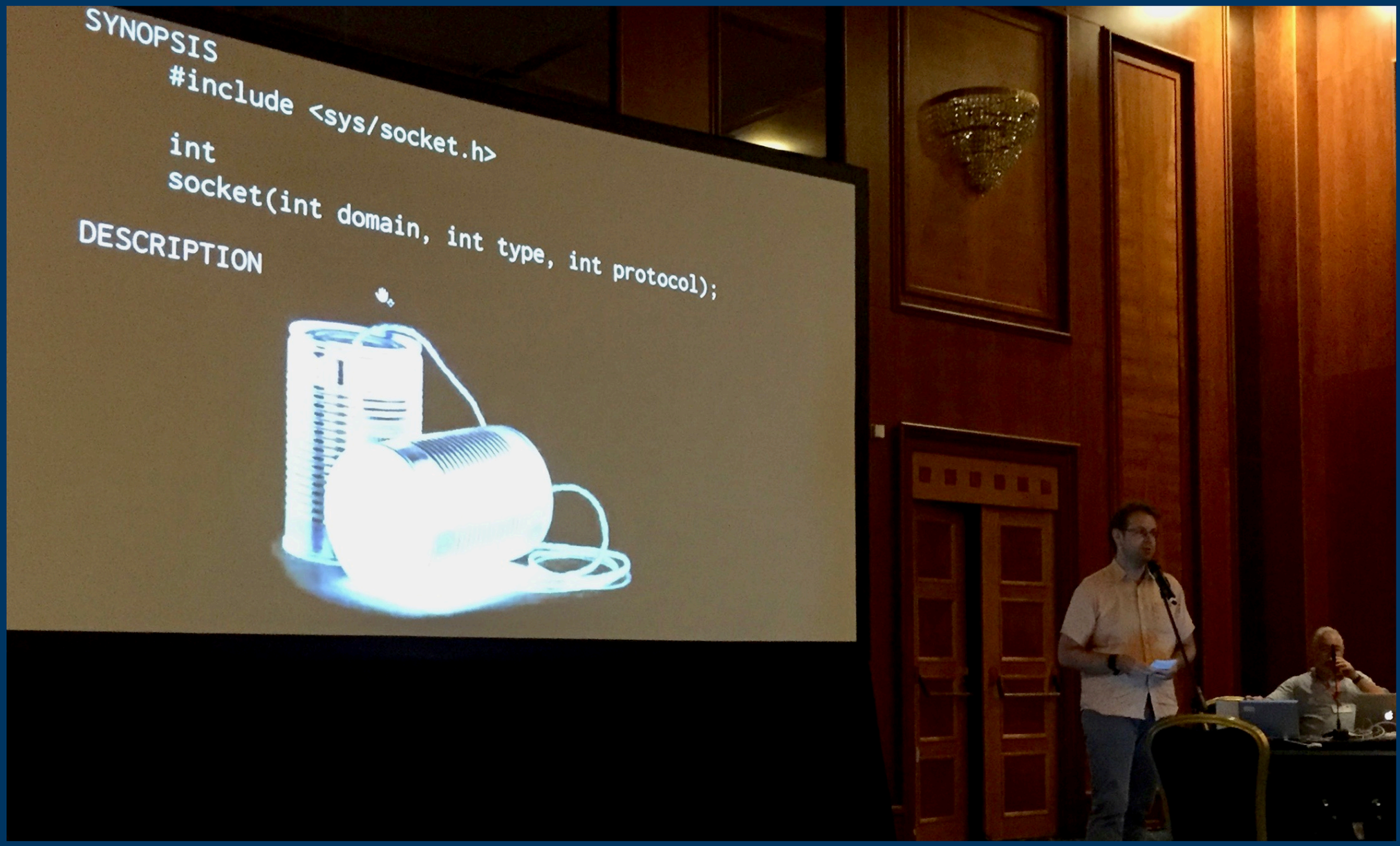
Networked Systems (H)

Lecture 7

Lecture Outline

- TCP and the Berkeley Sockets API
- Congestion control principles
- Congestion control in the Internet
  - TCP congestion control
  - Alternative approaches

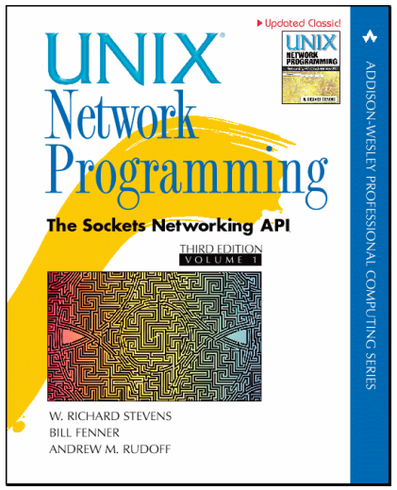
TCP and the Berkeley Sockets API



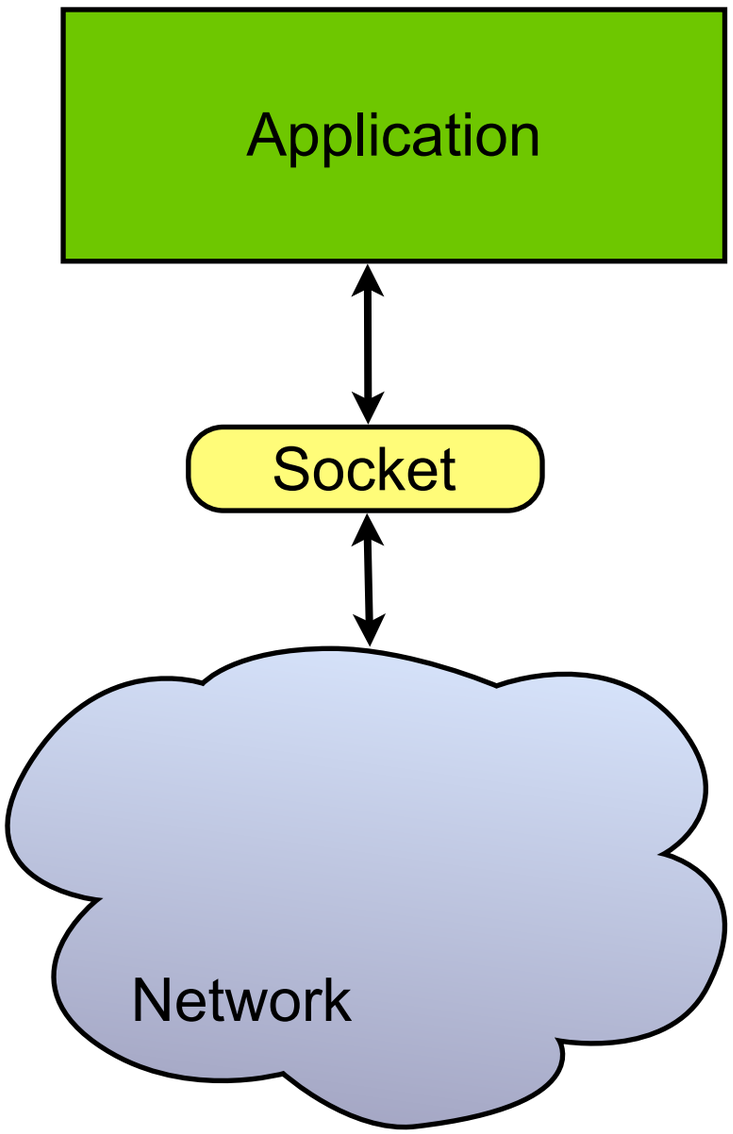
The Berkeley Sockets API

- Widely used low-level C networking API
- First introduced in 4.3BSD Unix
  - Now available on most platforms: Linux, MacOS X, Windows, FreeBSD, Solaris, etc.
  - Largely compatible cross-platform

- Recommended reading:
  - Stevens, Fenner, and Rudoff, “Unix Network Programming volume 1: The Sockets Networking API”, 3rd Edition, Addison-Wesley, 2003.

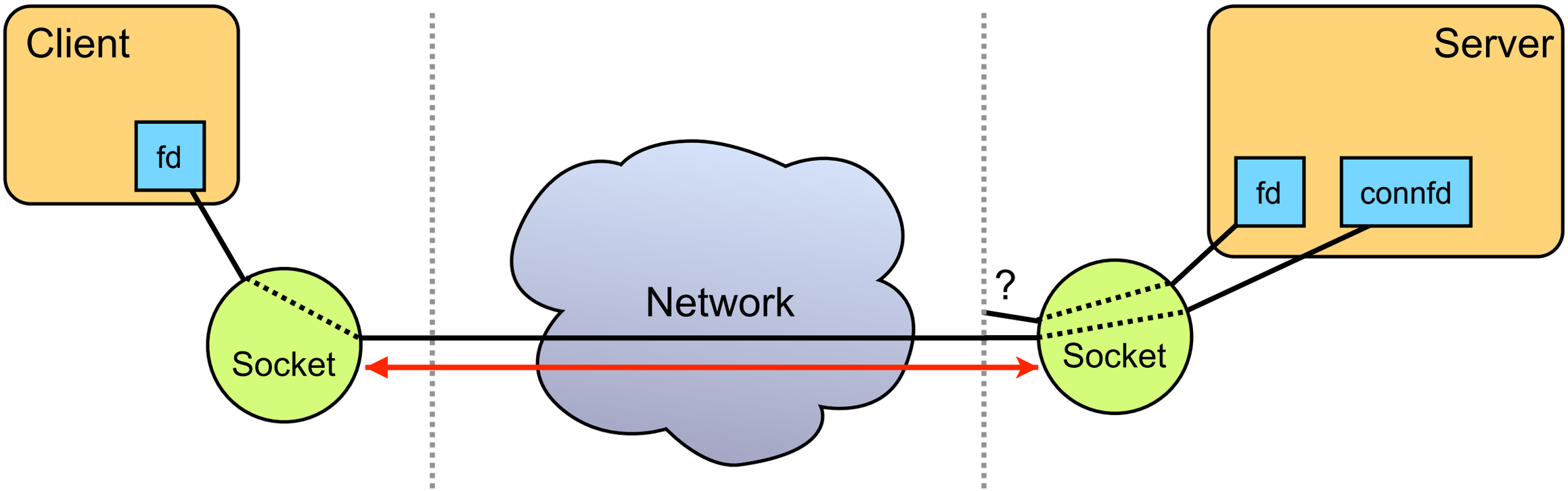


Concepts



- Sockets provide a standard interface between network and application
- Two types of socket:
  - Stream – provides a virtual circuit service
  - Datagram – delivers individual packets
- Independent of network type:
  - Commonly used with Internet Protocol sockets, so stream sockets map onto TCP/IP connections and datagram sockets onto UDP/IP, but not specific to the Internet protocols

TCP Sockets




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


connect(fd, ..., ...)

send(fd, data, datalen, flags)

recv(fd, buffer, buflen, flags)

close(fd)
```

```
int fd = socket(...)
bind(fd, ..., ...)
listen(fd, ...)
connfd = accept(fd, ...)

recv(connfd, buffer, buflen, flags)

send(connfd, data, datalen, flags)

close(connfd)
```

What services do TCP sockets provide?

- TCP provides five key features:
  - Service differentiation
  - Connection-oriented
  - Point-to-point
  - Reliable, in-order, delivery of a byte stream
  - Congestion control

- These are provided by the operating system, via the sockets API

Client-server or peer-to-peer?

- Sockets initially unbound, and can either accept or make a connection
- Most commonly used in a client-server fashion:
  - One host makes the socket `listen()` for, and `accept()`, connections on a well-known port, making it into a server
    - The port is a 16-bit number used to distinguish servers
      - E.g. web server listens on port 80, email server on port 25
  - The other host makes the socket `connect()` to that port on the server
  - Once connection is established, either side can `send()` data into the connection, where it becomes available for the other side to `recv()`
- Simultaneous connections are possible, using TCP in a peer-to-peer manner

Role of the TCP Port Number

Port Range		Name	Intended use
0	1023	Well-known (system) ports	Trusted operating system services
1024	49151	Registered (user) ports	User applications and services
49152	65535	Dynamic (ephemeral) ports	Private use, peer-to-peer applications, source ports for TCP client connections

RFC 6335

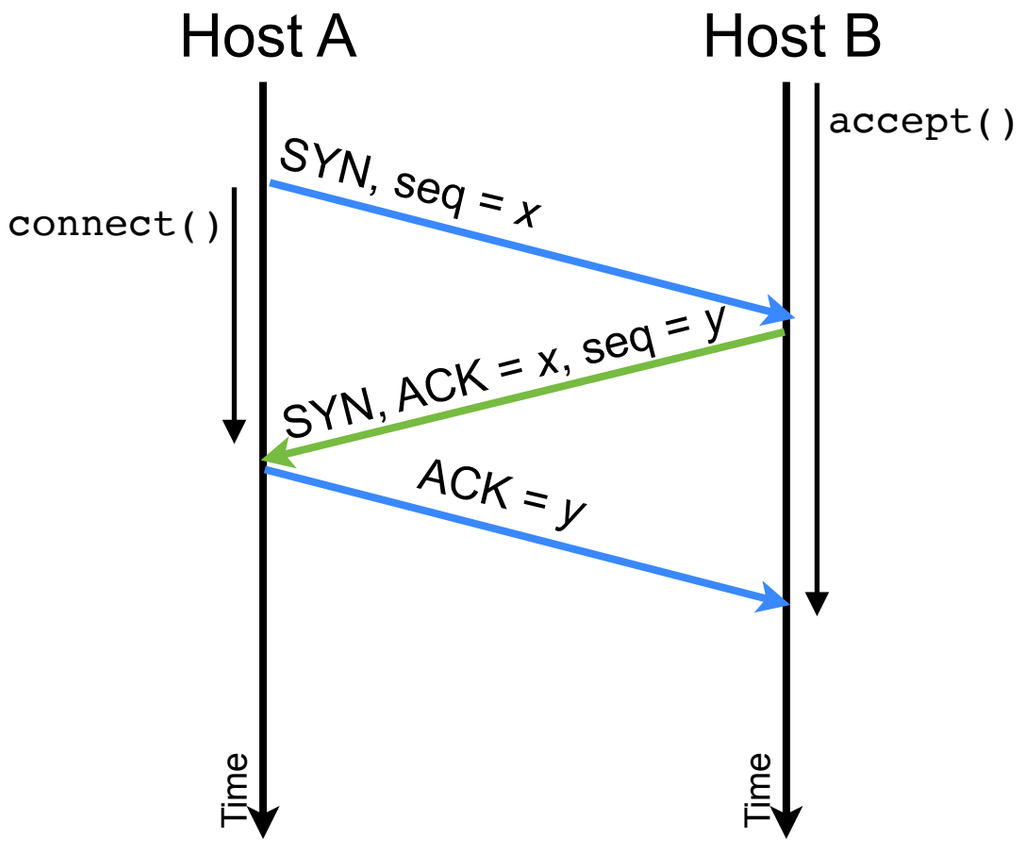
- Servers must listen on a known port; IANA maintains a registry
- Distinction between system and user ports ill-advised – security problems resulted
- Insufficient port space available (>75% of ports are registered)

- TCP clients traditionally connect from a randomly chosen port in the ephemeral range
  - The port must be chosen randomly, to prevent spoofing attacks
  - Many systems use the entire port range for source ports, to increase the amount of randomness available

http://www.iana.org/assignments/port-numbers

TCP Connection Setup

- Connections use 3-way handshake
  - The SYN and ACK flags in the TCP header signal connection progress
  - Initial packet has SYN bit set, includes randomly chosen initial sequence number
  - Reply also has SYN bit set and randomly chosen sequence number, acknowledges initial packet
  - Handshake completed by acknowledgement of second packet
  - Happens during the `connect()`/`accept()` calls
- Combination ensures robustness
  - Randomly chosen initial sequence numbers give robustness to delayed packets or restarted hosts
  - Acknowledgements ensure reliability



Similar handshake ends connection, with FIN bits signalling the teardown

Reading and Writing Data on a TCP Connection

```
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
    ...
}
```

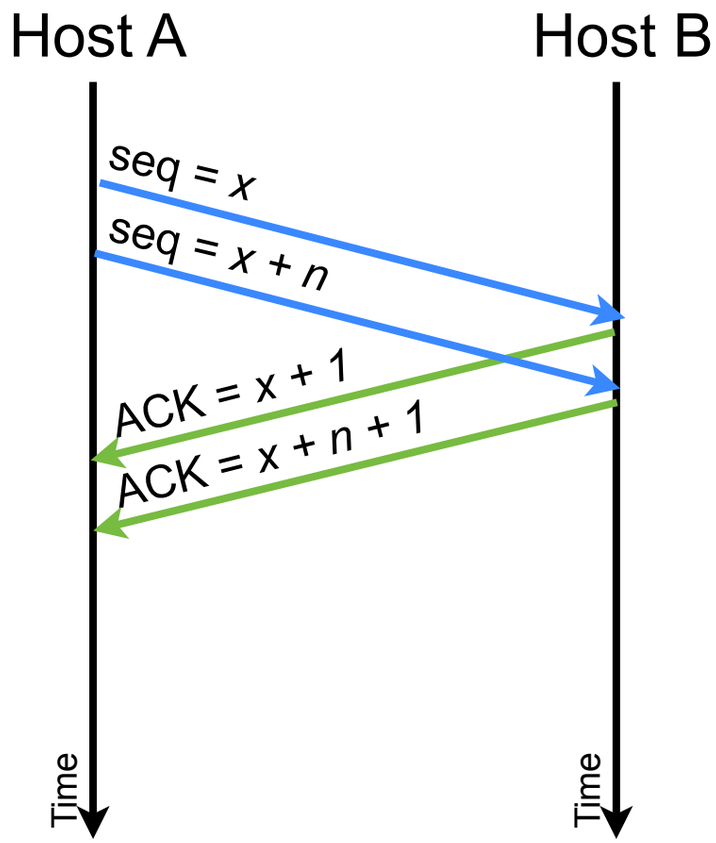
- Call `send()` to transmit data
  - Will block until the data can be written, and returns actual amount of data sent
    - Might not be able to send all the data, if the connection is congested
  - Returns -1 if error occurs, sets `errno`

```
#define BUFLEN 1500
...
ssize_t i;
ssize_t rcount;
char    buf[BUFLEN];
...
rcount = recv(fd, buf, BUFLEN, 0);
if (rcount == -1) {
    // Error has occurred
    ...
}
...
for (i = 0; i < rcount; i++) {
    printf("%c", buf[i]);
}
```

- Call recv() to read up to BUFLEN bytes of data from a connection
  - Will block until some data is available or the connection is closed
  - Returns the number of bytes read from the socket; 0 if the sender closed the connection; or -1 and sets `errno` if an error occurred
  - Received data is *not* null terminated – potential security risk

Reading and Writing Data on a TCP Connection

- The `send()` call enqueues data for transmission
- This data is split into *segments*, each segment is placed in a *TCP packet*, that packet is sent when allowed by the congestion control algorithm
  - Segments have sequence numbers → acknowledged by the receiver
- If the data in a `send()` call is too large to fit into one segment, the TCP implementation will split it into several segments; similarly, several `send()` requests might be aggregated into a single TCP segment
  - Both are done transparently by the TCP implementation and are invisible to the application
- Implication: the data returned by `recv()` doesn't necessarily correspond to a single `send()` call



Application Level Framing

The `recv()` call can return data in unpredictably sized chunks – applications must be written to cope with this

```
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

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

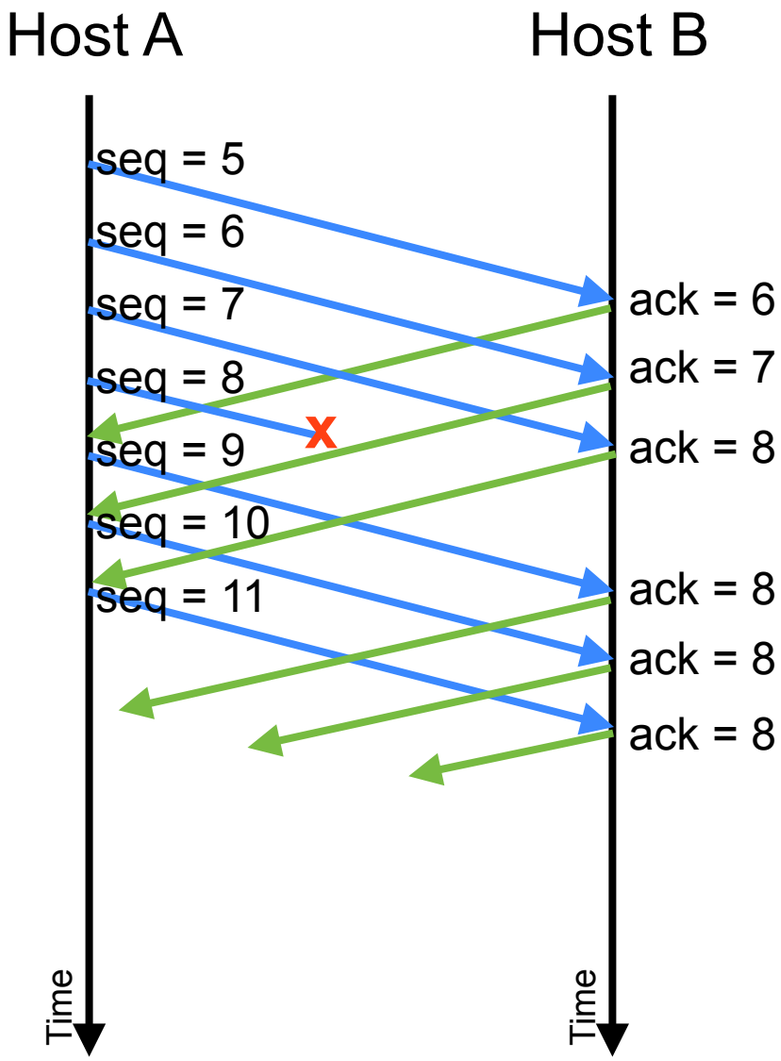
Example: HTTP/1.1 response

Ideally all headers received in one `recv()` call, then parsed to extract the Content-Length, then read entire body

TCP might split the response arbitrarily – parsing becomes more complex

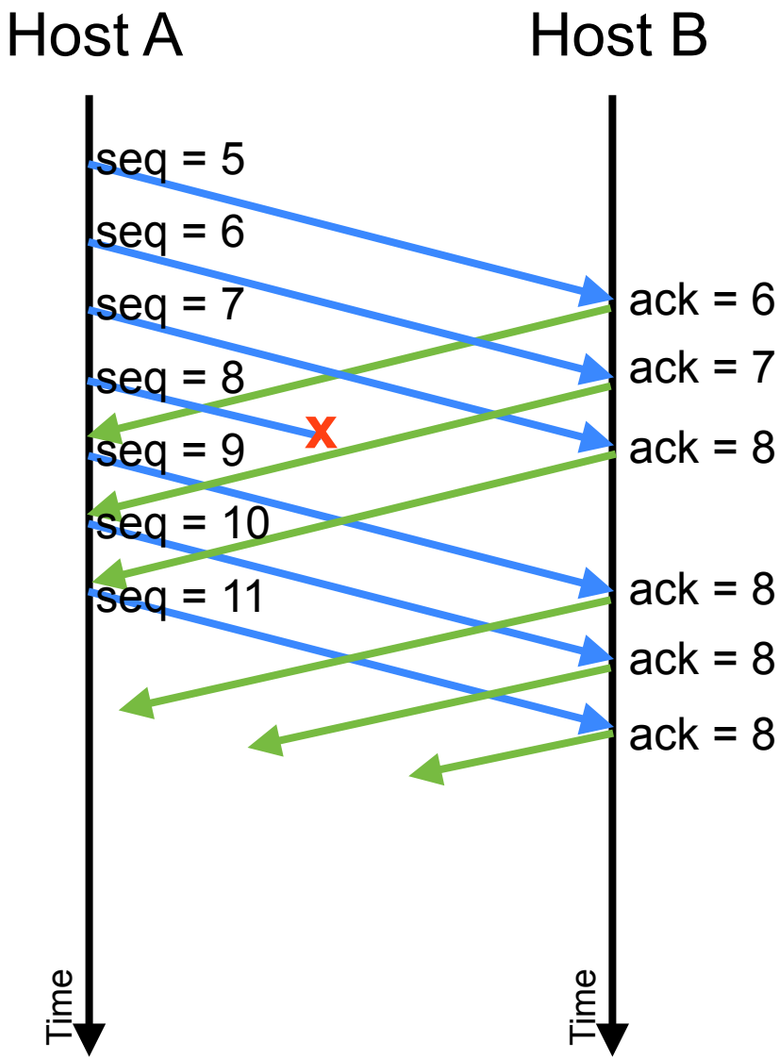
TCP Reliability

- TCP connections are reliable
  - Each TCP packet has a sequence number and an acknowledgement number
    - Sequence number counts how many bytes are sent (this example is unrealistic, since it shows one byte being sent per packet)
  - Acknowledgement number specifies the next byte expected to be received
    - Cumulative positive acknowledgement
    - Only acknowledge contiguous data packets (sliding window protocol, so several data packets in flight)
    - If a packet is lost, receipt of subsequent packets will trigger duplicate acknowledgements
- TCP layer retransmits lost packets – this is invisible to the application

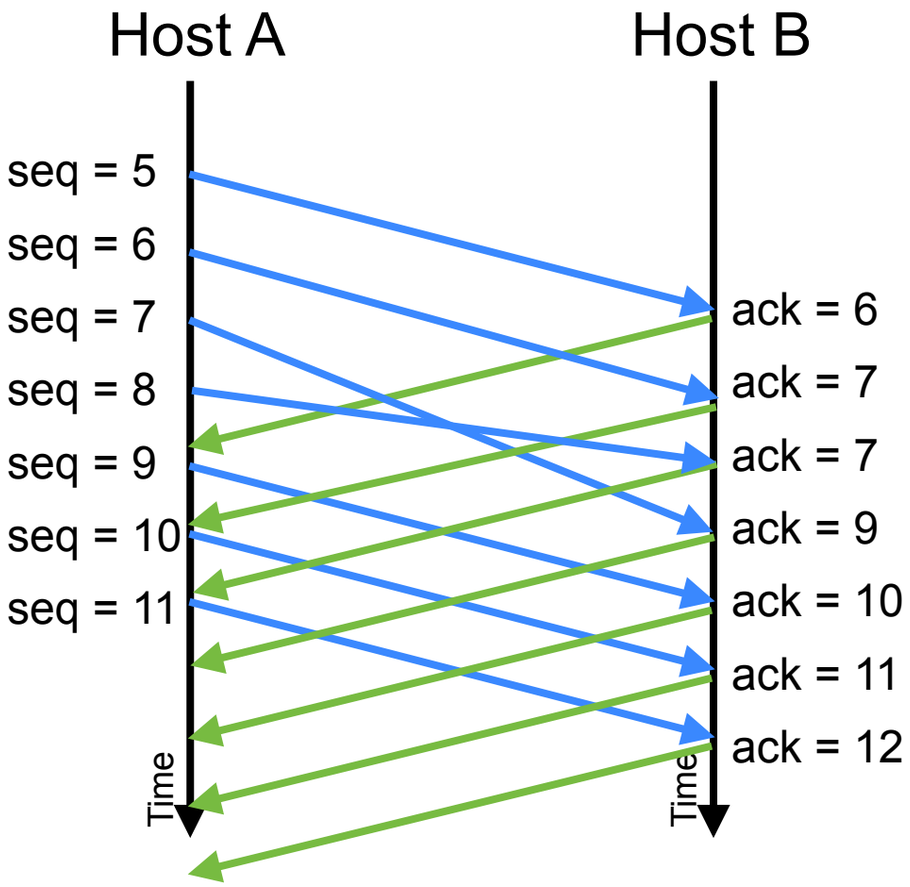


TCP Reliability: How is Loss Detected

- Triple duplicate ACK → some packets lost, but later packets arriving
  - Triple duplicate = Four identical ACKs in a row
- Timeout → send data but acknowledgements stop returning
  - Either the receiver or the network path has failed



TCP Reliability and Packet Reordering

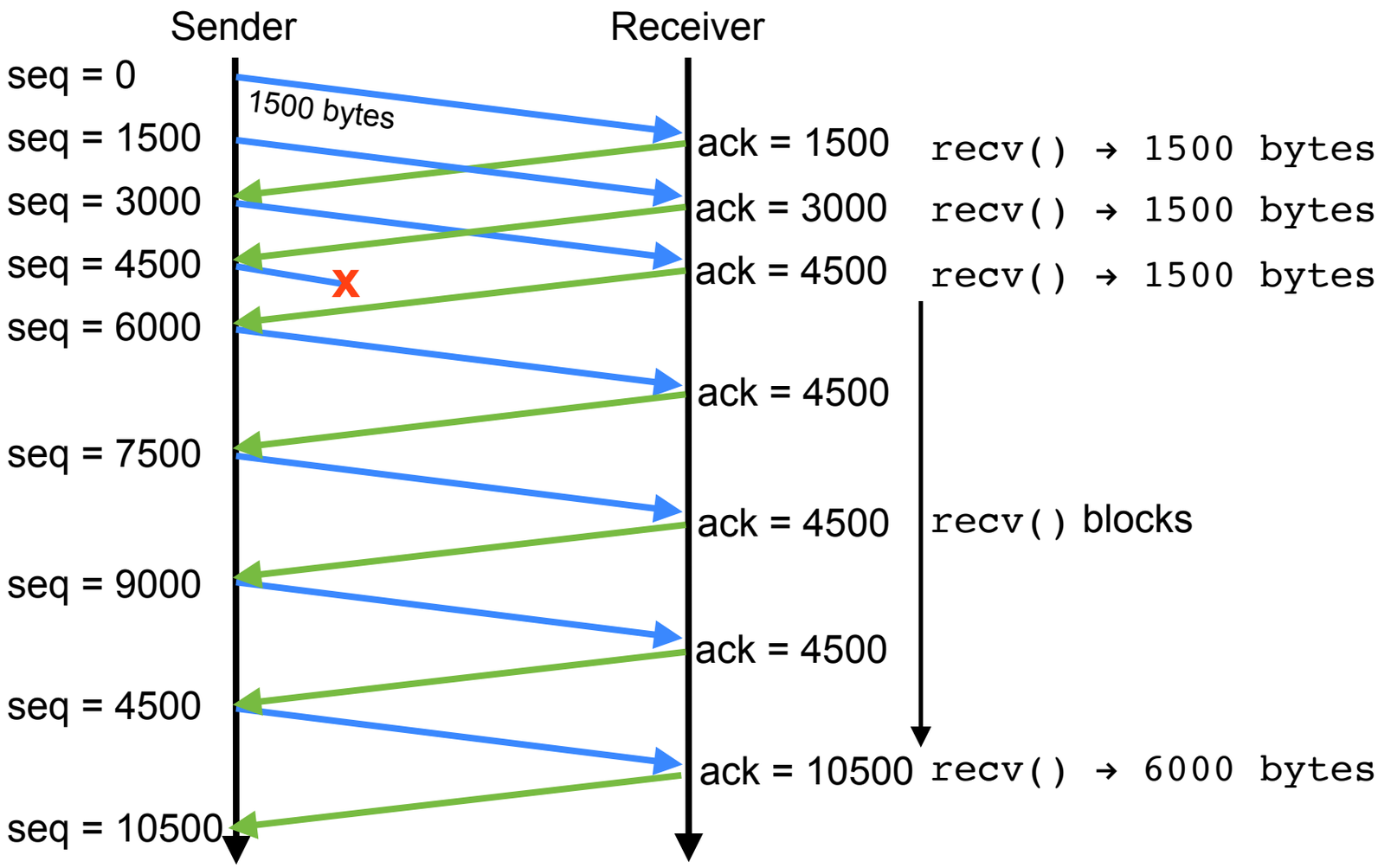


- Packet delay leading to reordering will also cause duplicate ACKs to be received
- Gives appearance of loss, when the data was merely delayed

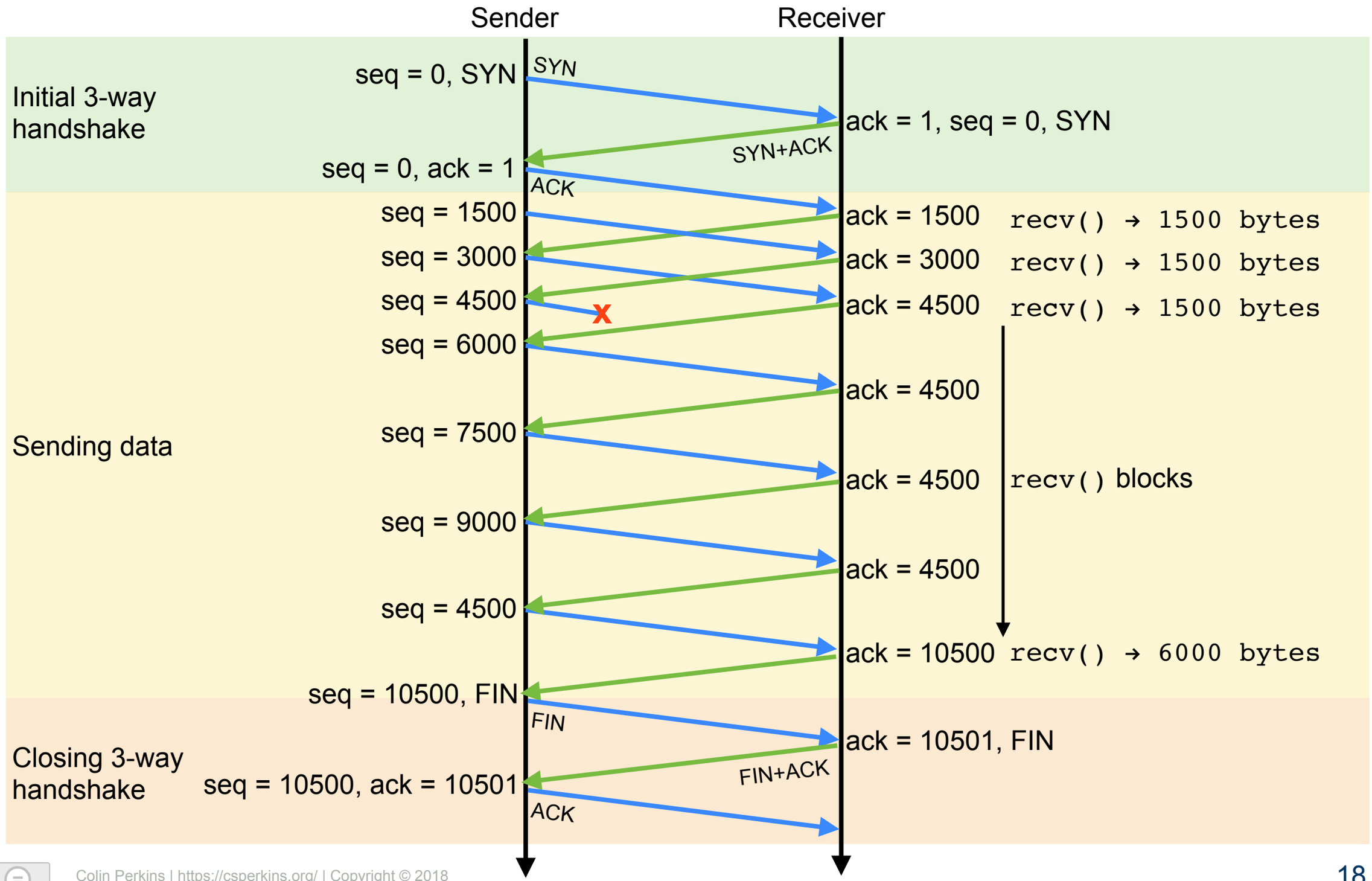
- TCP uses triple duplicate ACK as indication of packet loss to prevent reordered packets causing retransmissions
  - Assumption: packets will only be delayed a little; if delayed enough that a triple duplicate ACK is generated, TCP will treat the packet as lost and send a retransmission

Head of Line Blocking in TCP

- Data delivered in order, even after loss occurs
  - TCP will retransmit the missing data, transparently to the application
  - A `recv()` for missing data will block until it arrives; TCP always delivers data in an in-order contiguous sequence



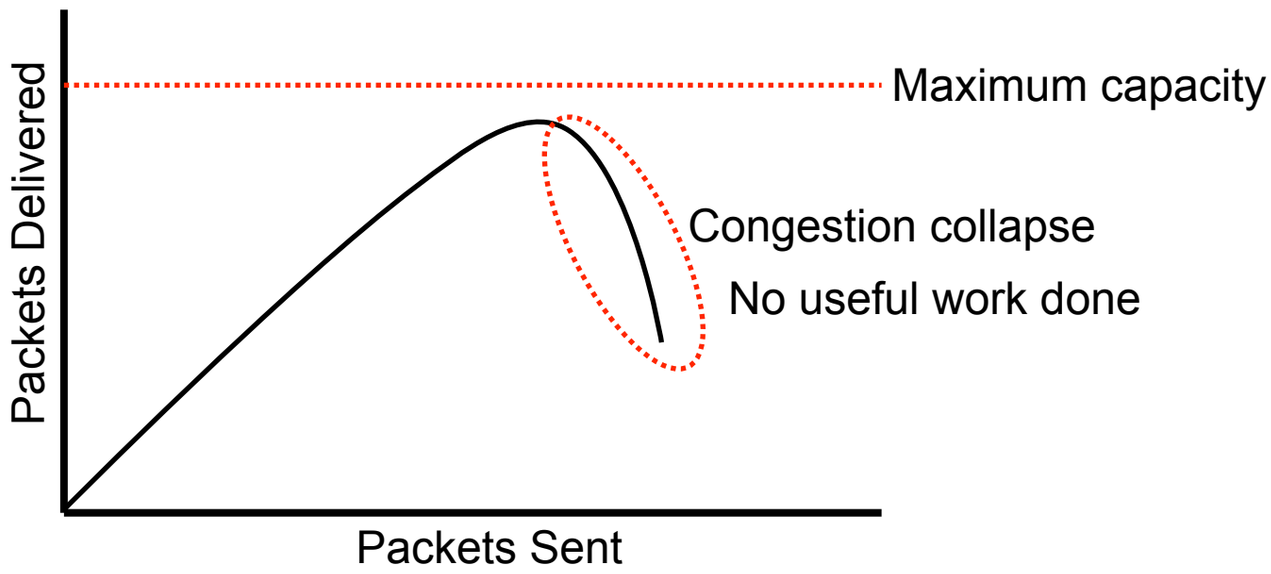
A Complete TCP Connection



Congestion Control

What is Congestion Control?

- Adapting speed of transmission to match available end-to-end network capacity
- Preventing congestion collapse of a network

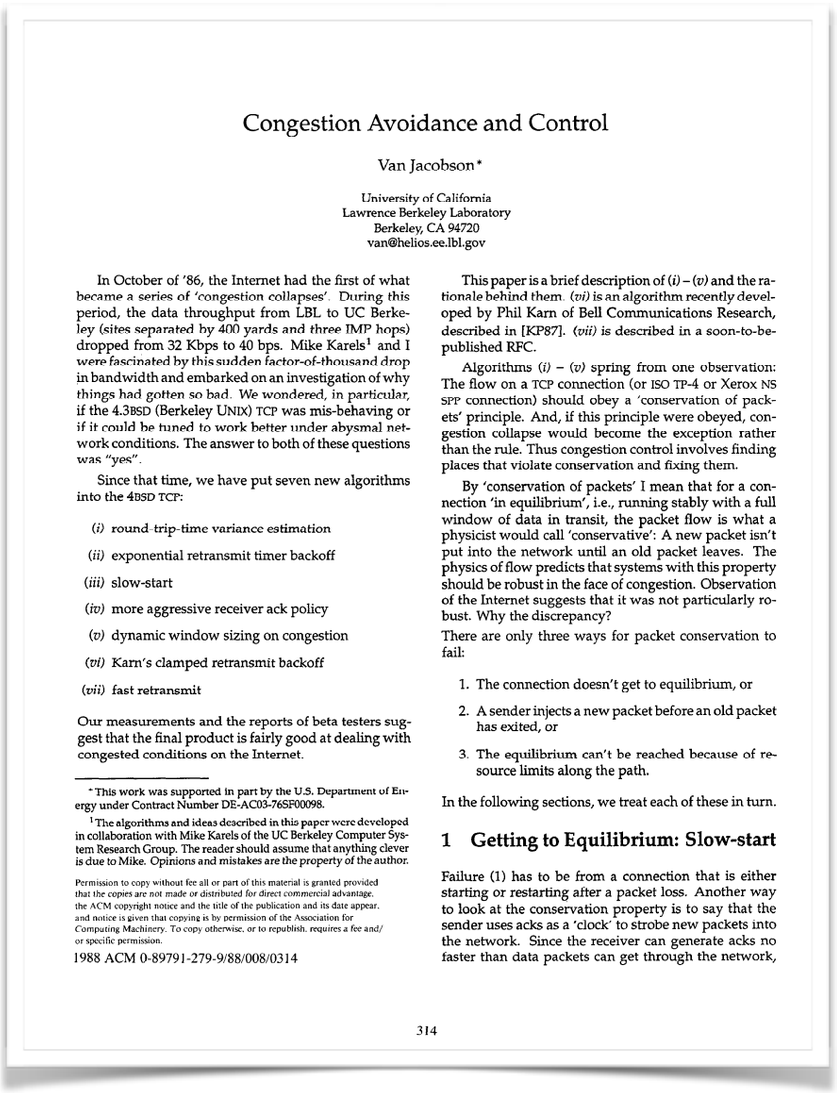


Occurred in the Internet in 1986, before congestion control added

Network or Transport Layer?

- Can implement congestion control at either the network or the transport layer
  - Network layer – safe, ensures all transport protocols are congestion controlled, requires all applications to use the same congestion control scheme
  - Transport layer – flexible, transports protocols can optimise congestion control for applications, but a misbehaving transport can congest the network

Congestion Control Principles



Congestion Avoidance and Control

Van Jacobson*

University of California
Lawrence Berkeley Laboratory
Berkeley, CA 94720
van@helios.ee.lbl.gov

In October of '86, the Internet had the first of what became a series of 'congestion collapses'. During this period, the data throughput from LBL to UC Berkeley (sites separated by 400 yards and three IMP hops) dropped from 32 Kbps to 40 bps. Mike Karels[1] and I were fascinated by this sudden factor-of-thousand drop in bandwidth and embarked on an investigation of why things had gotten so bad. We wondered, in particular, if the 4.3BSD (Berkeley UNIX) TCP was mis-behaving or if it could be tuned to work better under abysmal network conditions. The answer to both of these questions was "yes".

Since that time, we have put seven new algorithms into the 4BSD TCP:

(*i*) round-trip-time variance estimation

(*ii*) exponential retransmit timer backoff

(*iii*) slow-start

(*iv*) more aggressive receiver ack policy

(*v*) dynamic window sizing on congestion

(*vi*) Karn's clamped retransmit backoff

(*vii*) fast retransmit

Our measurements and the reports of beta testers suggest that the final product is fairly good at dealing with congested conditions on the Internet.

This paper is a brief description of (*i*) – (*v*) and the rationale behind them. (*vi*) is an algorithm recently developed by Phil Karn of Bell Communications Research, described in [KP87]. (*vii*) is described in a soon-to-be-published RFC.

Algorithms (*i*) – (*v*) spring from one observation: The flow on a TCP connection (or ISO TP-4 or Xerox NS SPP connection) should obey a 'conservation of packets' principle. And, if this principle were obeyed, congestion collapse would become the exception rather than the rule. Thus congestion control involves finding places that violate conservation and fixing them.

By 'conservation of packets' I mean that for a connection 'in equilibrium', i.e., running stably with a full window of data in transit, the packet flow is what a physicist would call 'conservative': A new packet isn't put into the network until an old packet leaves. The physics of flow predicts that systems with this property should be robust in the face of congestion. Observation of the Internet suggests that it was not particularly robust. Why the discrepancy?

There are only three ways for packet conservation to fail:

1. The connection doesn't get to equilibrium, or
2. A sender injects a new packet before an old packet has exited, or
3. The equilibrium can't be reached because of resource limits along the path.

In the following sections, we treat each of these in turn.

1 Getting to Equilibrium: Slow-start

Failure (1) has to be from a connection that is either starting or restarting after a packet loss. Another way to look at the conservation property is to say that the sender uses acks as a 'clock' to strobe new packets into the network. Since the receiver can generate acks no faster than data packets can get through the network,

* This work was supported in part by the U.S. Department of Energy under Contract Number DE-AC03-76SF00098.

[1] The algorithms and ideas described in this paper were developed in collaboration with Mike Karels of the UC Berkeley Computer System Research Group. The reader should assume that anything clever is due to Mike. Opinions and mistakes are the property of the author.

1988 ACM 0-89791-279-9/88/008/0314

V. Jacobson, "Congestion avoidance and control", Proceedings of the SIGCOMM Conference, Stanford, CA, USA, August 1988. ACM. http://dx.doi.org/10.1145/52324.52356

- Two key principles, first stated by Van Jacobson in 1988:
  - Conservation of packets
  - Additive increase and multiplicative decrease of the sending rate
- Together, ensure stability of the network

Van Jacobson

Source: PARC

- Congestion control standards in IETF maintained by Sally Floyd for many years
  - High-speed TCP extensions, Quick start, SACK, ECN, etc.

Sally Floyd

Source: Sally Floyd

Conservation of Packets

- The network has a certain capacity
  - The *bandwidth x delay* product of the path
- When in equilibrium at that capacity, send one packet for each acknowledgement received
  - Total number of packets in transit is constant
  - "ACK clocking" – each acknowledgement "clocks out" the next packet
- Automatically reduces sending rate as network gets congested and delivers packets more slowly

AIMD Algorithms

- Adjust sending rate according to an additive increase/multiplicative decrease algorithm
  - Start slowly, increase gradually to find equilibrium
    - Add a small amount to the sending speed each time interval without loss
    - For a window-based algorithm $w_i = w_{i-1} + \alpha$ each RTT, where $\alpha = 1$ typically
  - Respond to congestion rapidly
    - Multiply sending window by some factor $\beta < 1$ each interval loss seen
    - For a window-based algorithm $w_i = w_{i-1} \times \beta$ each RTT, where $\beta = 1/2$ typically
- Faster reduction than increase → stability

Congestion in the Internet

- Network layer signals that congestion is occurring to the transport
- Two ways this is done:
  - Packet arrives at router, but queue for outgoing link is full → router discards the packet (this is the common case)
  - Packet arrives at router, queue for outgoing link is getting close to full, and transport has signalled that it understands ECN → router sets ECN-CE bit in the packet header
- Transport protocol (e.g., TCP) detects congestion signal and reacts
  - Receiver detects packet loss due to gap in sequence number space; or the receiver notices the ECN-CE mark in the packet header
  - When no congestion signal → gradual additive increase in the sending rate
  - When congestion signal received → multiplicative decrease in sending rate
  - AIMD algorithm, following Jacobson's principles

TCP Congestion Control

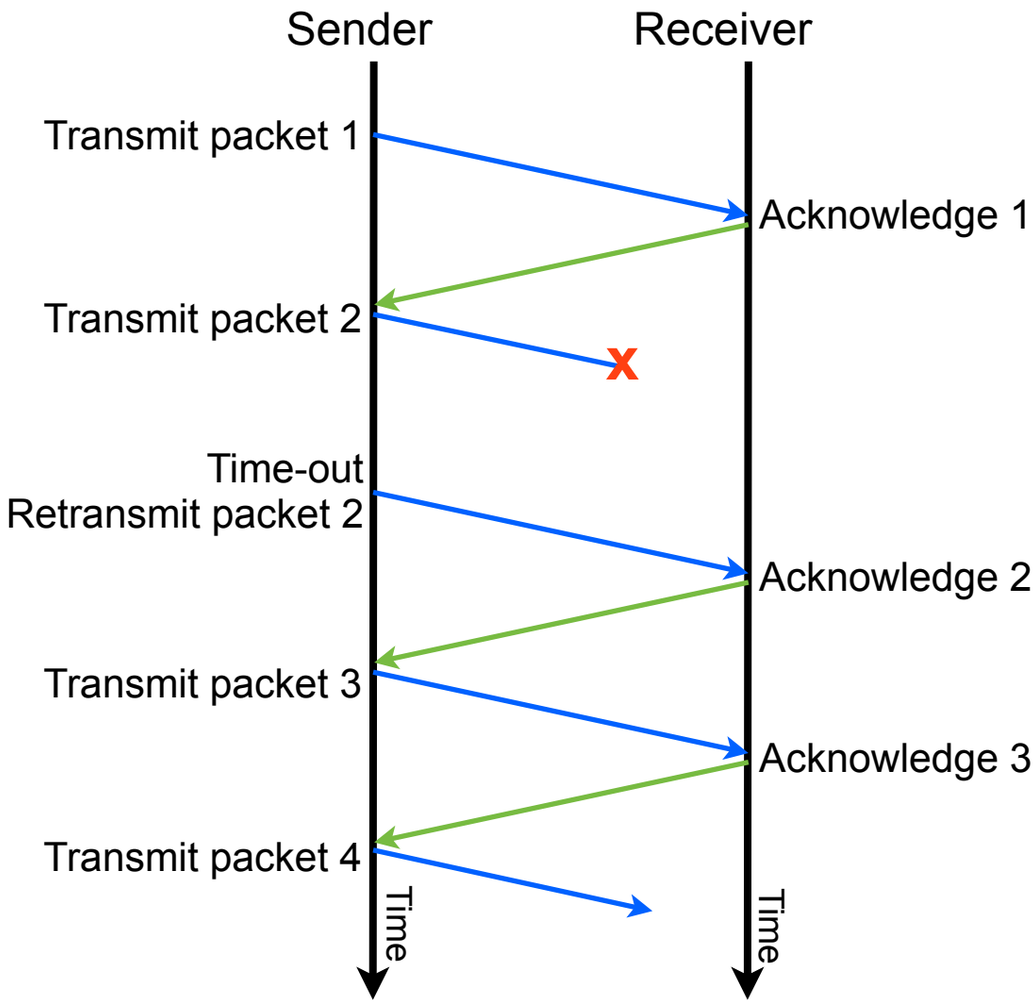
- TCP uses a window-based congestion control algorithm
  - Maintains a *sliding window* onto the available data that determines how much can be sent according to the AIMD algorithm
  - Plus slow start and congestion avoidance
  - Gives approximately equal share of the bandwidth to each flow sharing a link

- The following slides give an outline of TCP Reno congestion control
  - The state of the art in TCP as of ~1990
  - See RFC 7414 (https://tools.ietf.org/html/rfc7414) for a roadmap of current TCP specifications (57 pages, referencing ~150 other documents)
  - “The world’s most baroque sliding-window protocol” – Lloyd Wood

Sliding Window Protocol: Stop and Wait

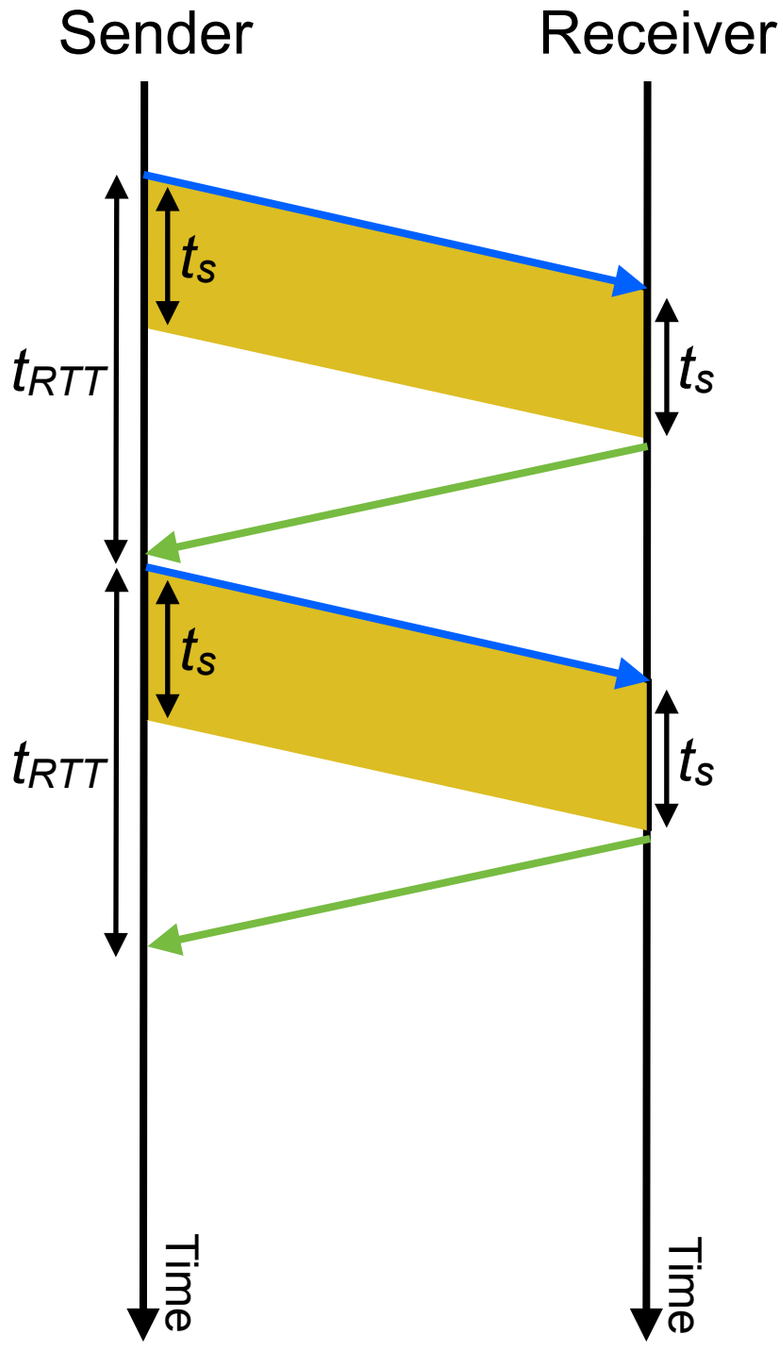
- Consider a simple stop-and-wait protocol
  - Transmit a packet of data, and then wait for acknowledgement from receiver
  - When acknowledgement received, send next packet
  - If no acknowledgement after some time out, retransmit packet

- Limits sender to one frame outstanding → poor performance



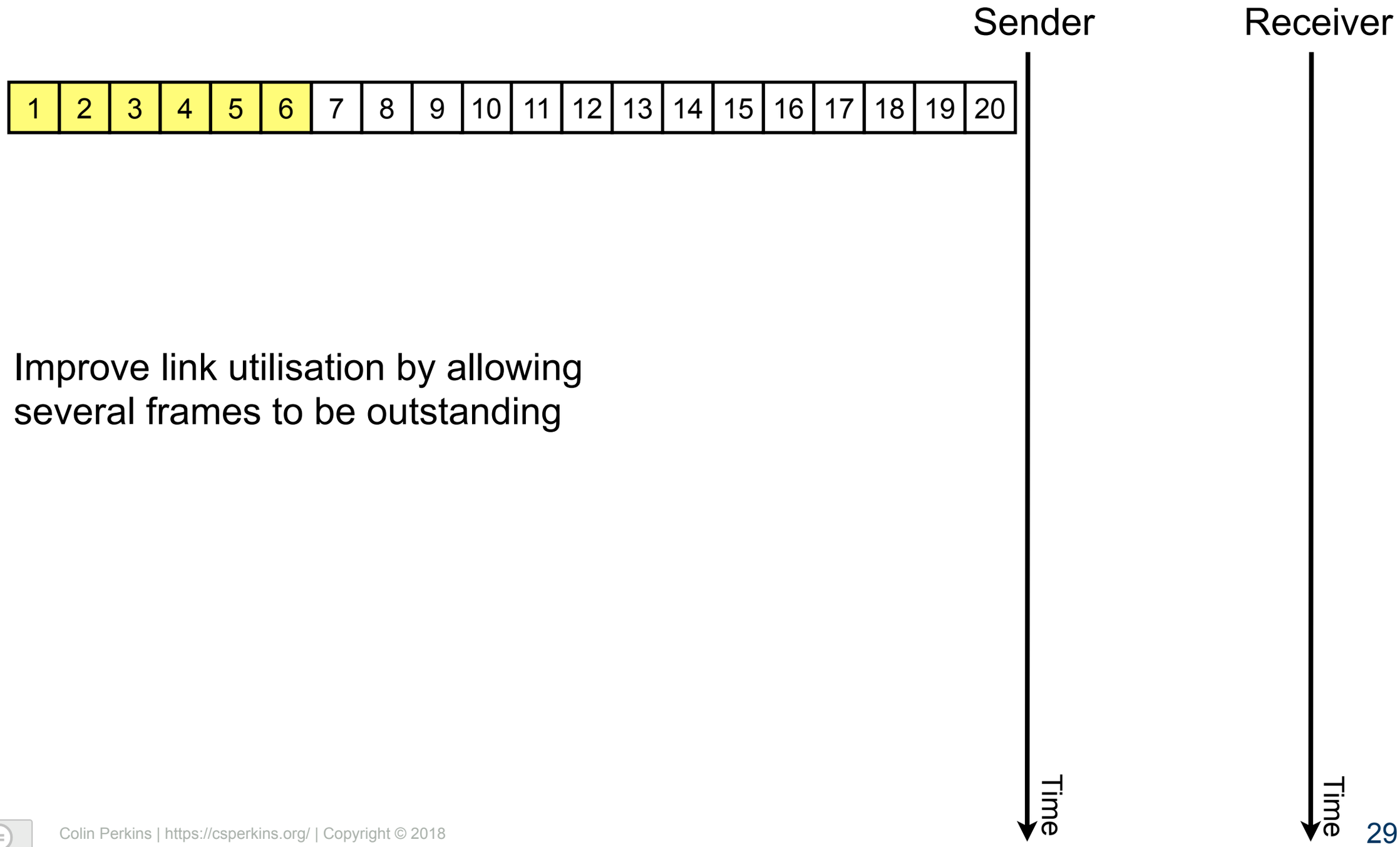
Sliding Window Protocol: Link Utilisation

- Why does stop-and wait perform poorly?
- It takes time, t_s, to send a packet
  - t_s = (packet size) / (link bandwidth)
- Acknowledgement returns t_{RTT} seconds later
- Link utilisation, $U = t_s / t_{RTT}$
  - The fraction of the time the link is in use sending packets – ideally, we want $U \approx 1.0$
  - Assume a gigabit link sending a 1500 byte packet from Glasgow to London:
    - t_s = 1500×8 bits / 10^9 bits per second = 0.000012s
    - $t_{RTT} \approx 0.010$ seconds
    - $U \approx 0.0012$
    - *i.e.*, the link is in use 0.12% of the time
- Sliding window protocols improve on stop-and-wait by sending more than one packet before stopping for acknowledgement



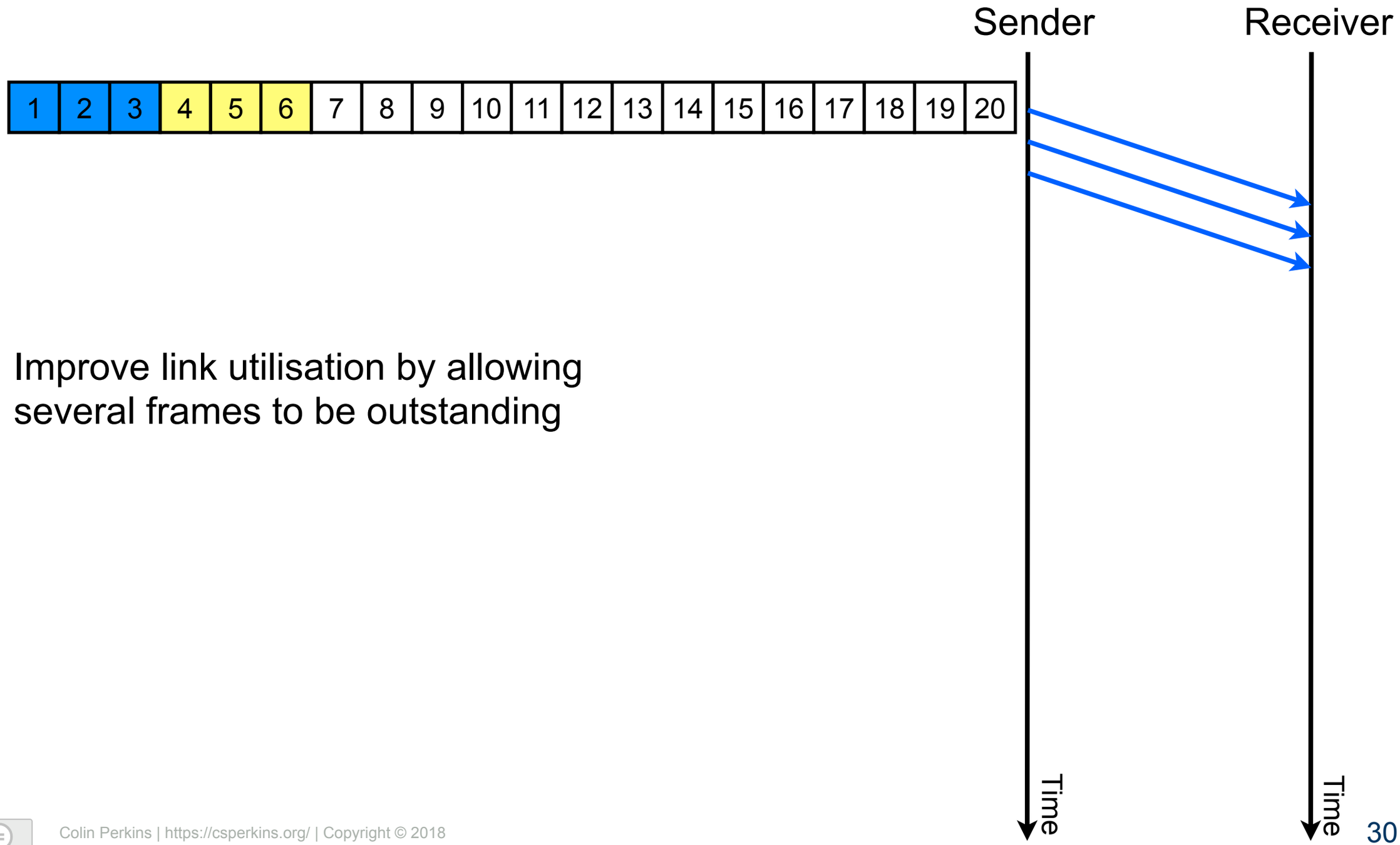
Sliding Window Protocol

Improve link utilisation by allowing several frames to be outstanding

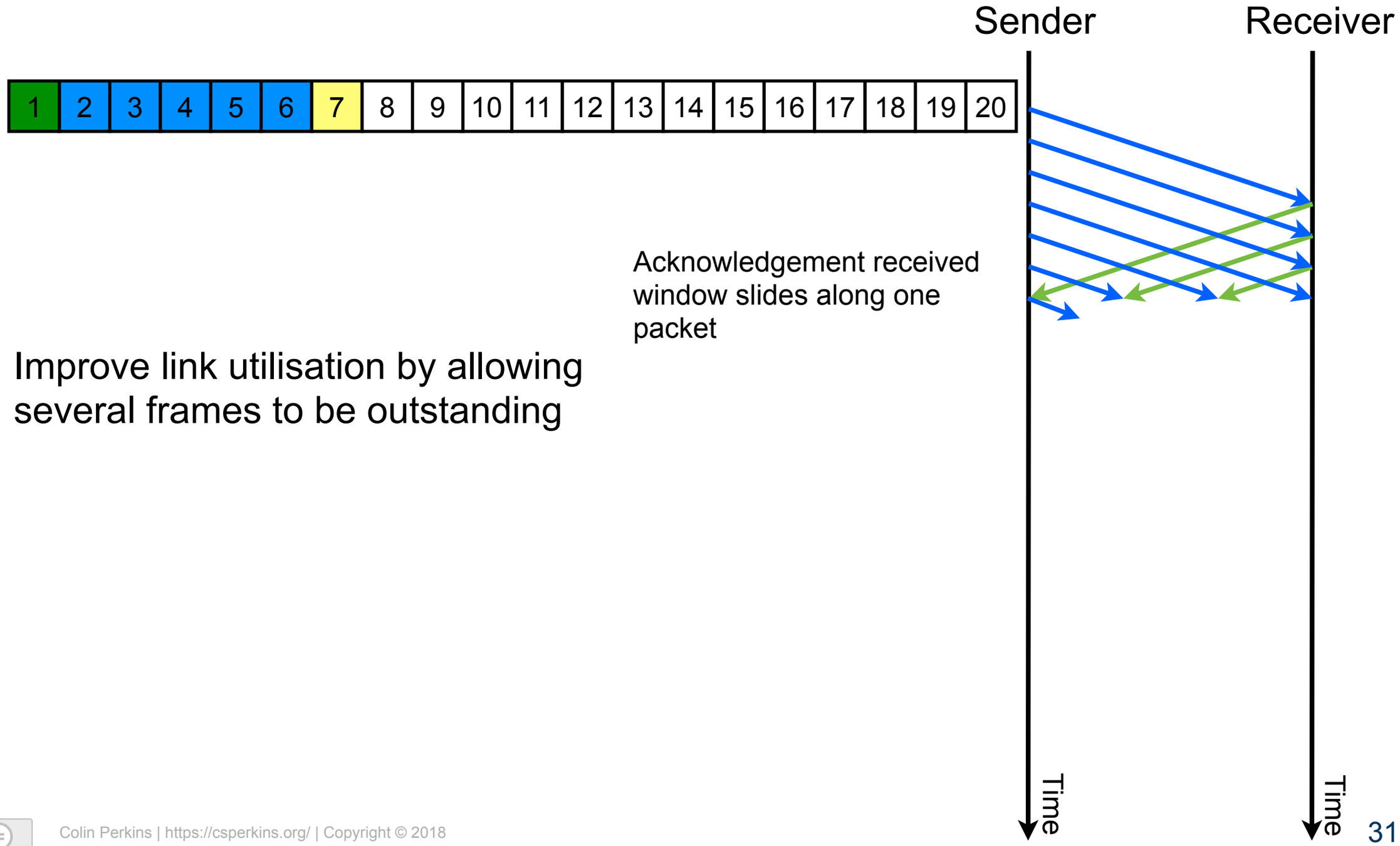


Sliding Window Protocol

Improve link utilisation by allowing several frames to be outstanding



Sliding Window Protocol

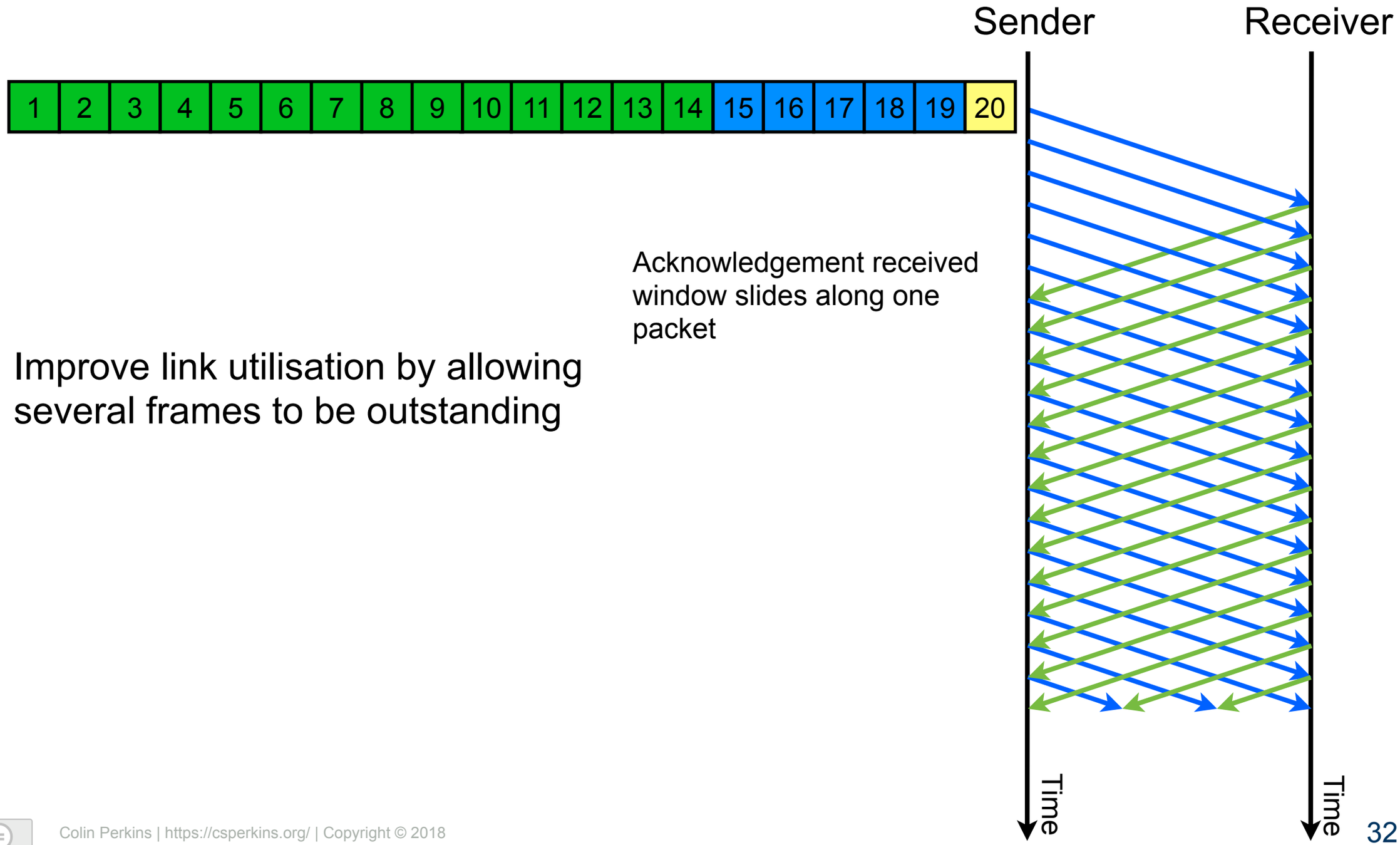


Improve link utilisation by allowing several frames to be outstanding

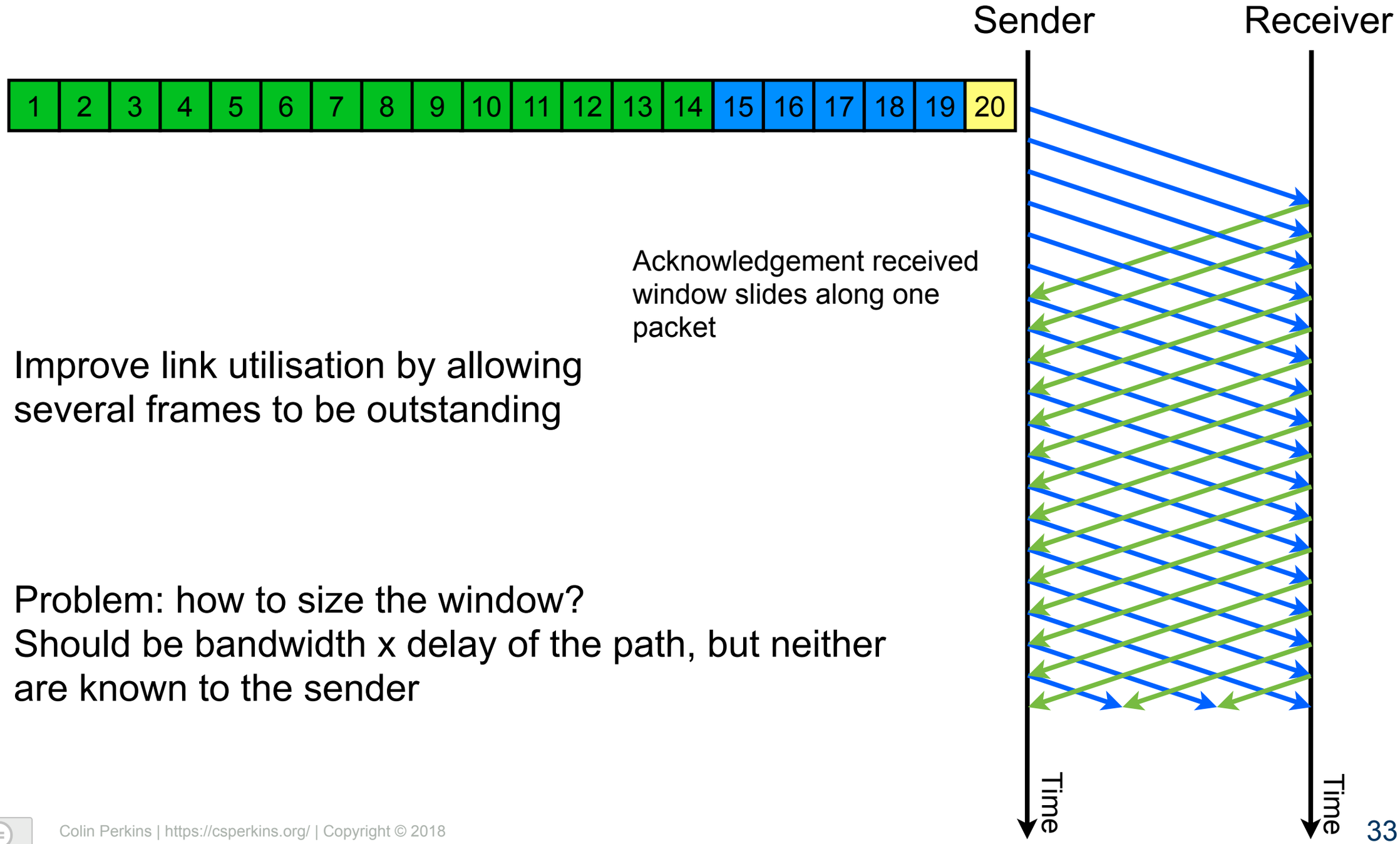
Sliding Window Protocol

Improve link utilisation by allowing several frames to be outstanding

Acknowledgement received window slides along one packet



Sliding Window Protocol



Acknowledgement received window slides along one packet

Improve link utilisation by allowing several frames to be outstanding

Problem: how to size the window?
Should be bandwidth x delay of the path, but neither are known to the sender

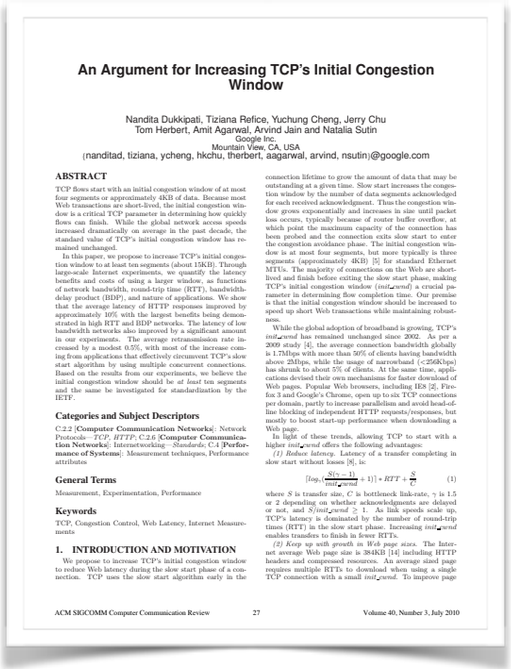
TCP Congestion Control

- A sliding window protocol for TCP:
  - How to choose initial window?
  - How to find the link capacity?
    - Slow start to estimate the bottleneck link capacity
    - Congestion avoidance to probe for changes in capacity

Choosing the Initial Window

- How to choose initial window size, *Winit*?
  - No information → need to measure path capacity
  - Start with a small window, increase until congestion
    - W_{init} of one packet per round-trip time is the only safe option, equivalent to stop-and-wait protocol, but is usually overly pessimistic
    - Traditionally, TCP used a slightly larger initial window: [RFC 3390]
      W_{init} = min(4 × MSS, max(2 × MSS, 4380 bytes)) packets per RTT
    - e.g., Ethernet with MTU = 1500 bytes, TCP/IP headers = 40 bytes gives W_{init} = min(4 × 1460, max(2 × 1460, 4380)) = 4380 bytes (~3 packets)
  - Modern TCP uses an initial window of 10 packets per RTT [RFC 6928]
    - Experimental, but data from Google shows network capacity has increased enough so this is likely safe

MSS = Maximum Segment Size (MTU minus TCP/IP header size)



An Argument for Increasing TCP's Initial Congestion Window

Nandita Dukkipati, Tiziana Refice, Yuchung Cheng, Jerry Chu
Tom Herbert, Amit Agarwal, Arvind Jain and Natalia Sutin
Google Inc.
Mountain View, CA, USA
{nanditad, tiziana, ycheng, hkchu, therbert, aagarwal, arvind, nsutin}@google.com

ABSTRACT

TCP flows start with an initial congestion window of at most four segments or approximately 4KB of data. Because most Web transactions are short-lived, the initial congestion window is a critical TCP parameter in determining how quickly flows can finish. While the global network access speeds increased dramatically on average in the past decade, the standard value of TCP's initial congestion window has remained unchanged.

In this paper, we propose to increase TCP's initial congestion window to at least ten segments (about 15KB). Through large-scale Internet experiments, we quantify the latency benefits and costs of using a larger window, as functions of network bandwidth, round-trip time (RTT), bandwidth-delay product (BDP), and nature of applications. We show that the average latency of HTTP responses improved by approximately 10% with the largest benefits being demonstrated in high RTT and BDP networks. The latency of low bandwidth networks also improved by a significant amount in our experiments. The average retransmission rate increased by a modest 0.5%, with most of the increase coming from applications that effectively circumvent TCP's slow start algorithm by using multiple concurrent connections. Based on the results from our experiments, we believe the initial congestion window should be *at least* ten segments and the same be investigated for standardization by the IETF.

Categories and Subject Descriptors

C.2.2 [**Computer Communication Networks**]: Network Protocols—*TCP, HTTP*; C.2.6 [**Computer Communication Networks**]: Internetworking—*Standards*; C.4 [**Performance of Systems**]: Measurement techniques, Performance attributes

General Terms

Measurement, Experimentation, Performance

Keywords

TCP, Congestion Control, Web Latency, Internet Measurements

1. INTRODUCTION AND MOTIVATION

We propose to increase TCP's initial congestion window to reduce Web latency during the slow start phase of a connection. TCP uses the slow start algorithm early in the connection lifetime to grow the amount of data that may be outstanding at a given time. Slow start increases the congestion window by the number of data segments acknowledged for each received acknowledgment. Thus the congestion window grows exponentially and increases in size until packet loss occurs, typically because of router buffer overflow, at which point the maximum capacity of the connection has been probed and the connection exits slow start to enter the congestion avoidance phase. The initial congestion window is at most four segments, but more typically is three segments (approximately 4KB) [5] for standard Ethernet MTUs. The majority of connections on the Web are short-lived and finish before exiting the slow start phase, making TCP's initial congestion window (*init_cwnd*) a crucial parameter in determining flow completion time. Our premise is that the initial congestion window should be increased to speed up short Web transactions while maintaining robustness.

While the global adoption of broadband is growing, TCP's *init_cwnd* has remained unchanged since 2002. As per a 2009 study [4], the average connection bandwidth globally is 1.7Mbps with more than 50% of clients having bandwidth above 2Mbps, while the usage of narrowband (<256Kbps) has shrunk to about 5% of clients. At the same time, applications devised their own mechanisms for faster download of Web pages. Popular Web browsers, including IE8 [2], Firefox 3 and Google's Chrome, open up to six TCP connections per domain, partly to increase parallelism and avoid head-of-line blocking of independent HTTP requests/responses, but mostly to boost start-up performance when downloading a Web page.

In light of these trends, allowing TCP to start with a higher *init_cwnd* offers the following advantages:

(1) Reduce latency. Latency of a transfer completing in slow start without losses [8], is:

$$\lceil \log_\gamma (\frac{S(\gamma-1)}{init\_cwnd} + 1) \rceil * RTT + \frac{S}{C} \quad (1)$$

where S is transfer size, C is bottleneck link-rate, γ is 1.5 or 2 depending on whether acknowledgments are delayed or not, and $S/init\_cwnd \geq 1$. As link speeds scale up, TCP's latency is dominated by the number of round-trip times (RTT) in the slow start phase. Increasing *init_cwnd* enables transfers to finish in fewer RTTs.

(2) Keep up with growth in Web page sizes. The Internet average Web page size is 384KB [14] including HTTP headers and compressed resources. An average sized page requires multiple RTTs to download when using a single TCP connection with a small *init_cwnd*. To improve page

ACM SIGCOMM Computer Communication Review 27 Volume 40, Number 3, July 2010

N. Dukkipati, T. Refice, Y. Cheng, J. Chu, T. Herbert, A. Agarwal, A. Jain, and N. Sutin. An argument for increasing TCP's initial congestion window. Computer Communication Review, 40(3):27–33, July 2010. http://dx.doi.org/10.1145/1823844.1823848

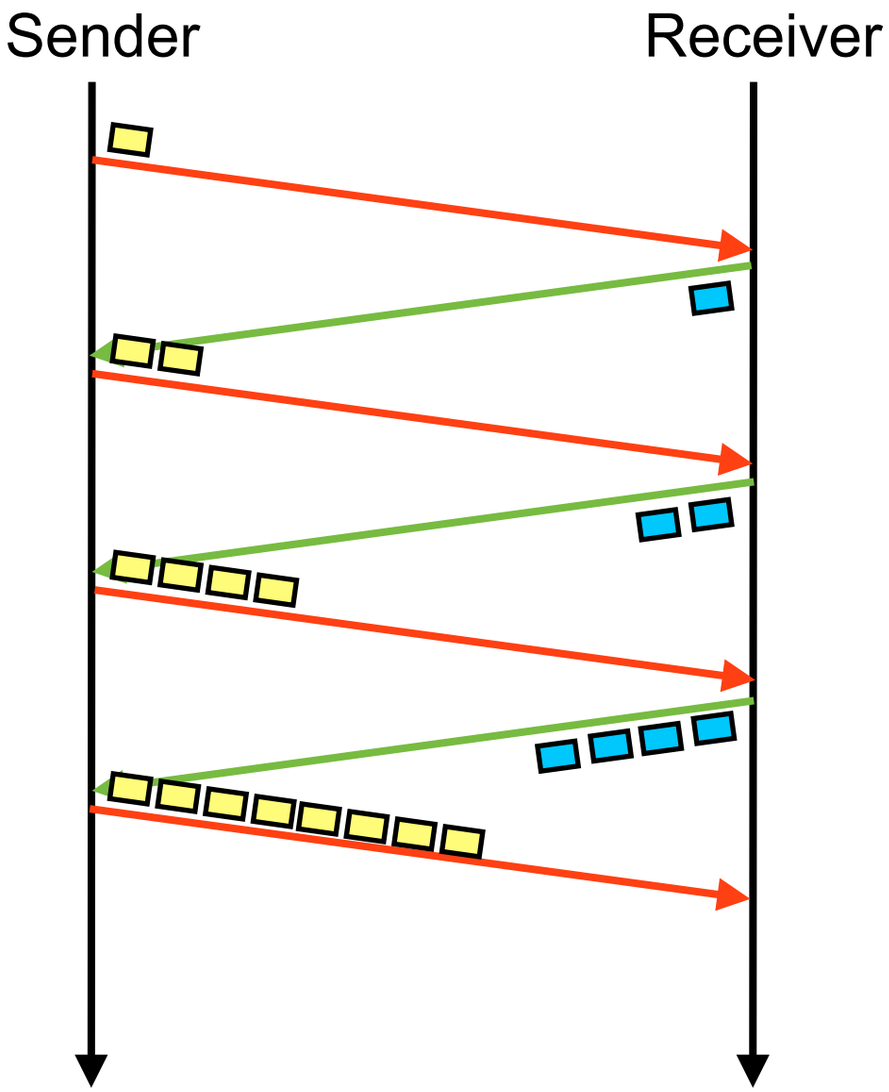
Finding the Link Capacity

- The initial window allows you to send
- How to choose the right window size to match the link capacity? Two issues:
  - How to find the correct window for the path when a new connection starts – *slow start*
  - How to adapt to changes in the available capacity once a connection is running – *congestion avoidance*

Slow Start

- Initial window, *Winit* = 1 packet per RTT
  - Or similar… a “slow start” to the connection
- Need to rapidly increase to the correct value for the network
  - Each acknowledgement for new data increases the window by 1 packet per RTT
  - On packet loss, immediately stop increasing window

Slow Start



- Two packets for each acknowledgement
- The window doubles on every round trip time – until loss occurs
- Rapidly finds the correct window size for the path

Congestion Avoidance

- Congestion avoidance mode used to probe for changes in network capacity
  - E.g., is sharing a connection with other traffic, and that traffic stops, meaning the available capacity increases
- Window increased by 1 packet per RTT
  - Slow, additive increase in window: $w_i = w_{i-1} + 1$
  - Until congestion is observed → respond to loss

Detecting Congestion

- TCP uses cumulative positive ACKs → two ways to detect congestion
  - Triple duplicate ACK → packet lost due to congestion
  - ACKs stop arriving → no data reaching receiver; link has failed completely somewhere
    - How long to wait before assuming ACKs have stopped?
    - T_{rto} = max(1 second, average *RTT* + (4 x *RTT* variance))
      - Statistical theory: 99.99% of data lies with 4σ of the mean, assuming normal distribution (where variance of the distribution = σ2)

Responding to Congestion

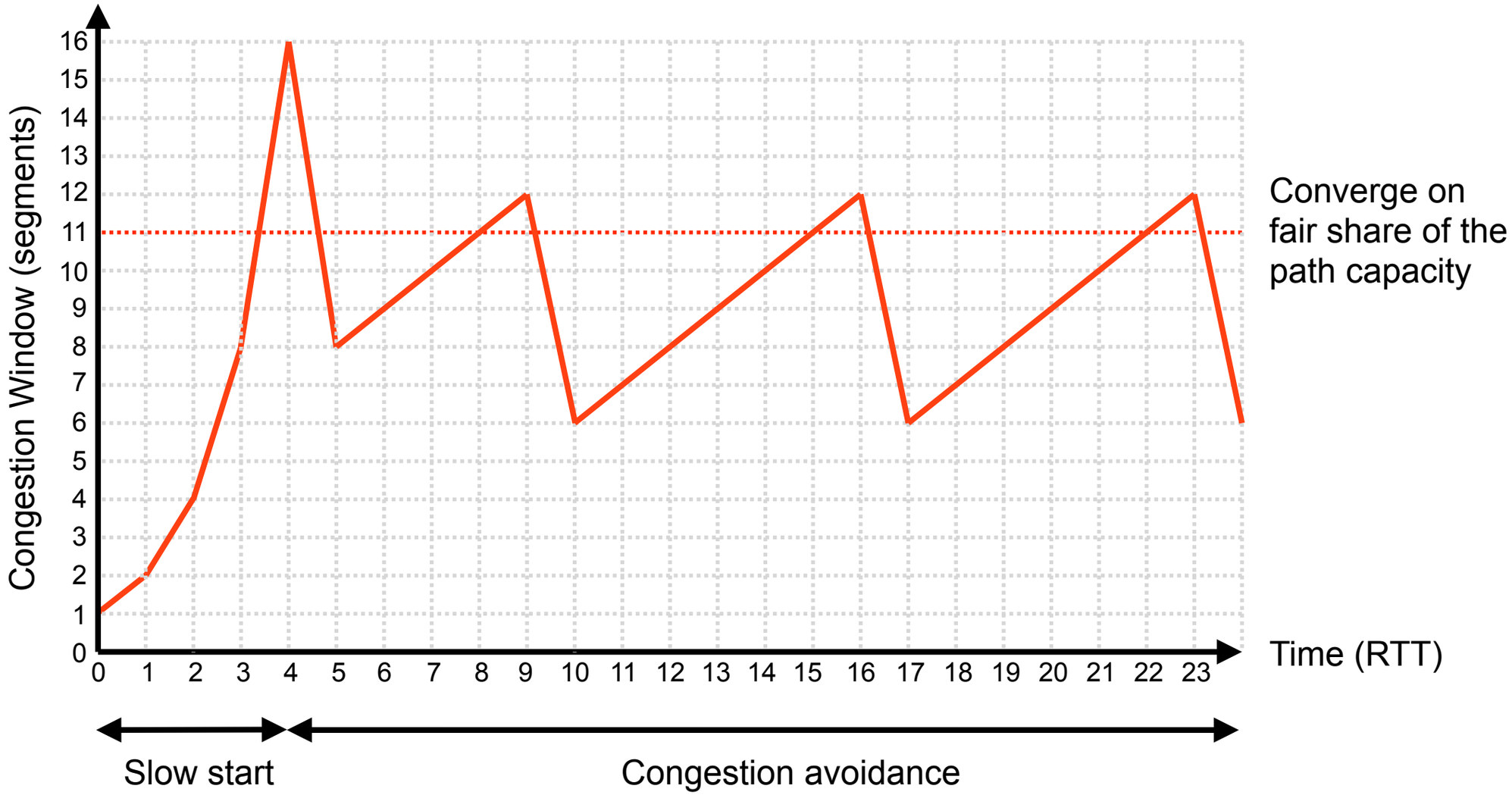
- If loss detected by triple-duplicate ACK:
  - Transient congestion, but data still being received
  - Multiplicative decrease in window: $w_i = w_{i-1} \times 0.5$
  - Rapid reduction in sending speed allows congestion to clear quickly, avoids congestion collapse

Responding to Congestion

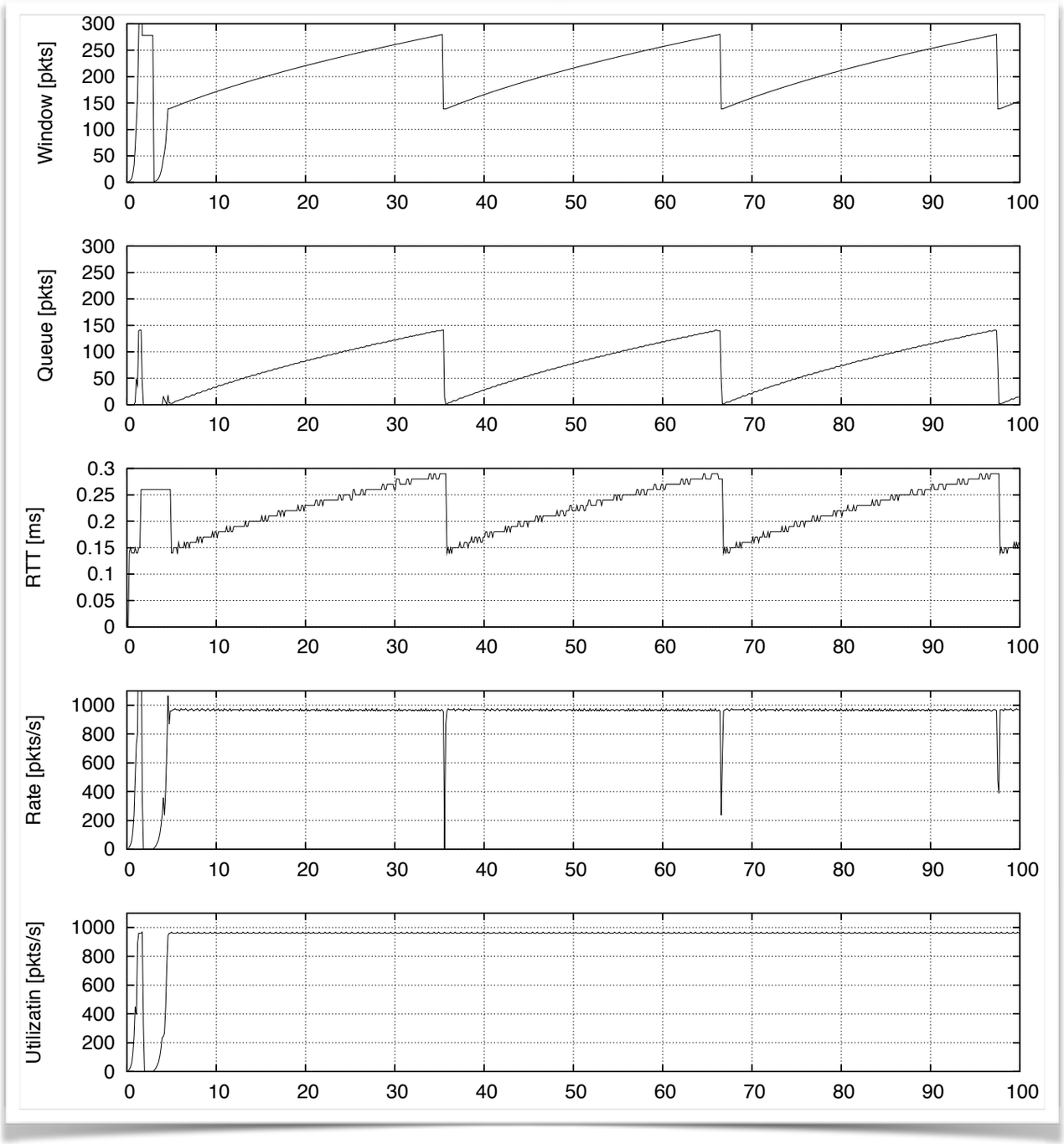
- If loss detected by time-out:
  - No packets received for a long period of time – likely a significant problem with network (e.g., link failed)
  - Return to initial sending window, and probe for the new capacity using slow start
  - Assume the route has changed, and you know nothing about the new path

Congestion Window Evolution

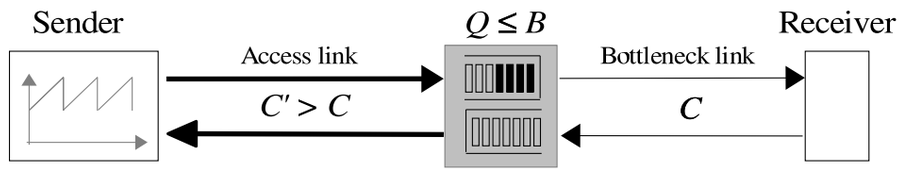
Typical evolution of TCP window, assuming W_{init} = 1



Congestion Window Evolution, Buffering, and Throughput



Source: G. Appenzeller, “Sizing Router Buffers”, PhD thesis, Stanford University, March 2005. http://tiny-tera.stanford.edu/~nickm/papers/guido-thesis.pdf (Figures 2.1 and 2.2)



buffer size = bandwidth × delay

- Bottleneck queue never empty
- Bottleneck link never becomes idle → sending rate varies, but receiver sees continuous flow

Performance and Limitations of TCP

- TCP congestion control highly effective at keeping bottleneck link fully utilised
  - Provided sufficient buffering in the network: buffer size = bandwidth × delay
  - Packets queued in buffer → delay
  - TCP trades some extra delay to ensure high throughput
- Unless ECN used, TCP assumes loss is due to congestion
  - Too much traffic queued at an intermediate link → some packets dropped
  - This is not always true:
    - Wireless networks
    - High-speed long-distance optical networks
  - Much research into improved versions of TCP for wireless links

Summary

- Congestion control principles
  - Conservation of packets
  - Additive increase, multiplicative decrease (AIMD)
- TCP congestion control
  - Slow start
  - Congestion avoidance
  - AIMD