6.1800 Spring 2025

Lecture #11: Reliable Transport

adding reliability while also keeping things efficient and fair

Katrina LaCurts | lacurts@mit.edu | 6.1800 2025

6.1800 in the news

Home / NPR News

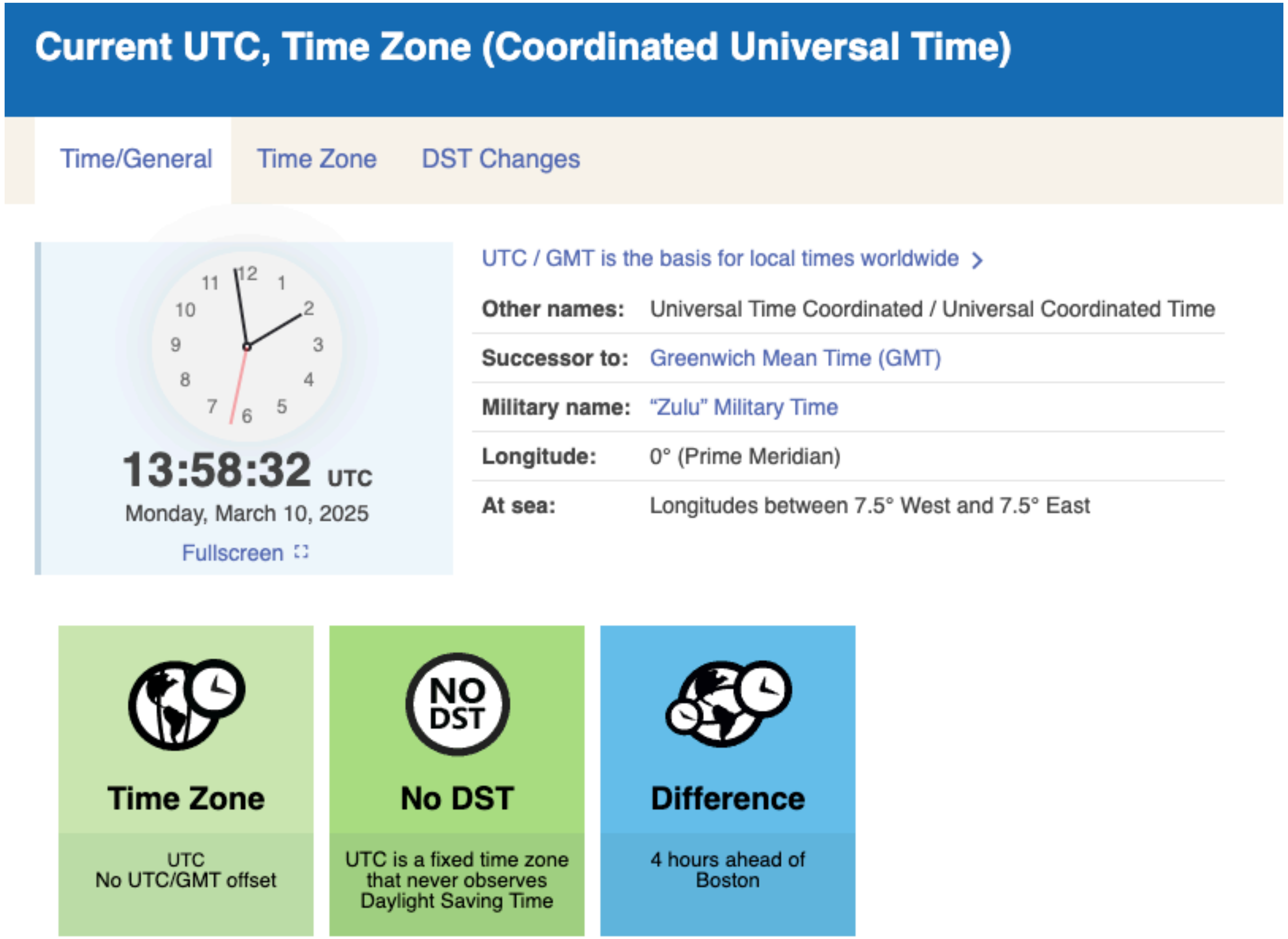
Daylight saving time has started. Here's how to adjust

March 09, 2025 By Sarah Boden

https://www.wbur.org/npr/nx-s1-5320160/time-change-daylight-saving-spring-health

6.1800 in the news

the majority of Internet standards use UTC, which doesn't observe Daylight Savings Time



Current UTC, Time Zone (Coordinated Universal Time)

Time/General | Time Zone | DST Changes

13:58:32 UTC

Monday, March 10, 2025

Fullscreen

UTC / GMT is the basis for local times worldwide >

Other names:	Universal Time Coordinated / Universal Coordinated Time
Successor to:	Greenwich Mean Time (GMT)
Military name:	"Zulu" Military Time
Longitude:	0° (Prime Meridian)
At sea:	Longitudes between 7.5° West and 7.5° East

Time Zone
UTC
No UTC/GMT offset

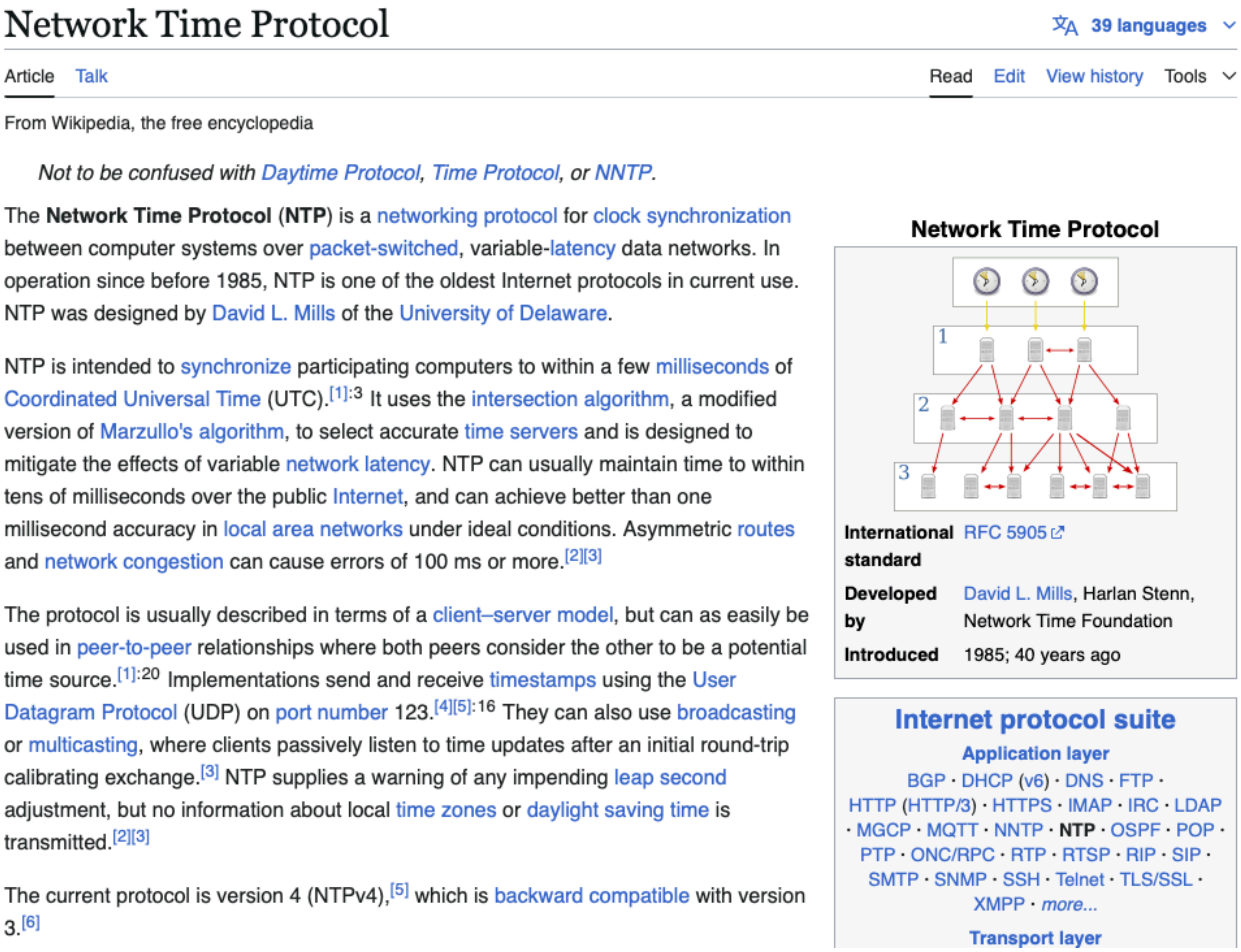
No DST
UTC is a fixed time zone that never observes Daylight Saving Time

Difference
4 hours ahead of Boston

https://www.timeanddate.com/worldclock/timezone/utc

6.1800 in the news

the **network time protocol** synchronizes clocks to UTC



Network Time Protocol

From Wikipedia, the free encyclopedia

Not to be confused with Daytime Protocol, Time Protocol, or NNTP.

The **Network Time Protocol** (**NTP**) is a networking protocol for clock synchronization between computer systems over packet-switched, variable-latency data networks. In operation since before 1985, NTP is one of the oldest Internet protocols in current use. NTP was designed by David L. Mills of the University of Delaware.

NTP is intended to synchronize participating computers to within a few milliseconds of Coordinated Universal Time (UTC).[1]:3 It uses the intersection algorithm, a modified version of Marzullo's algorithm, to select accurate time servers and is designed to mitigate the effects of variable network latency. NTP can usually maintain time to within tens of milliseconds over the public Internet, and can achieve better than one millisecond accuracy in local area networks under ideal conditions. Asymmetric routes and network congestion can cause errors of 100 ms or more.[2][3]

The protocol is usually described in terms of a client–server model, but can as easily be used in peer-to-peer relationships where both peers consider the other to be a potential time source.[1]:20 Implementations send and receive timestamps using the User Datagram Protocol (UDP) on port number 123.[4][5]:16 They can also use broadcasting or multicasting, where clients passively listen to time updates after an initial round-trip calibrating exchange.[3] NTP supplies a warning of any impending leap second adjustment, but no information about local time zones or daylight saving time is transmitted.[2][3]

The current protocol is version 4 (NTPv4),[5] which is backward compatible with version 3.[6]

Network Time Protocol

International standard	RFC 5905
Developed by	David L. Mills, Harlan Stenn, Network Time Foundation
Introduced	1985; 40 years ago

Internet protocol suite

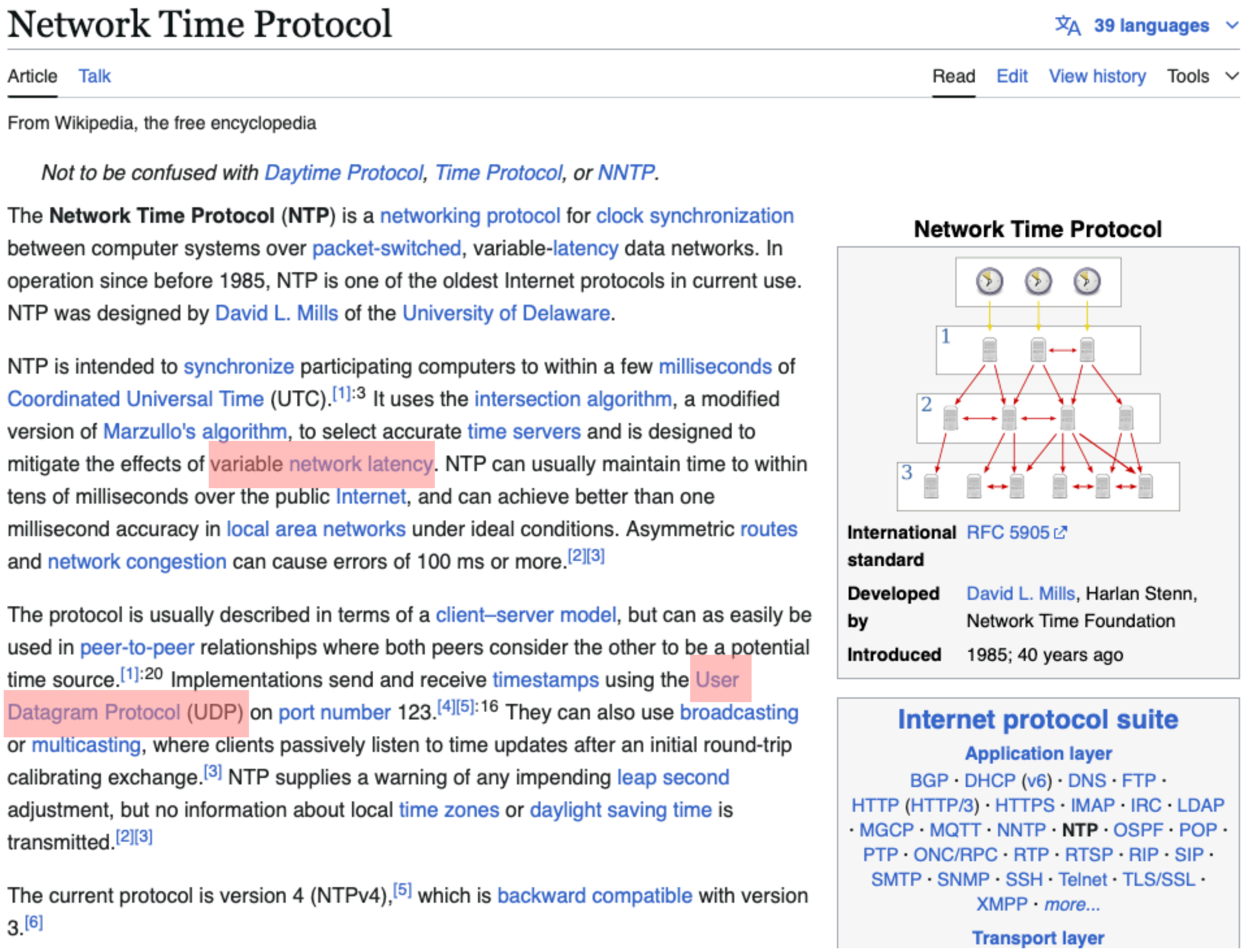
Application layer

BGP · DHCP (v6) · DNS · FTP · HTTP (HTTP/3) · HTTPS · IMAP · IRC · LDAP · MGCP · MQTT · NNTP · **NTP** · OSPF · POP · PTP · ONC/RPC · RTP · RTSP · RIP · SIP · SMTP · SNMP · SSH · Telnet · TLS/SSL · XMPP · *more...*

Transport layer

6.1800 in the news

the **network time protocol** synchronizes clocks to UTC



Network Time Protocol

39 languages

Article Talk | Read Edit View history Tools

From Wikipedia, the free encyclopedia

Not to be confused with Daytime Protocol, Time Protocol, or NNTP.

The **Network Time Protocol** (**NTP**) is a networking protocol for clock synchronization between computer systems over packet-switched, variable-latency data networks. In operation since before 1985, NTP is one of the oldest Internet protocols in current use. NTP was designed by David L. Mills of the University of Delaware.

NTP is intended to synchronize participating computers to within a few milliseconds of Coordinated Universal Time (UTC).[1]:3 It uses the intersection algorithm, a modified version of Marzullo's algorithm, to select accurate time servers and is designed to mitigate the effects of variable network latency. NTP can usually maintain time to within tens of milliseconds over the public Internet, and can achieve better than one millisecond accuracy in local area networks under ideal conditions. Asymmetric routes and network congestion can cause errors of 100 ms or more.[2][3]

The protocol is usually described in terms of a client–server model, but can as easily be used in peer-to-peer relationships where both peers consider the other to be a potential time source.[1]:20 Implementations send and receive timestamps using the User Datagram Protocol (UDP) on port number 123.[4][5]:16 They can also use broadcasting or multicasting, where clients passively listen to time updates after an initial round-trip calibrating exchange.[3] NTP supplies a warning of any impending leap second adjustment, but no information about local time zones or daylight saving time is transmitted.[2][3]

The current protocol is version 4 (NTPv4),[5] which is backward compatible with version 3.[6]

Network Time Protocol

International standard	RFC 5905
Developed by	David L. Mills, Harlan Stenn, Network Time Foundation
Introduced	1985; 40 years ago

Internet protocol suite

Application layer

BGP · DHCP (v6) · DNS · FTP · HTTP (HTTP/3) · HTTPS · IMAP · IRC · LDAP · MGCP · MQTT · NNTP · **NTP** · OSPF · POP · PTP · ONC/RPC · RTP · RTSP · RIP · SIP · SMTP · SNMP · SSH · Telnet · TLS/SSL · XMPP · *more...*

Transport layer

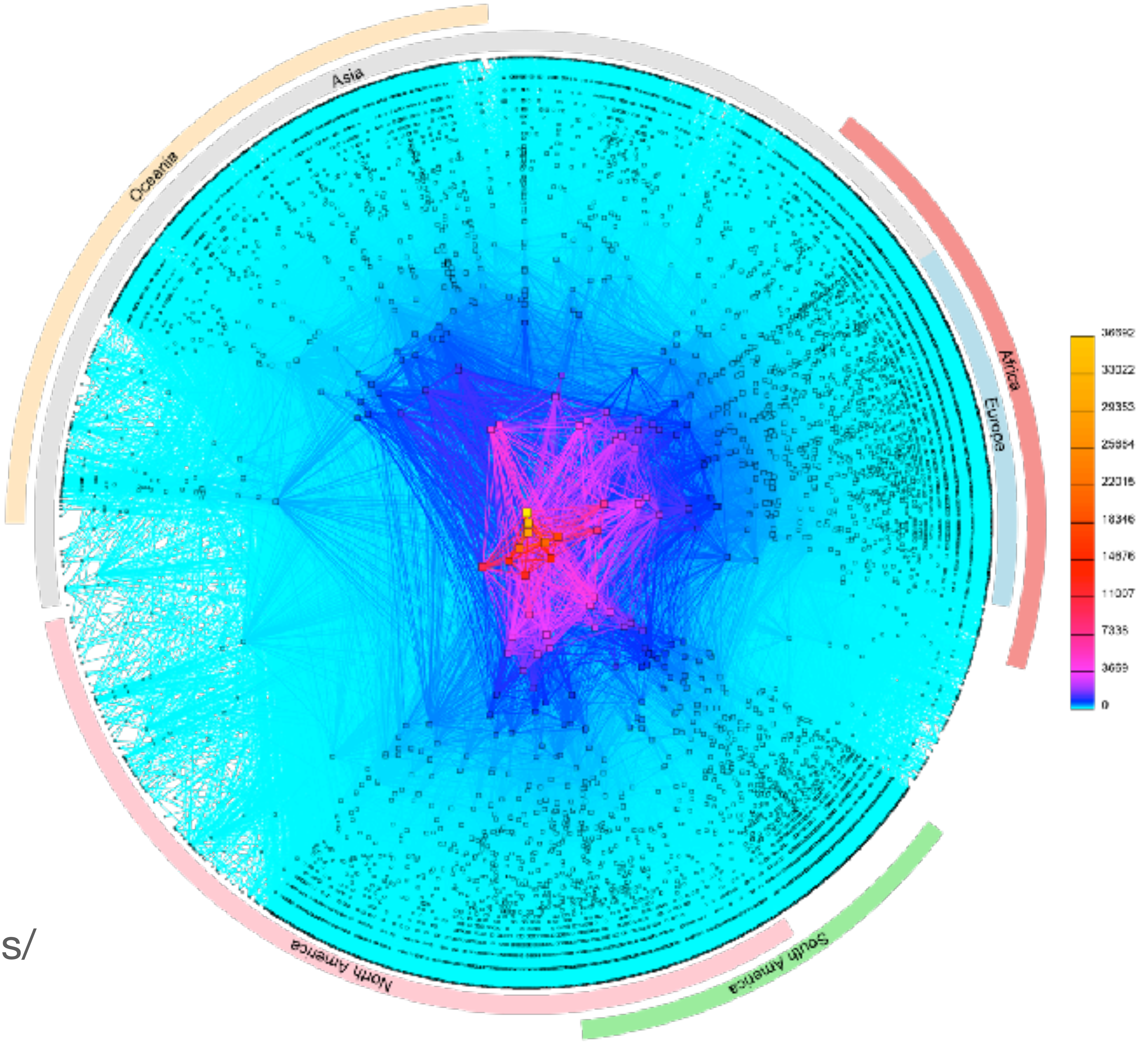
1970s: ARPAnet — hosts.txt

1978: flexibility and layering — distance-vector routing

early 80s: growth → change — **TCP**, UDP; OSPF, EGP, DNS

late 80s: growth → problems — congestion collapse (which led to **congestion control**); policy routing; CIDR

1993: commercialization



CAIDA's IPv4 AS Core, January 2020 (https://www.caida.org/projects/cartography/as-core/2020/)

today: moving up to the transport layer to discuss **reliable transport**

application	the things that actually generate traffic
transport	sharing the network, reliability (or not) *examples: TCP, UDP*
network	naming, addressing, routing *examples: IP*
link	communication between two directly-connected nodes *examples: ethernet, bluetooth, 802.11 (wifi)*

our (first) goal today is to create a **reliable transport protocol**, which delivers each byte of data **exactly once, in-order**, to the receiving application

application

the things that actually generate traffic

transport

sharing the network, reliability (or not)

examples: TCP, UDP

network

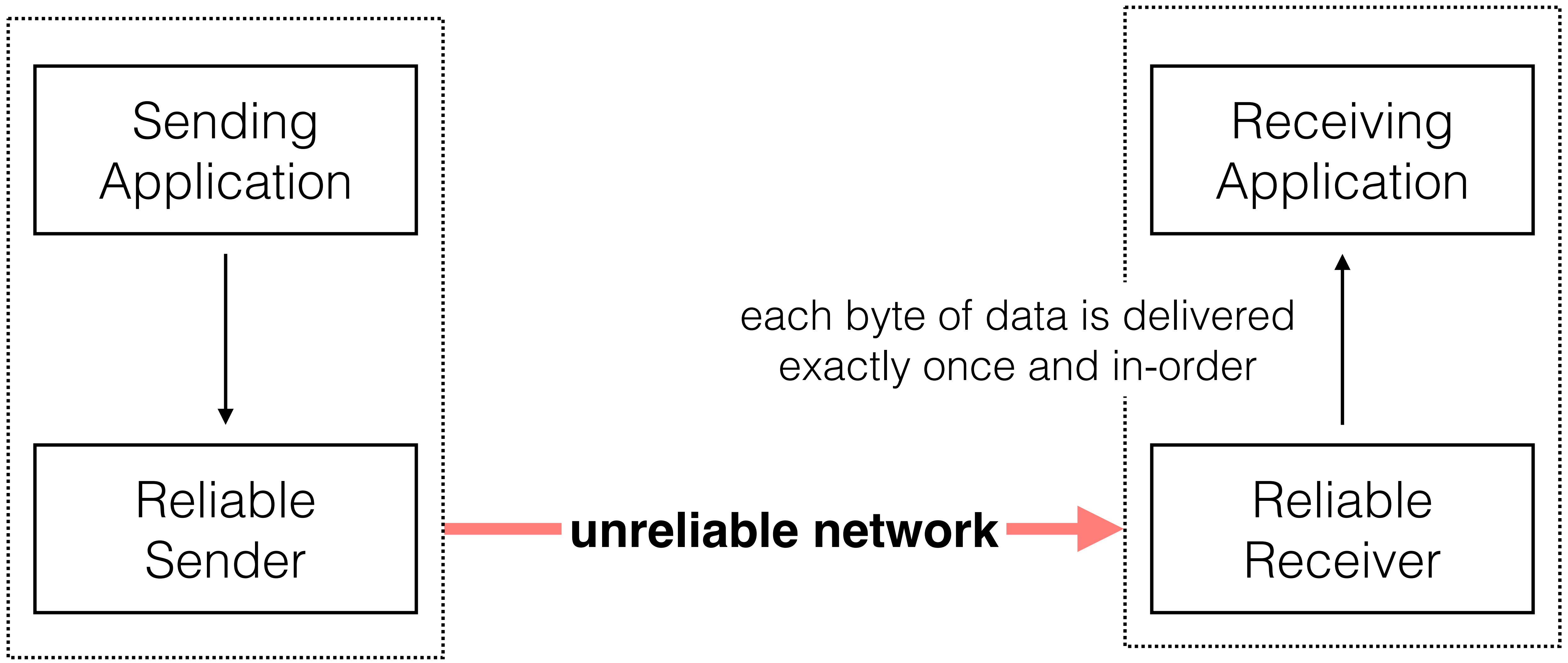
naming, addressing, routing

examples: IP

link

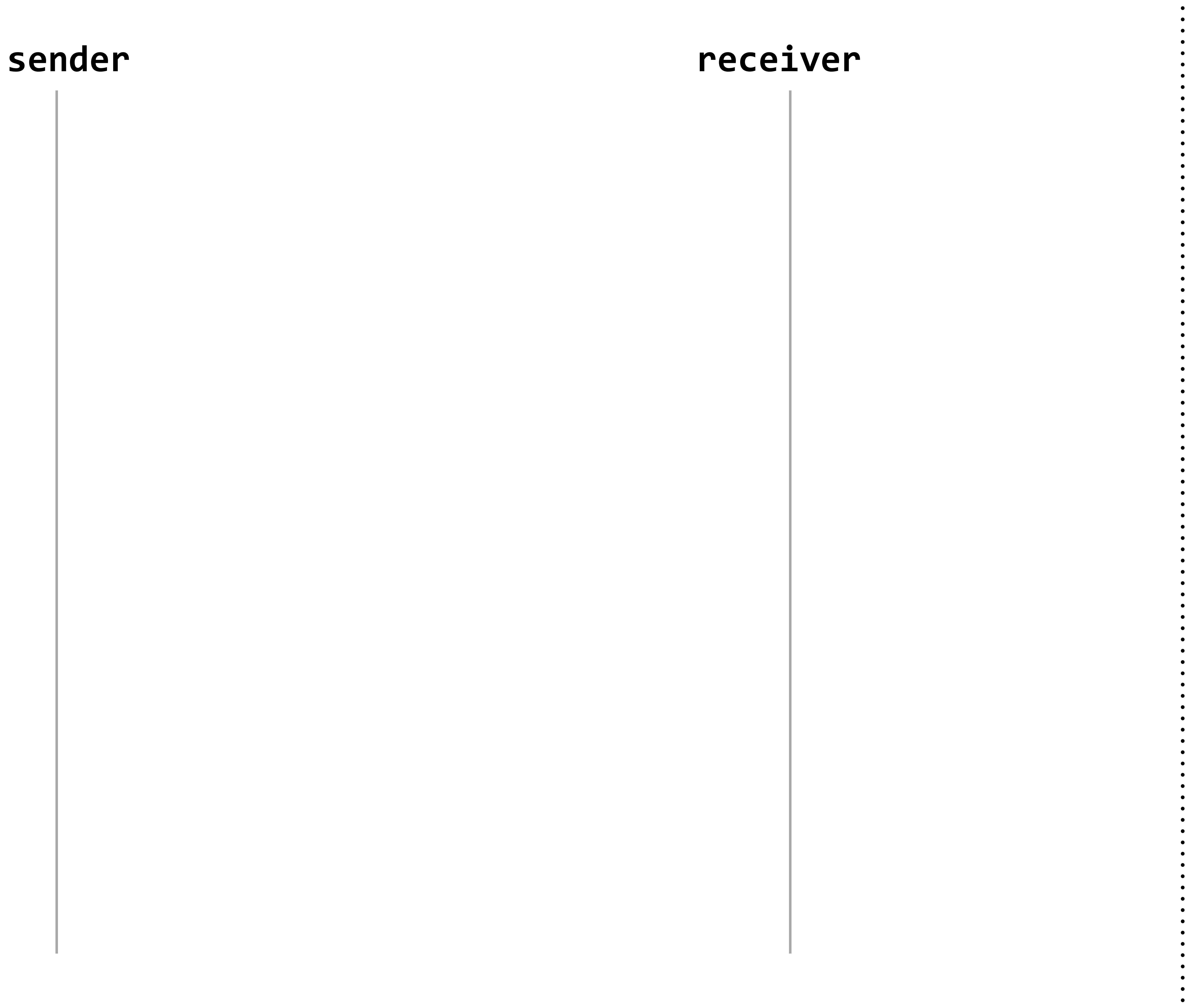
communication between two directly-connected nodes

examples: ethernet, bluetooth, 802.11 (wifi) our (first) goal today is to create a **reliable transport protocol**, which delivers each byte of data **exactly once, in-order**, to the receiving application

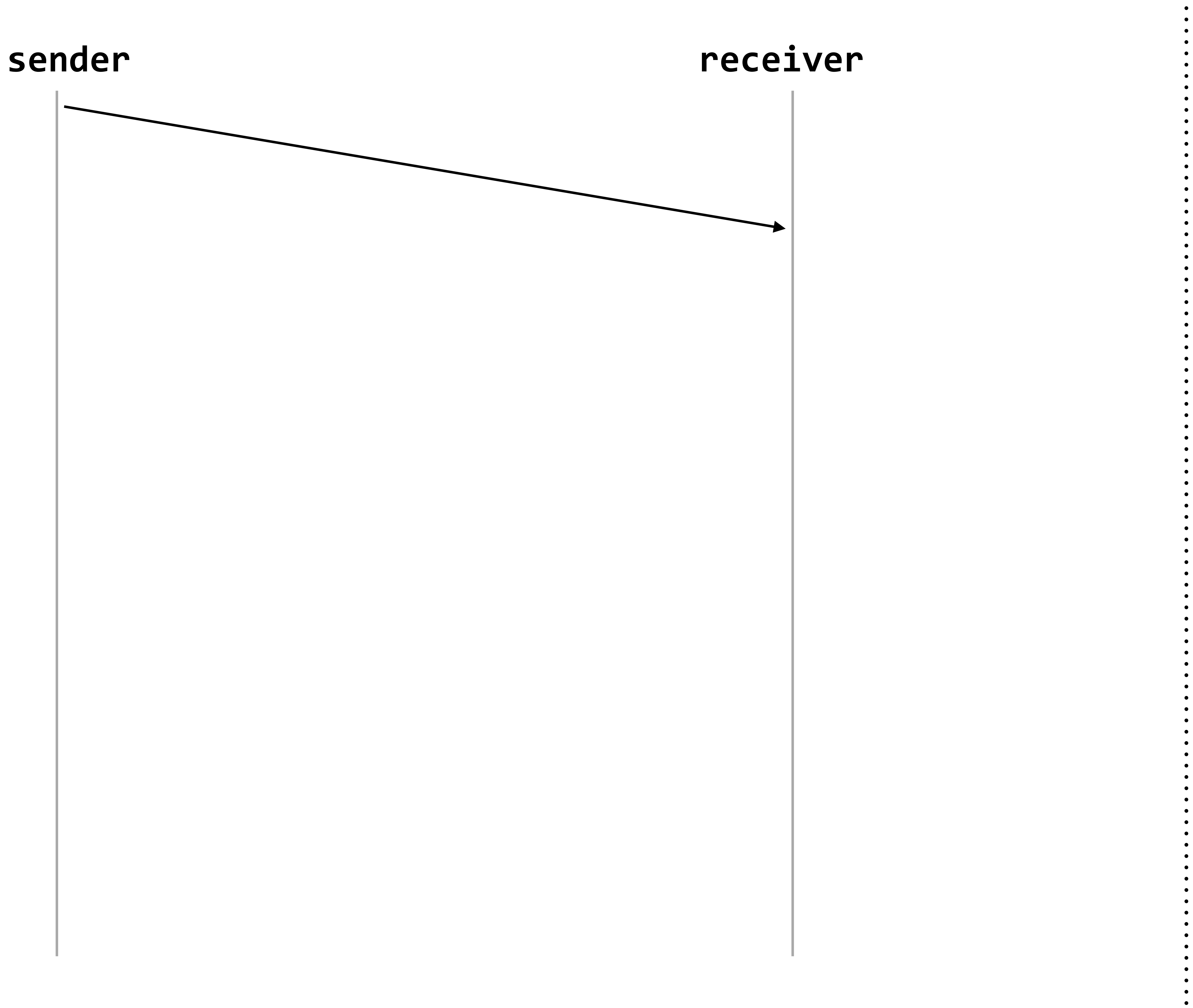


application	the things that actually generate traffic
transport	sharing the network, reliability (or not) *examples: TCP, UDP*
network	naming, addressing, routing *examples: IP*
link	communication between two directly-connected nodes *examples: ethernet, bluetooth, 802.11 (wifi)*

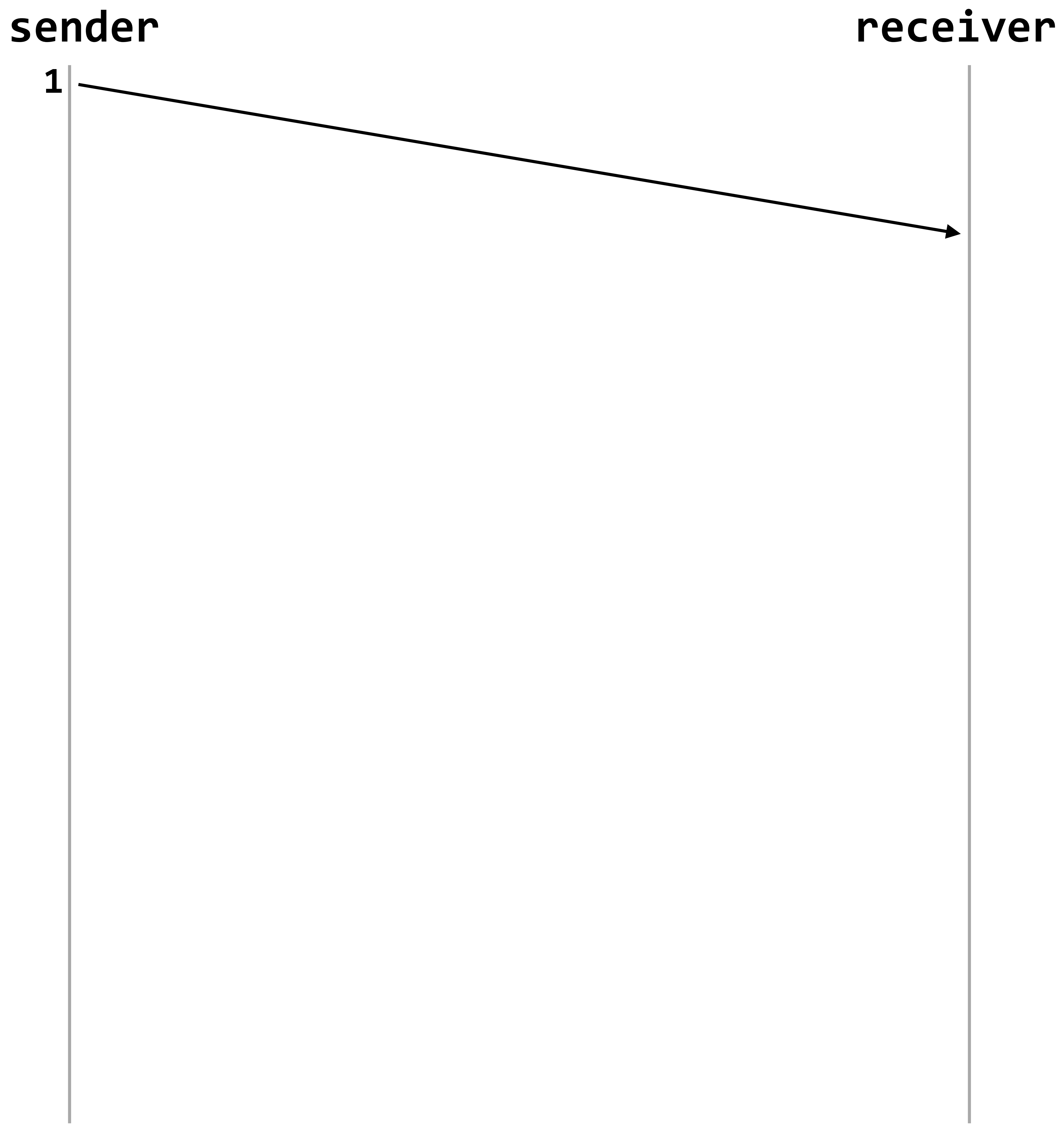
reliable transport protocols deliver each byte of data **exactly once, in-order**, to the receiving application



reliable transport protocols deliver each byte of data **exactly once, in-order**, to the receiving application

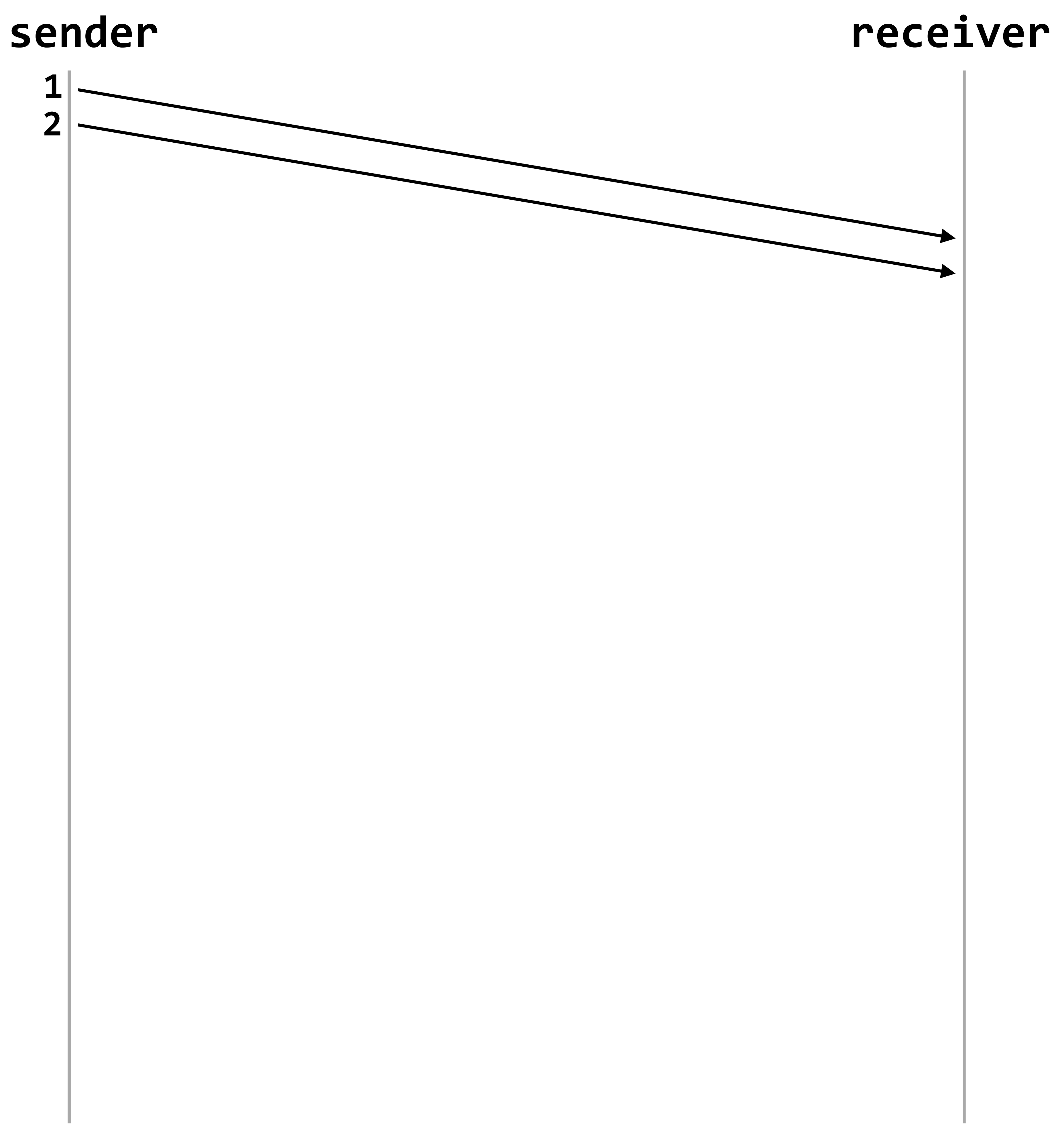


reliable transport protocols deliver each byte of data **exactly once, in-order**, to the receiving application



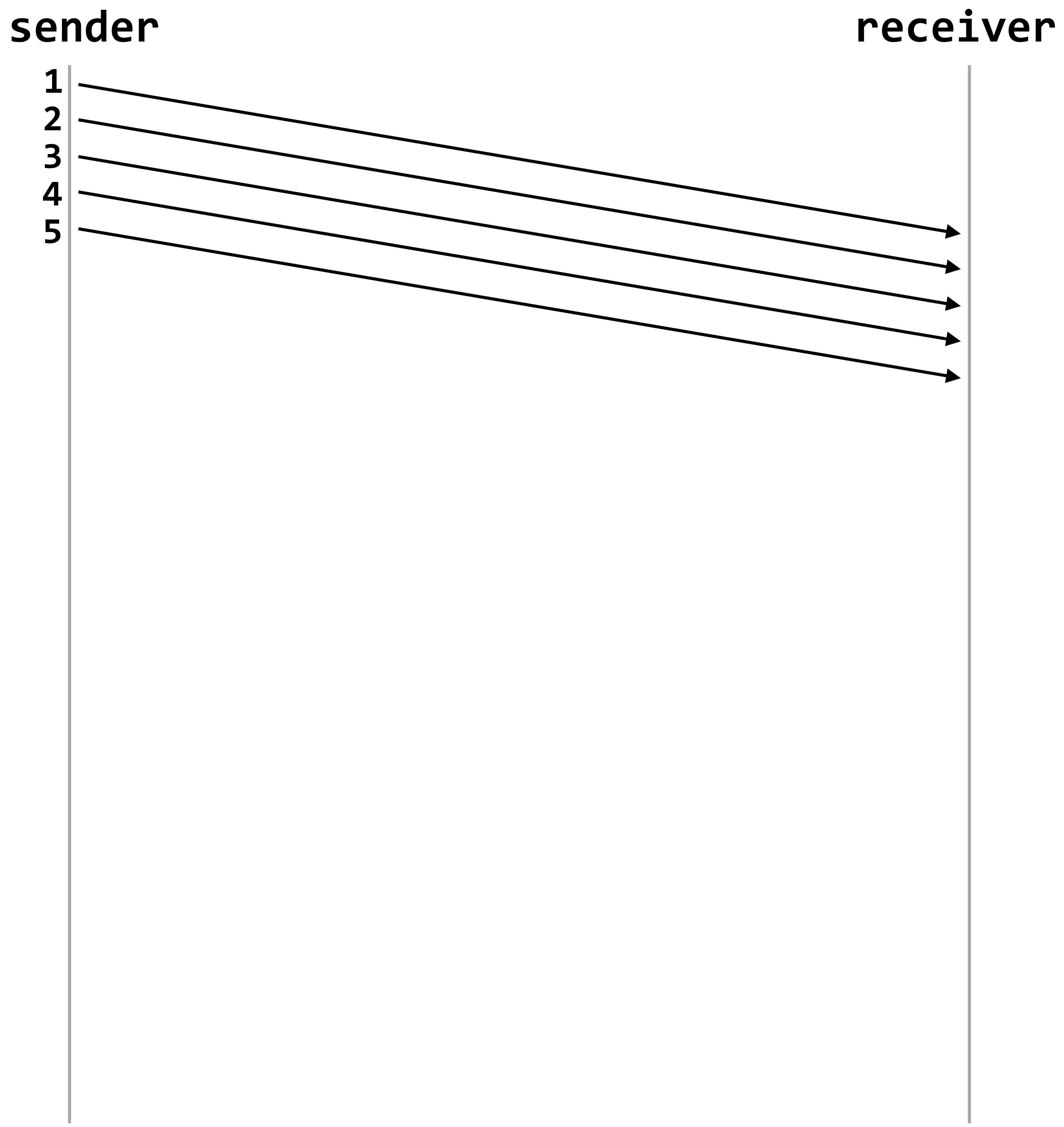
sequence numbers: used to order the packets

reliable transport protocols deliver each byte of data **exactly once, in-order**, to the receiving application



sequence numbers: used to order the packets

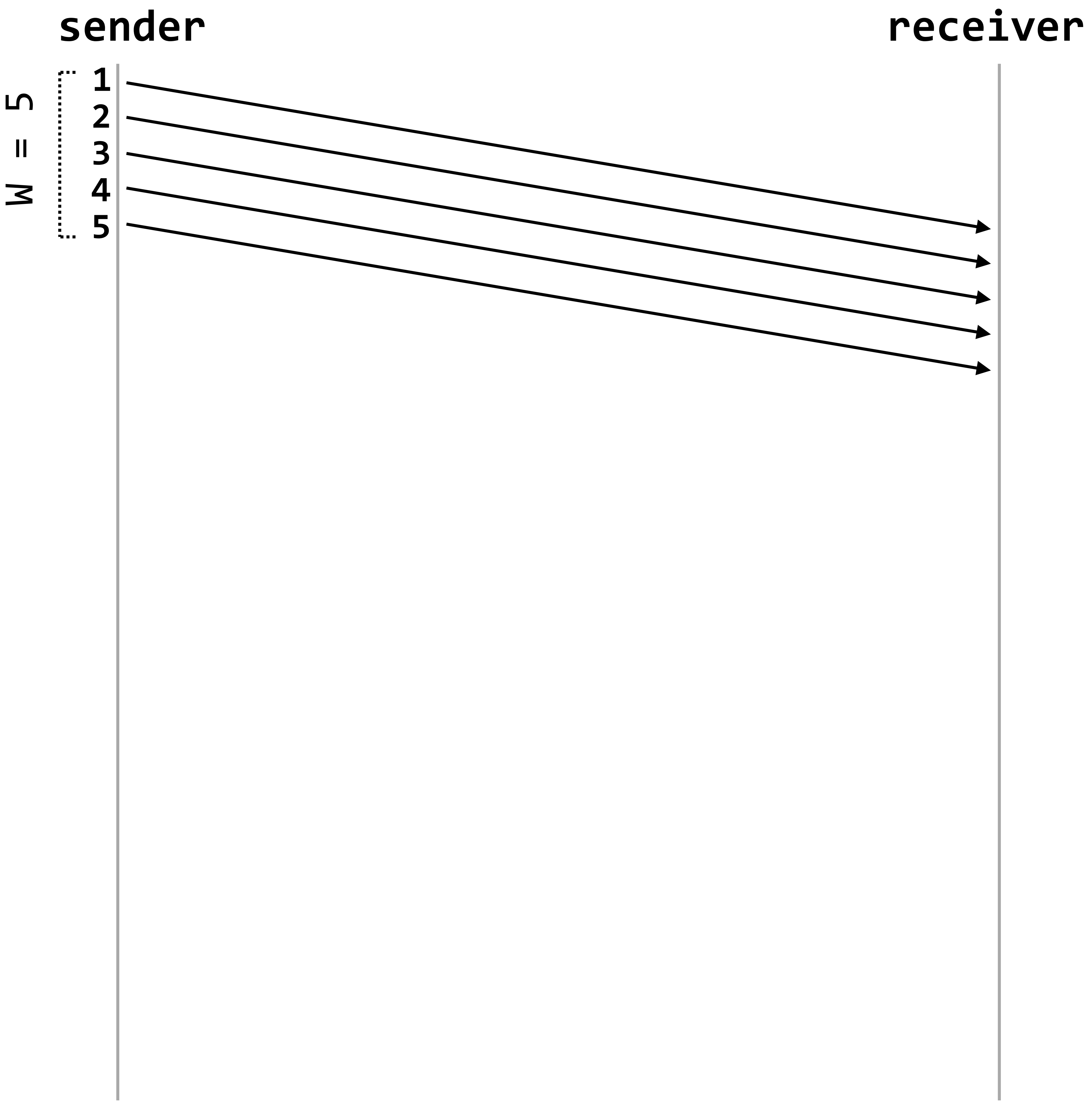
reliable transport protocols deliver each byte of data **exactly once, in-order**, to the receiving application



sequence numbers: used to order the packets

reliable transport protocols deliver each byte of data **exactly once, in-order**, to the receiving application

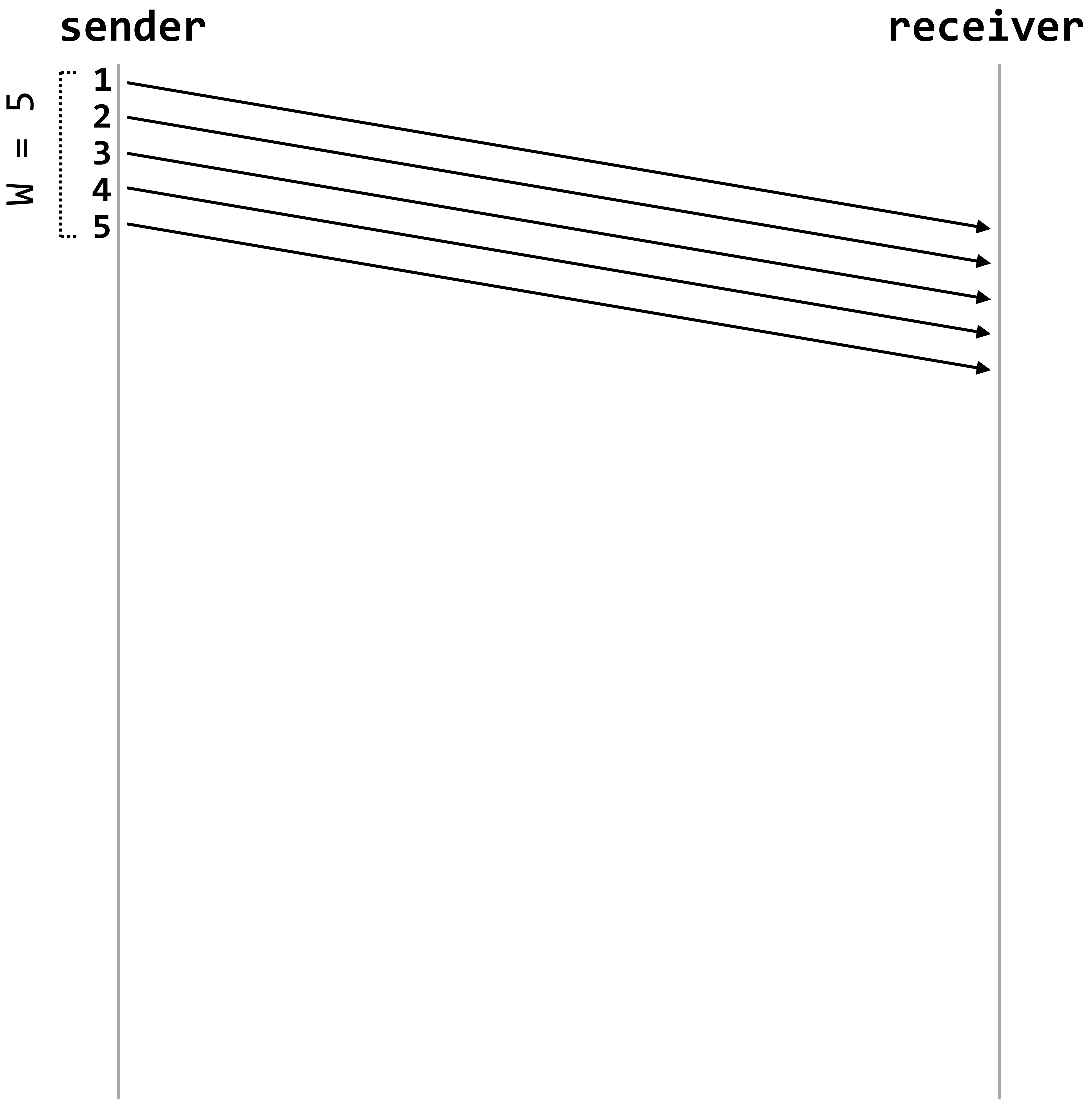
the sender is allowed to have W outstanding packets at once, but no more



sequence numbers: used to order the packets

reliable transport protocols deliver each byte of data **exactly once, in-order**, to the receiving application

the sender is allowed to have W outstanding packets at once, but no more



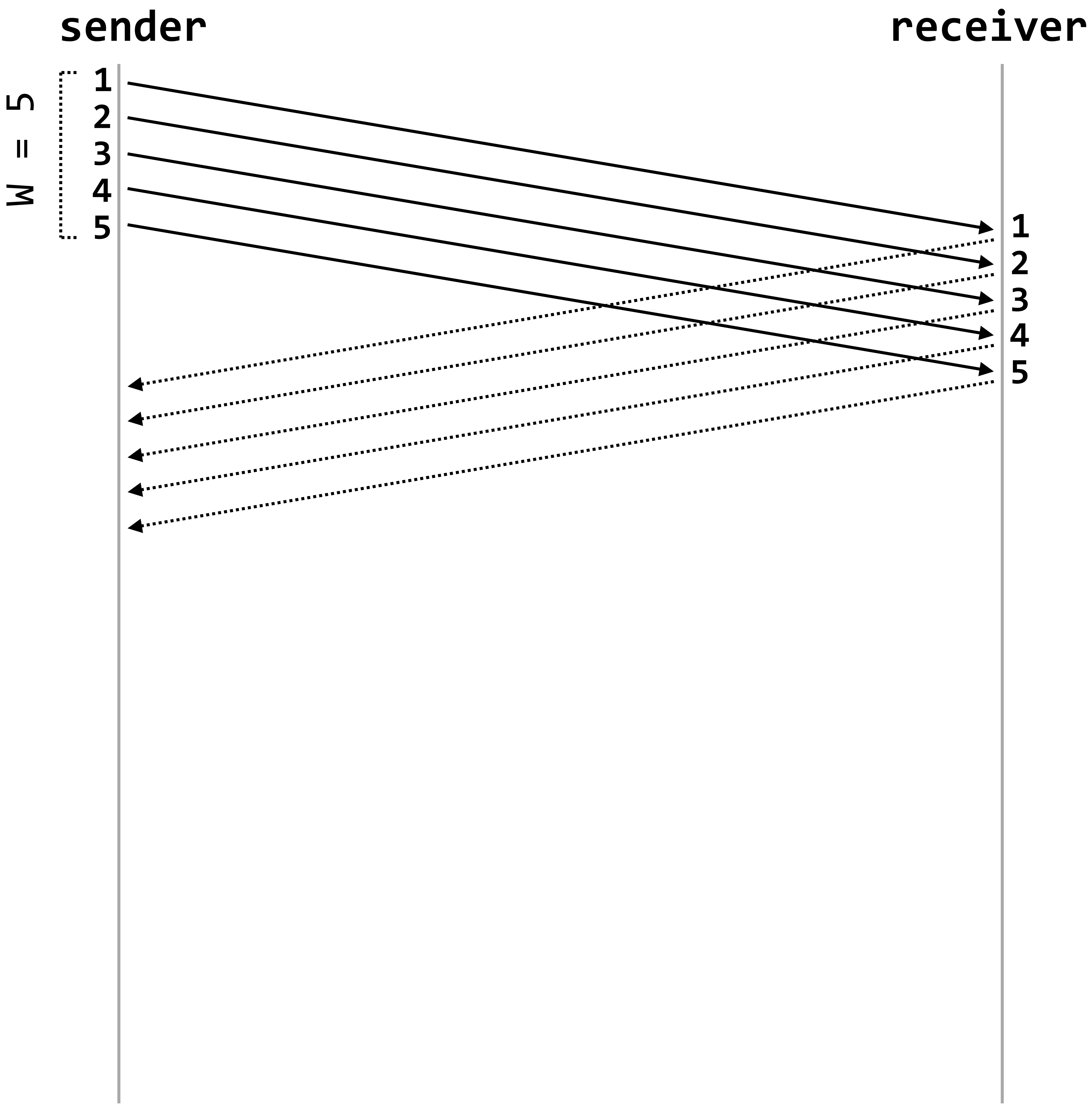
sequence numbers: used to order the packets

acknowledgments ("ACKs"): used to confirm that a packet has been received

an ACK with sequence number k indicates that the receiver has received **all packets up to and including k**

reliable transport protocols deliver each byte of data **exactly once, in-order**, to the receiving application

the sender is allowed to have W outstanding packets at once, but no more



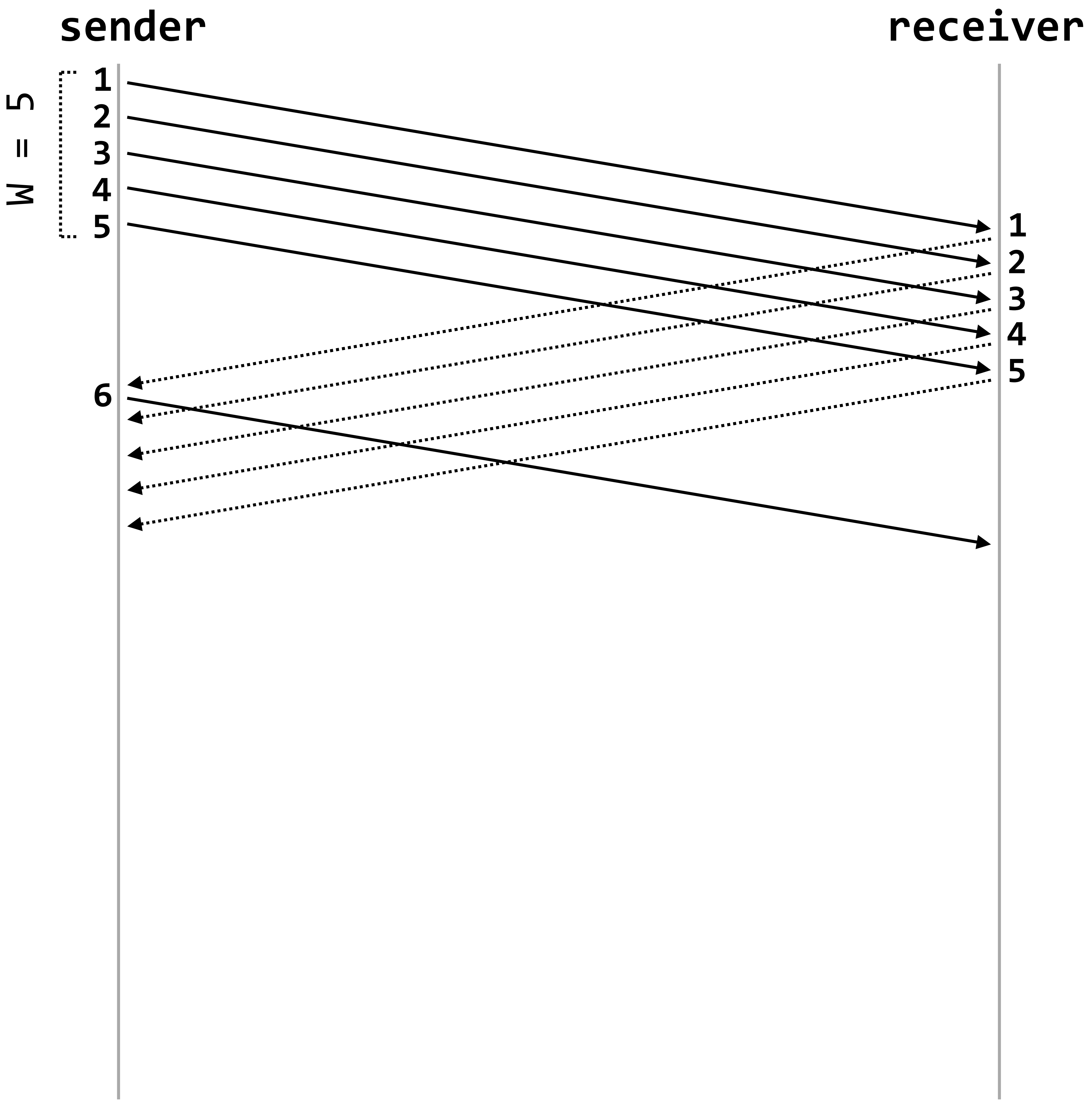
sequence numbers: used to order the packets

acknowledgments ("ACKs"): used to confirm that a packet has been received

an ACK with sequence number k indicates that the receiver has received **all packets up to and including k**

reliable transport protocols deliver each byte of data **exactly once, in-order**, to the receiving application

the sender is allowed to have W outstanding packets at once, but no more



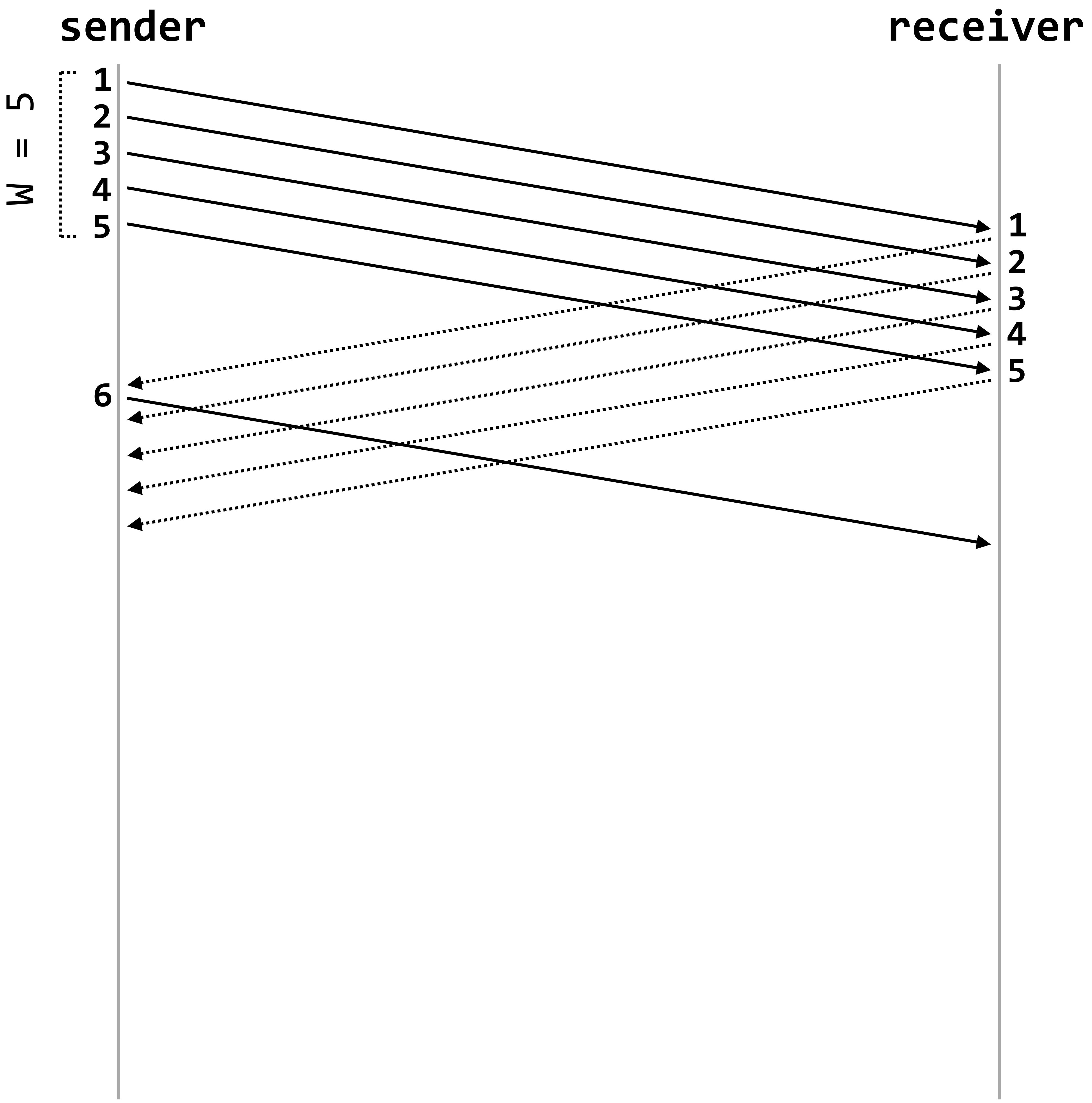
sequence numbers: used to order the packets

acknowledgments ("ACKs"): used to confirm that a packet has been received

an ACK with sequence number k indicates that the receiver has received **all packets up to and including k**

reliable transport protocols deliver each byte of data **exactly once, in-order**, to the receiving application

the sender is allowed to have W outstanding packets at once, but no more



this is known as a **sliding-window protocol**

the **window** of outstanding (un-ACKed) packets **slides** along the sequence number space

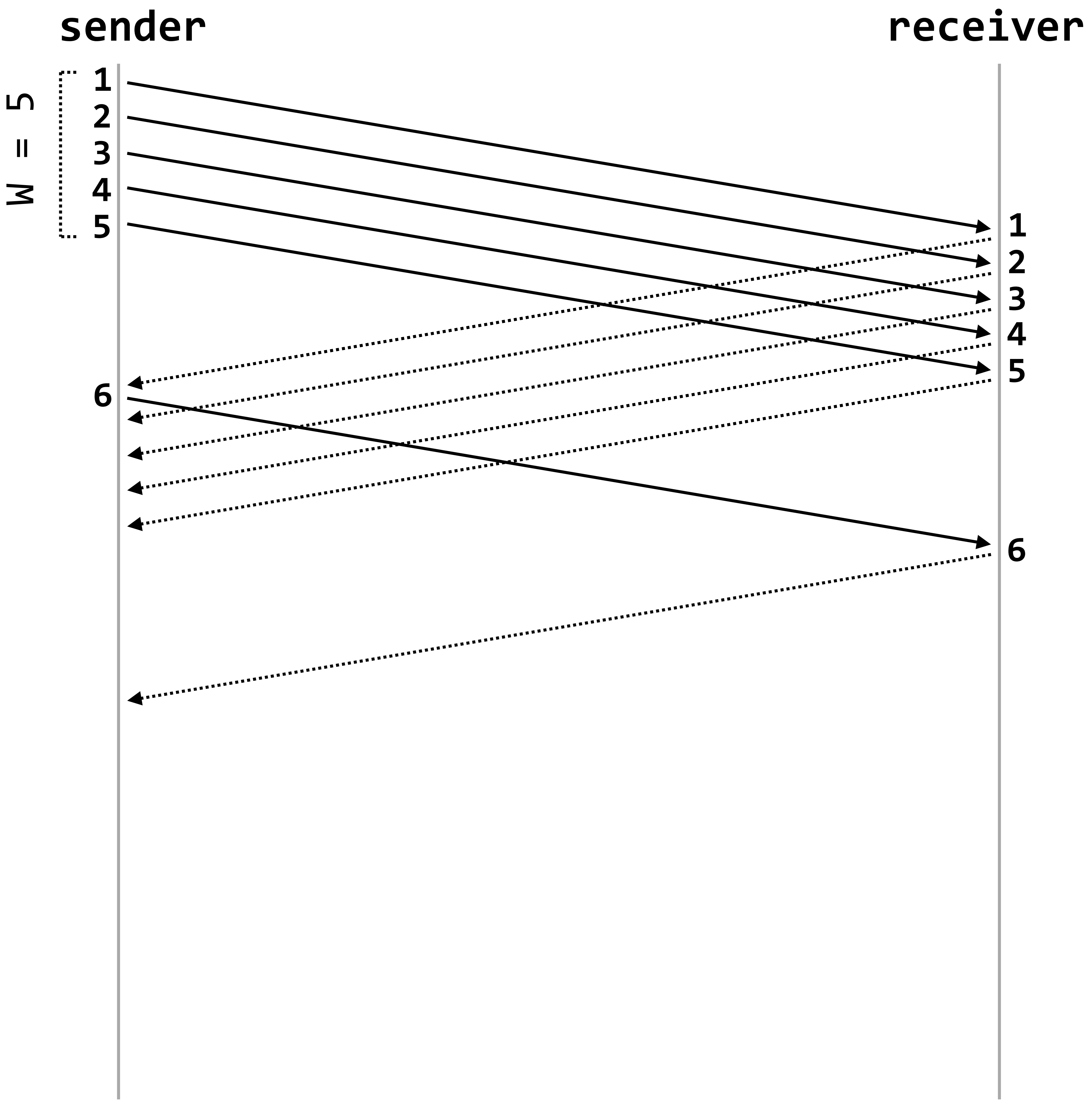
sequence numbers: used to order the packets

acknowledgments (“ACKs”): used to confirm that a packet has been received

an ACK with sequence number k indicates that the receiver has received **all packets up to and including k**

reliable transport protocols deliver each byte of data **exactly once, in-order**, to the receiving application

the sender is allowed to have W outstanding packets at once, but no more



sequence numbers: used to order the packets

acknowledgments ("ACKs"): used to confirm that a packet has been received

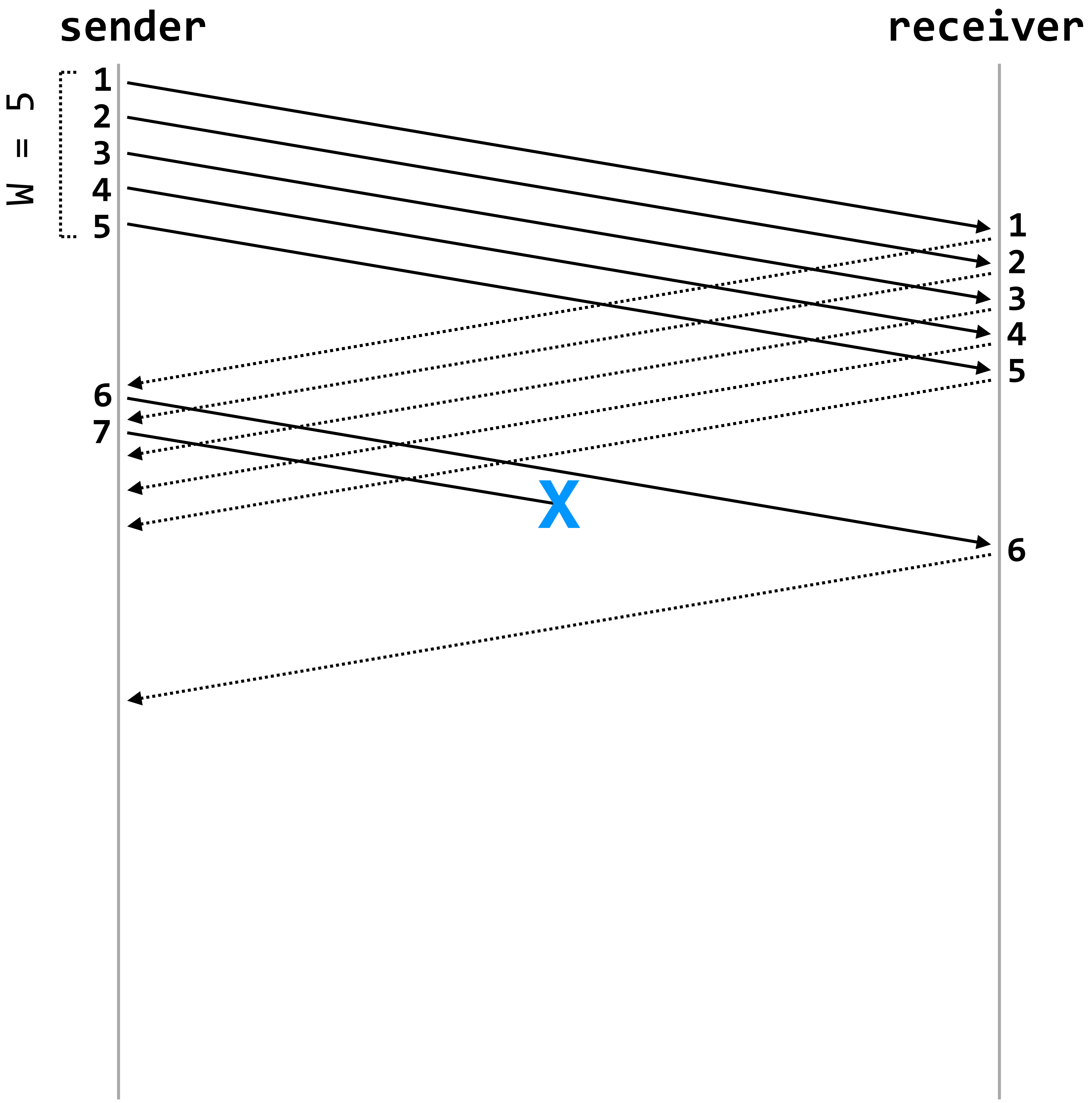
an ACK with sequence number k indicates that the receiver has received **all packets up to and including k**

this is known as a **sliding-window protocol**

the **window** of outstanding (un-ACKed) packets **slides** along the sequence number space

reliable transport protocols deliver each byte of data **exactly once, in-order**, to the receiving application

the sender is allowed to have W outstanding packets at once, but no more



sequence numbers: used to order the packets

acknowledgments ("ACKs"): used to confirm that a packet has been received

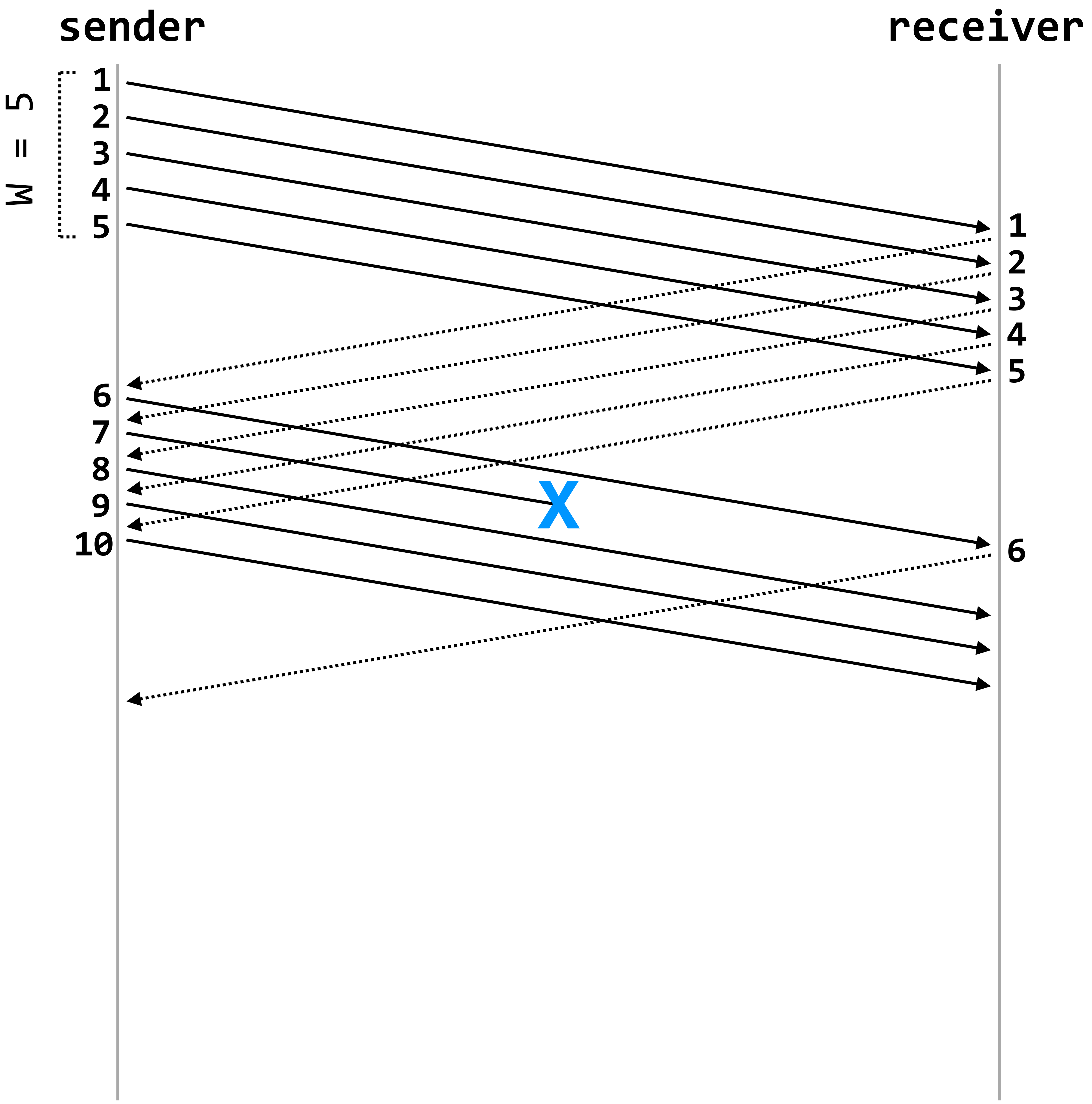
an ACK with sequence number k indicates that the receiver has received **all packets up to and including k**

this is known as a **sliding-window protocol**

the **window** of outstanding (un-ACKed) packets **slides** along the sequence number space

reliable transport protocols deliver each byte of data **exactly once, in-order**, to the receiving application

the sender is allowed to have W outstanding packets at once, but no more



this is known as a **sliding-window protocol**

the **window** of outstanding (un-ACKed) packets **slides** along the sequence number space

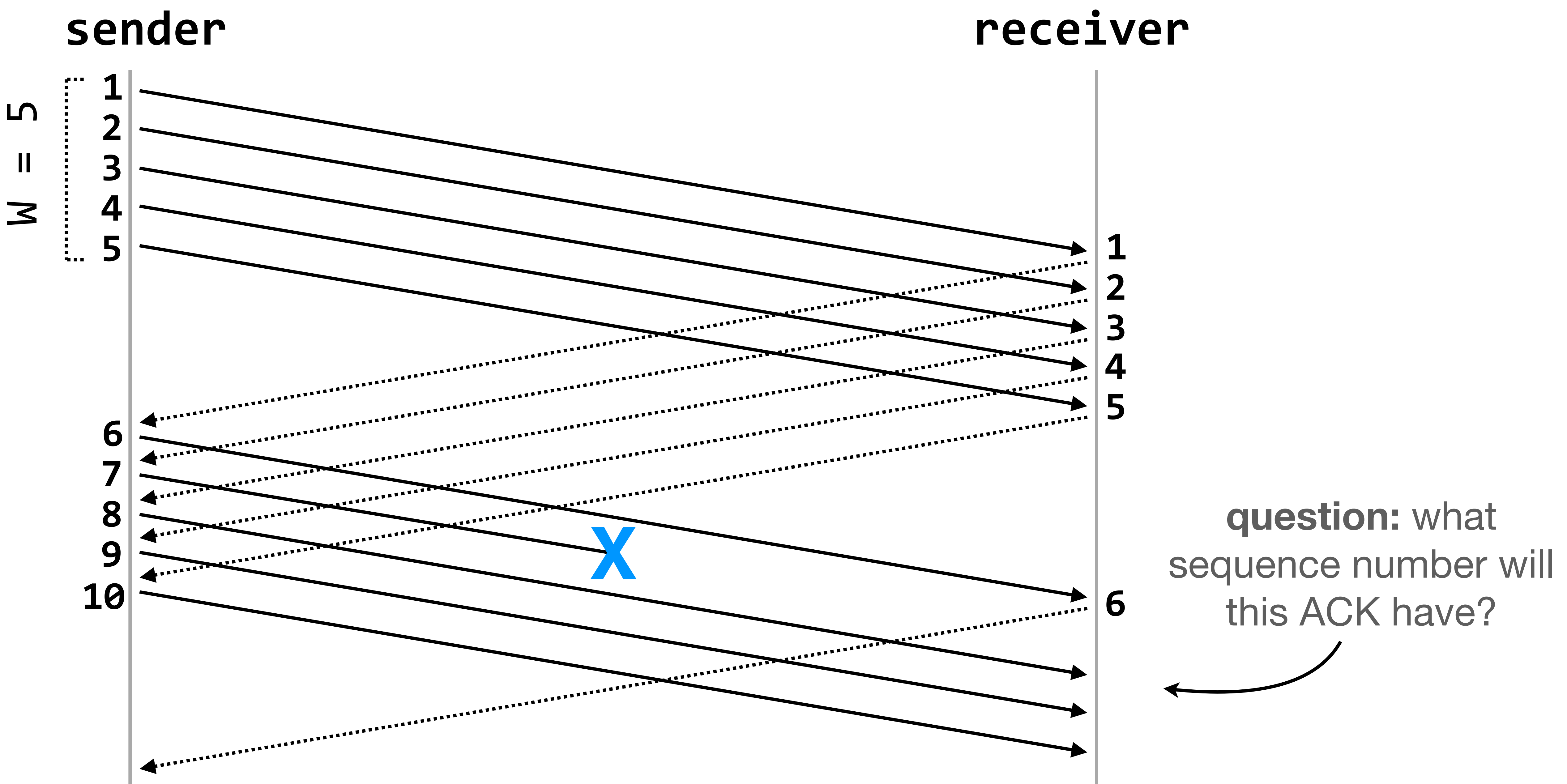
sequence numbers: used to order the packets

acknowledgments ("ACKs"): used to confirm that a packet has been received

an ACK with sequence number k indicates that the receiver has received **all packets up to and including k**

reliable transport protocols deliver each byte of data **exactly once, in-order**, to the receiving application

the sender is allowed to have W outstanding packets at once, but no more



question: what sequence number will this ACK have?

sequence numbers: used to order the packets

acknowledgments ("ACKs"): used to confirm that a packet has been received

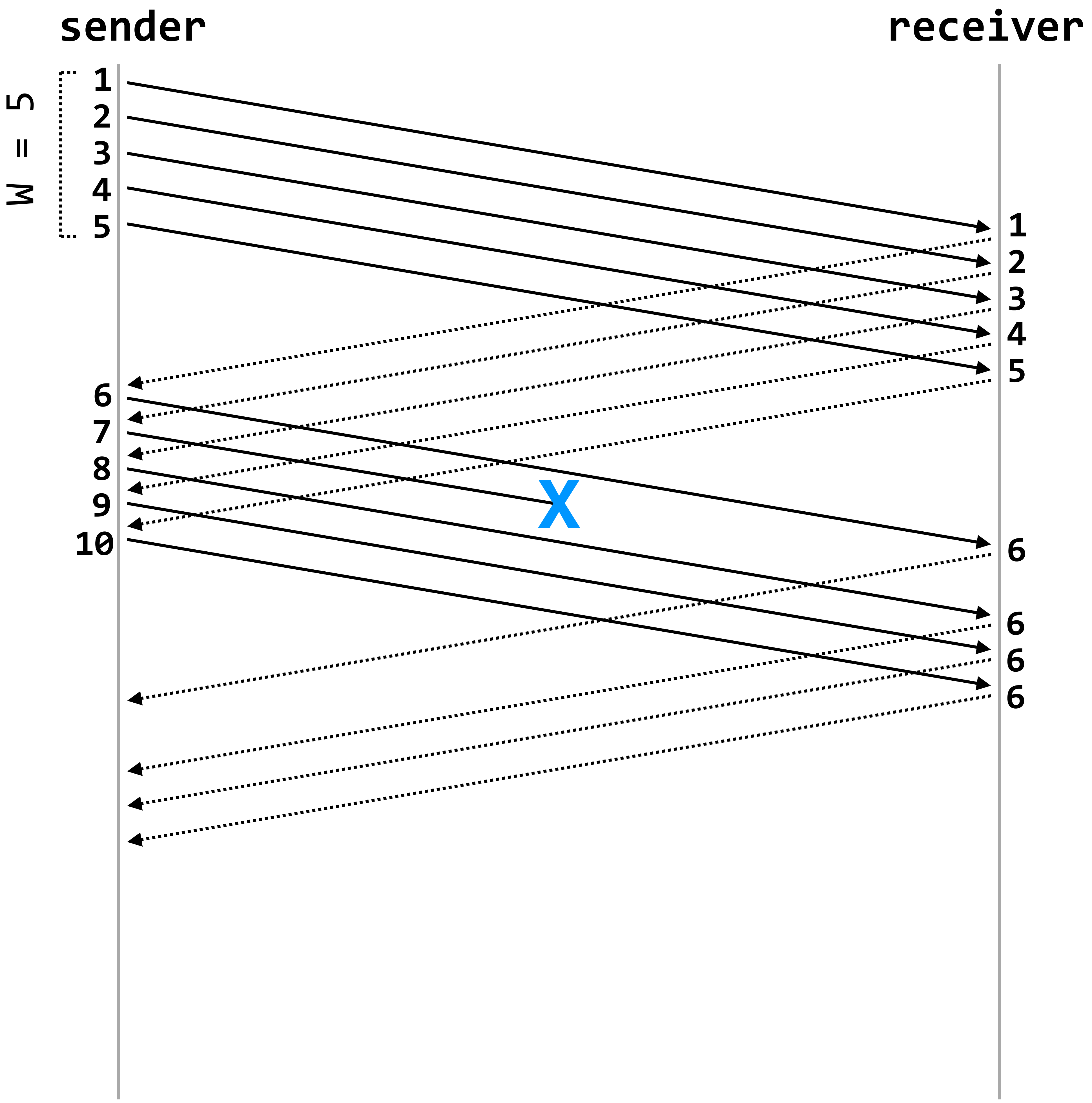
an ACK with sequence number k indicates that the receiver has received **all packets up to and including k**

this is known as a **sliding-window protocol**

the **window** of outstanding (un-ACKed) packets **slides** along the sequence number space

reliable transport protocols deliver each byte of data **exactly once, in-order**, to the receiving application

the sender is allowed to have W outstanding packets at once, but no more



sequence numbers: used to order the packets

acknowledgments ("ACKs"): used to confirm that a packet has been received

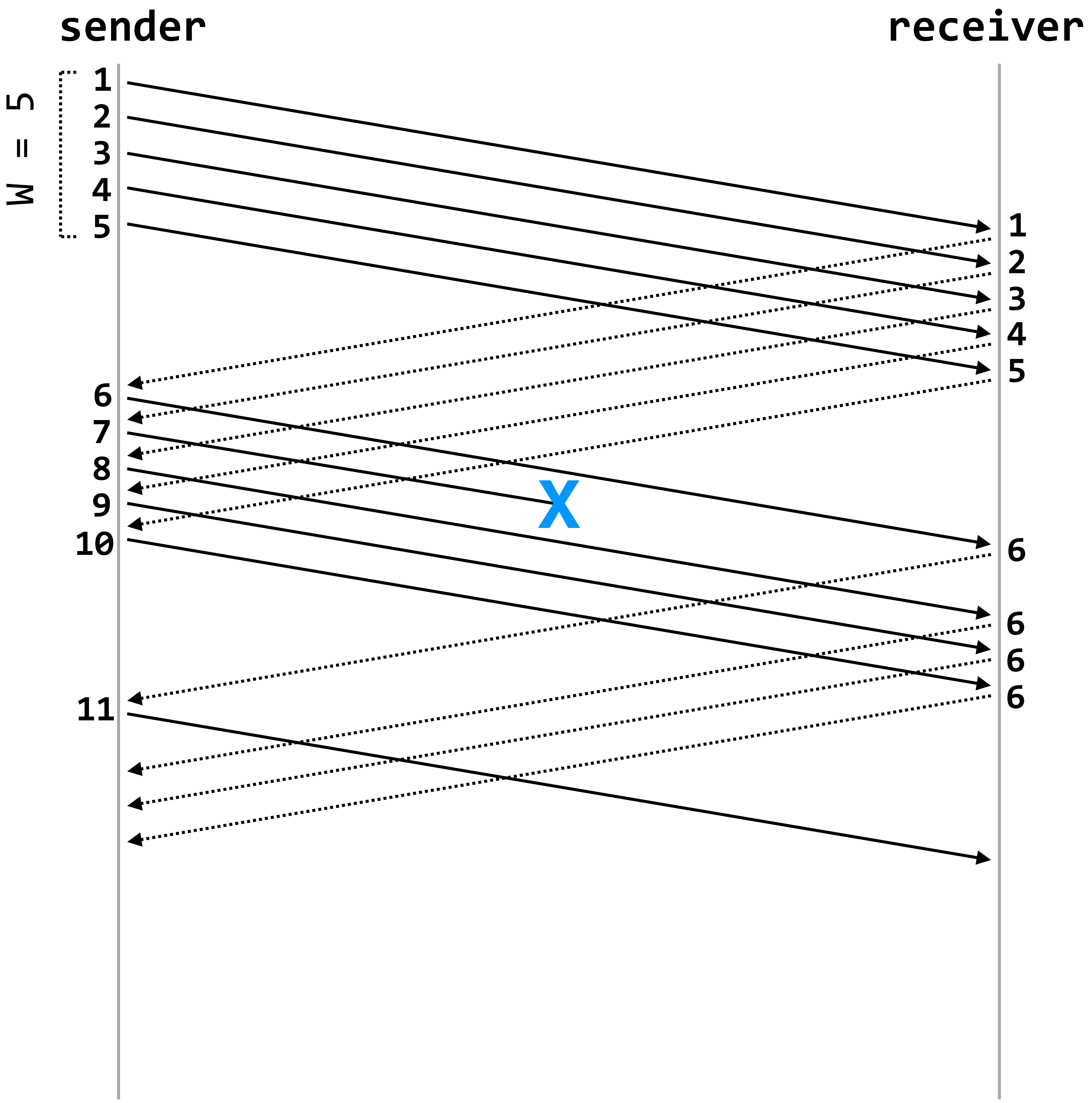
an ACK with sequence number k indicates that the receiver has received **all packets up to and including k**

this is known as a **sliding-window protocol**

the **window** of outstanding (un-ACKed) packets **slides** along the sequence number space

reliable transport protocols deliver each byte of data **exactly once, in-order**, to the receiving application

the sender is allowed to have W outstanding packets at once, but no more



sequence numbers: used to order the packets

acknowledgments ("ACKs"): used to confirm that a packet has been received

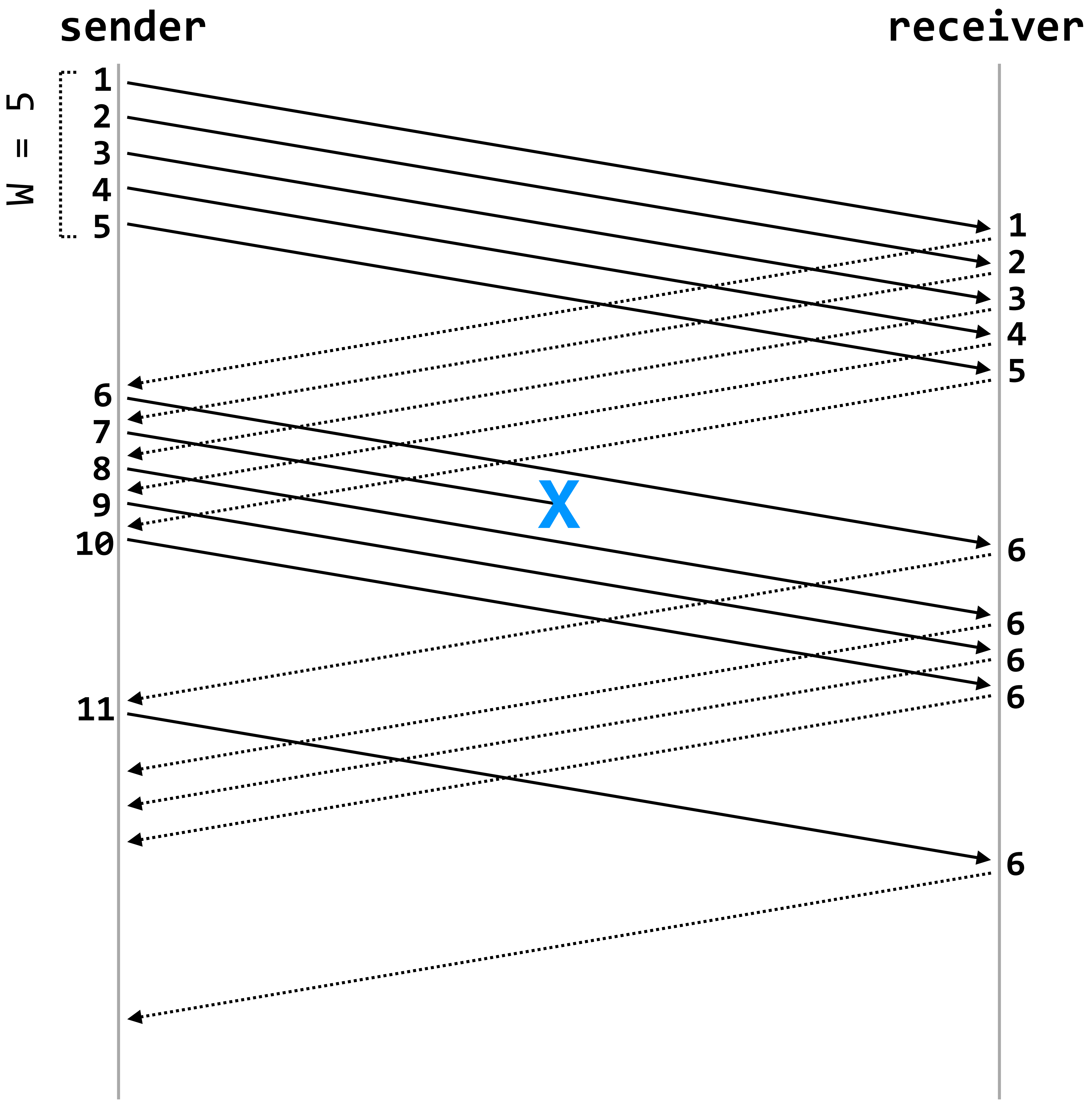
an ACK with sequence number k indicates that the receiver has received **all packets up to and including k**

this is known as a **sliding-window protocol**

the **window** of outstanding (un-ACKed) packets **slides** along the sequence number space

reliable transport protocols deliver each byte of data **exactly once, in-order**, to the receiving application

the sender is allowed to have W outstanding packets at once, but no more



sequence numbers: used to order the packets

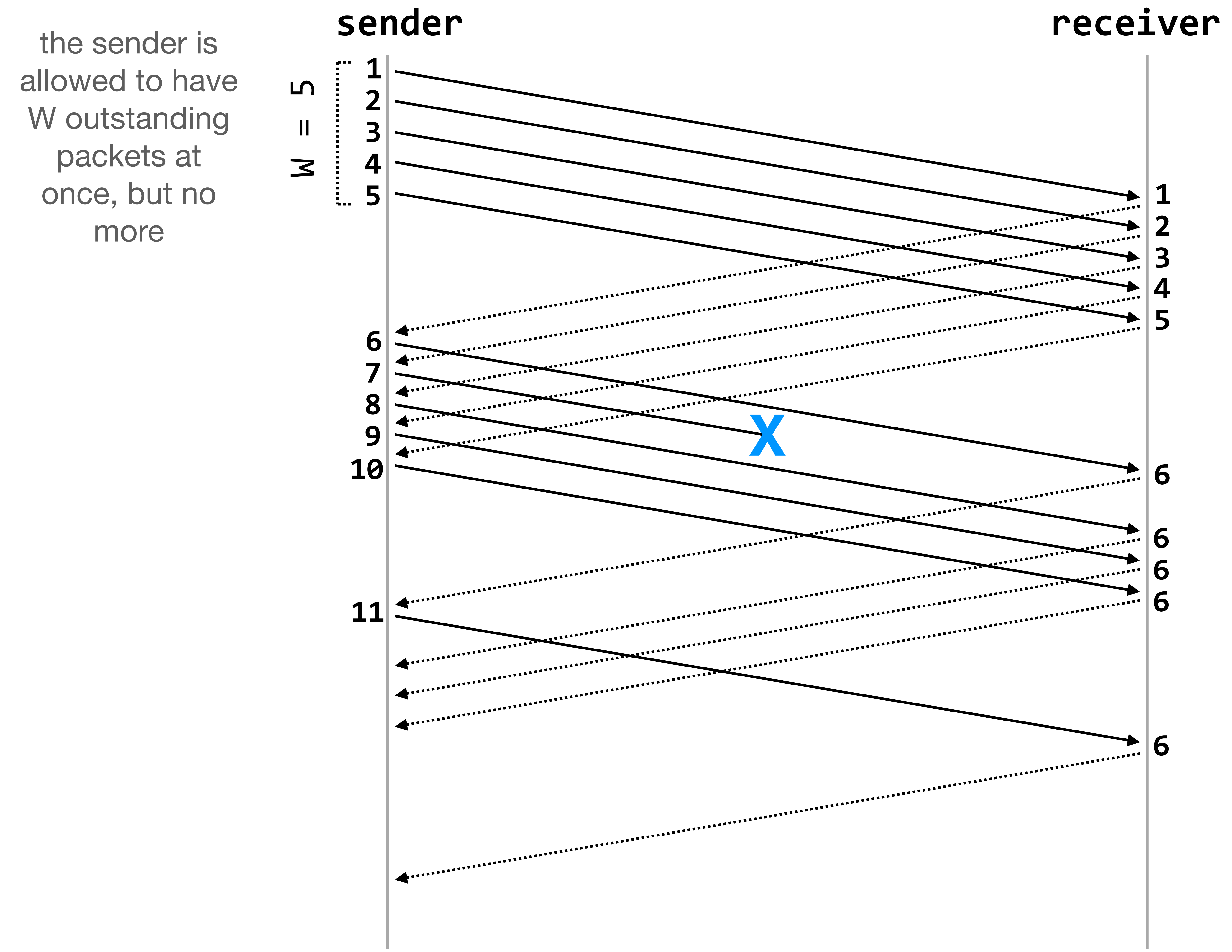
acknowledgments ("ACKs"): used to confirm that a packet has been received

an ACK with sequence number k indicates that the receiver has received **all packets up to and including k**

this is known as a **sliding-window protocol**

the **window** of outstanding (un-ACKed) packets **slides** along the sequence number space

reliable transport protocols deliver each byte of data **exactly once, in-order**, to the receiving application



the sender is allowed to have W outstanding packets at once, but no more
W = 5
sender
receiver
1
2
3
4
5
1
2
3
4
5
6
7
8
9
10
X
6
6
6
6
11
6

this is known as a **sliding-window protocol**

the **window** of outstanding (un-ACKed) packets **slides** along the sequence number space

sequence numbers: used to order the packets

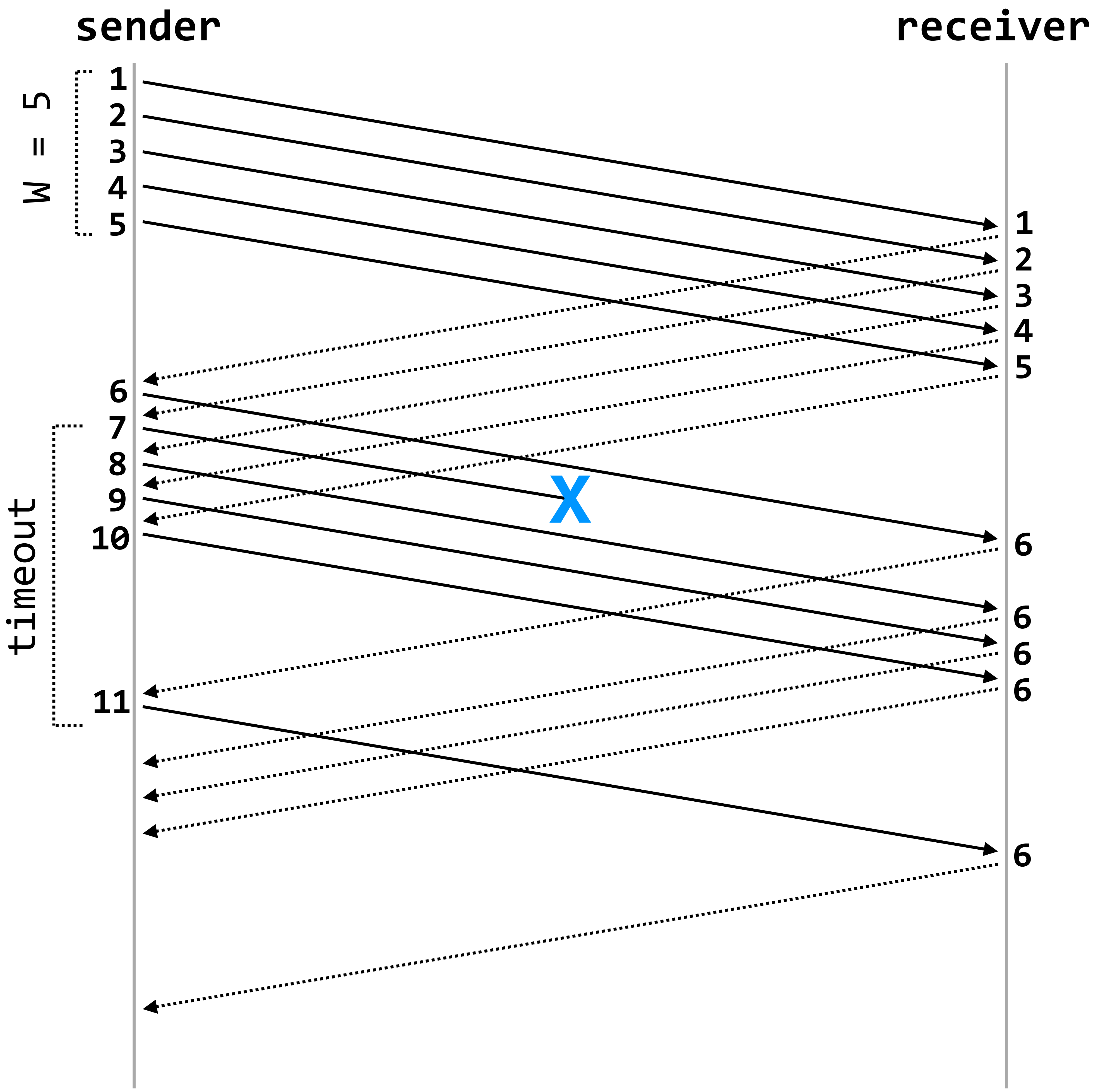
acknowledgments ("ACKs"): used to confirm that a packet has been received

an ACK with sequence number k indicates that the receiver has received **all packets up to and including k**

question: can the sender infer that packet 7 has been lost?

reliable transport protocols deliver each byte of data **exactly once, in-order**, to the receiving application

the sender is allowed to have W outstanding packets at once, but no more



this is known as a **sliding-window protocol**

the **window** of outstanding (un-ACKed) packets **slides** along the sequence number space

sequence numbers: used to order the packets

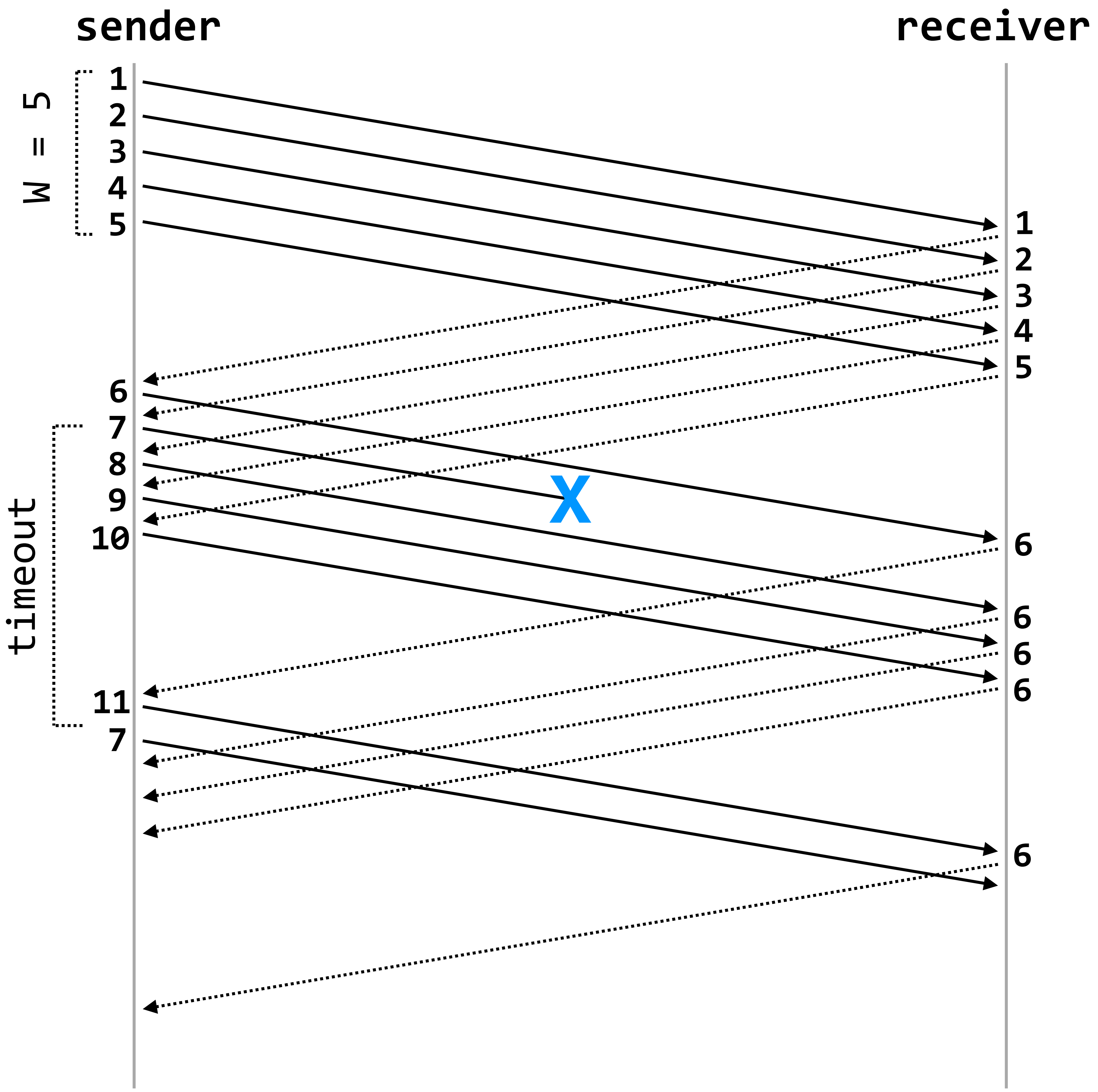
acknowledgments ("ACKs"): used to confirm that a packet has been received

an ACK with sequence number k indicates that the receiver has received **all packets up to and including k**

timeouts: used to retransmit packets

reliable transport protocols deliver each byte of data **exactly once, in-order**, to the receiving application

the sender is allowed to have W outstanding packets at once, but no more



this is known as a **sliding-window protocol**

the **window** of outstanding (un-ACKed) packets **slides** along the sequence number space

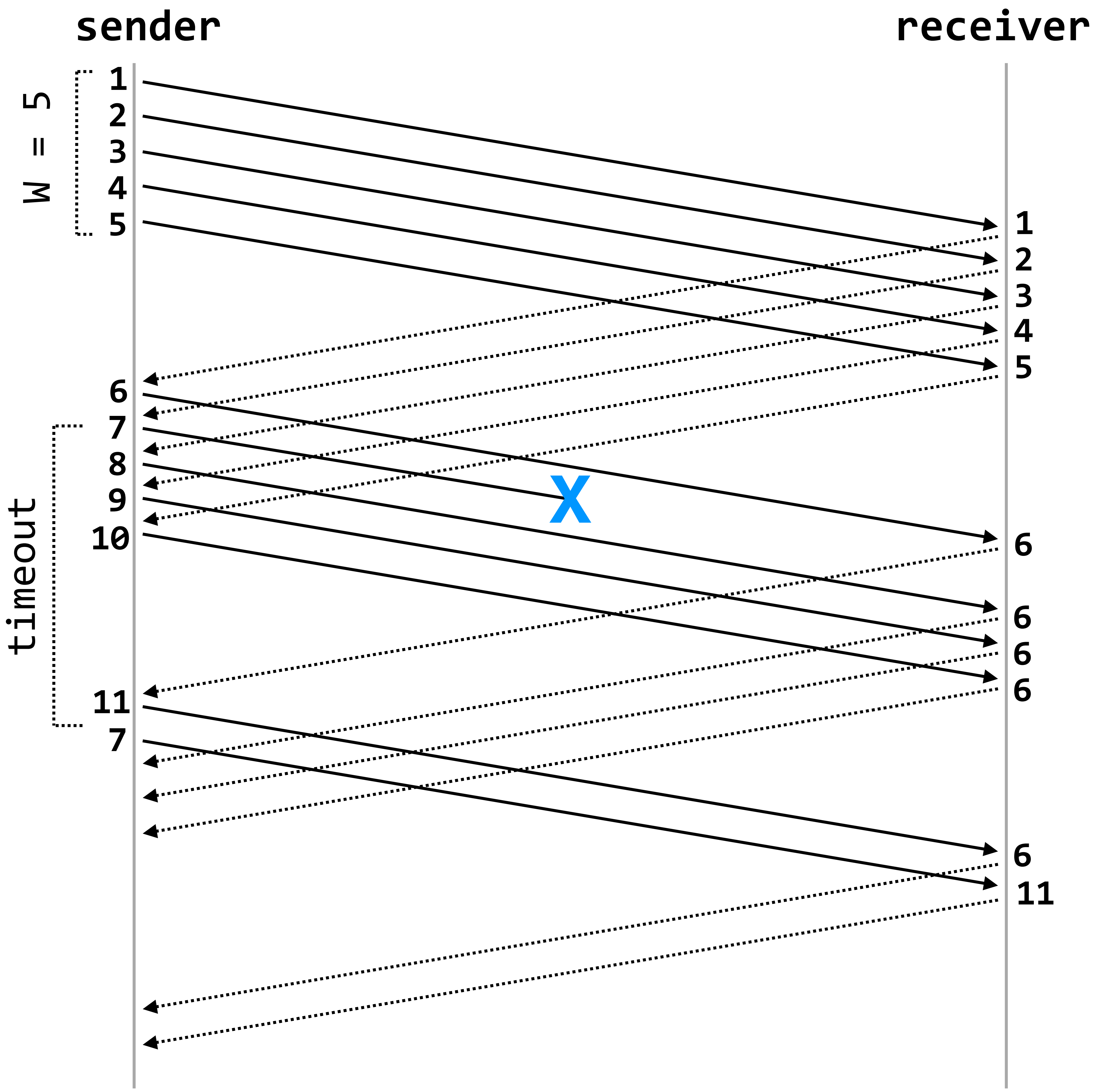
sequence numbers: used to order the packets

acknowledgments ("ACKs"): used to confirm that a packet has been received

an ACK with sequence number k indicates that the receiver has received **all packets up to and including k**

timeouts: used to retransmit packets

reliable transport protocols deliver each byte of data **exactly once, in-order**, to the receiving application



the sender is allowed to have W outstanding packets at once, but no more

sequence numbers: used to order the packets

acknowledgments ("ACKs"): used to confirm that a packet has been received

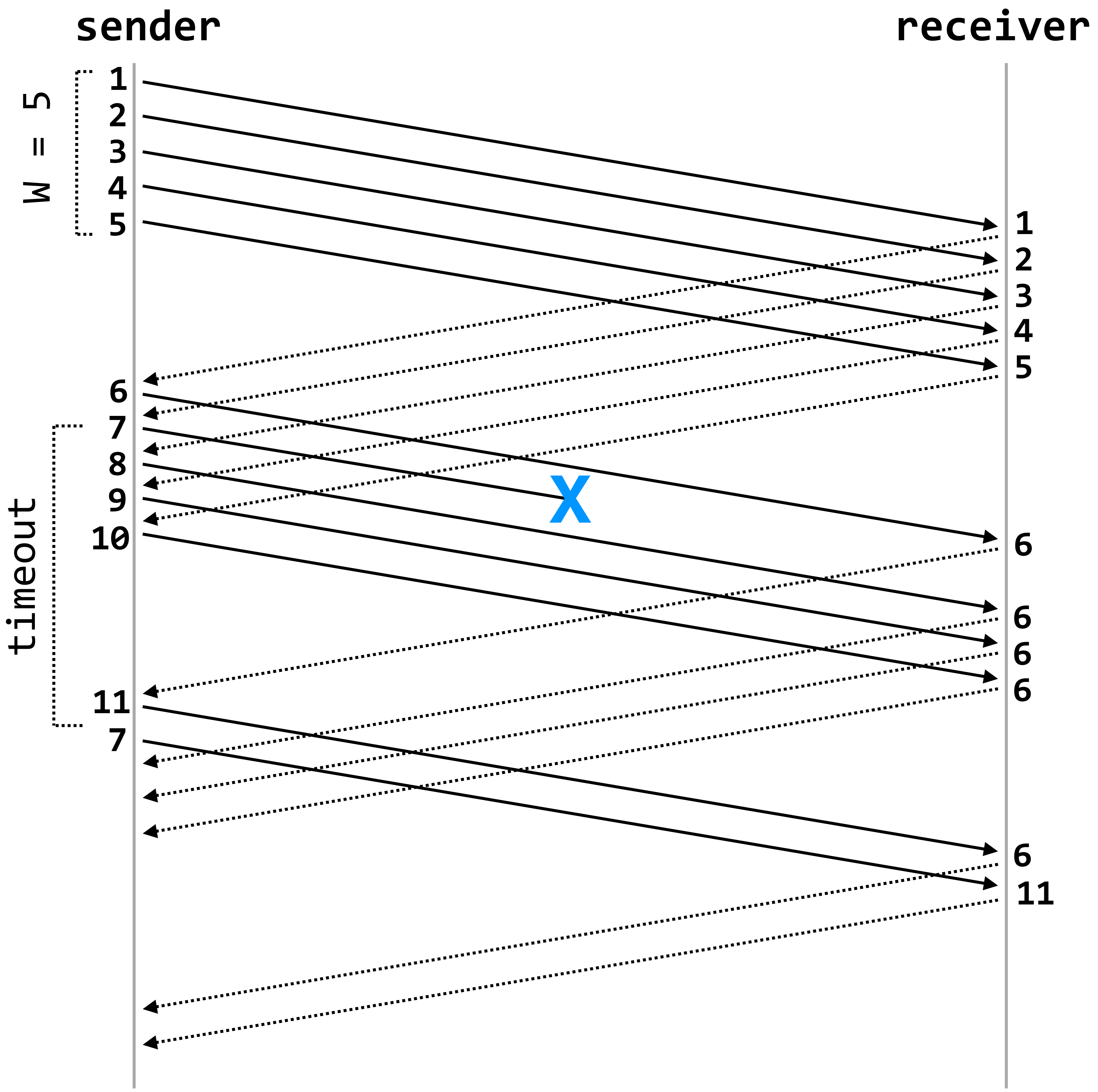
an ACK with sequence number k indicates that the receiver has received **all packets up to and including k**

timeouts: used to retransmit packets

this is known as a **sliding-window protocol**

the **window** of outstanding (un-ACKed) packets **slides** along the sequence number space

reliable transport protocols deliver each byte of data **exactly once, in-order**, to the receiving application



the sender is allowed to have W outstanding packets at once, but no more

sequence numbers: used to order the packets

acknowledgments ("ACKs"): used to confirm that a packet has been received

an ACK with sequence number k indicates that the receiver has received **all packets up to and including k**

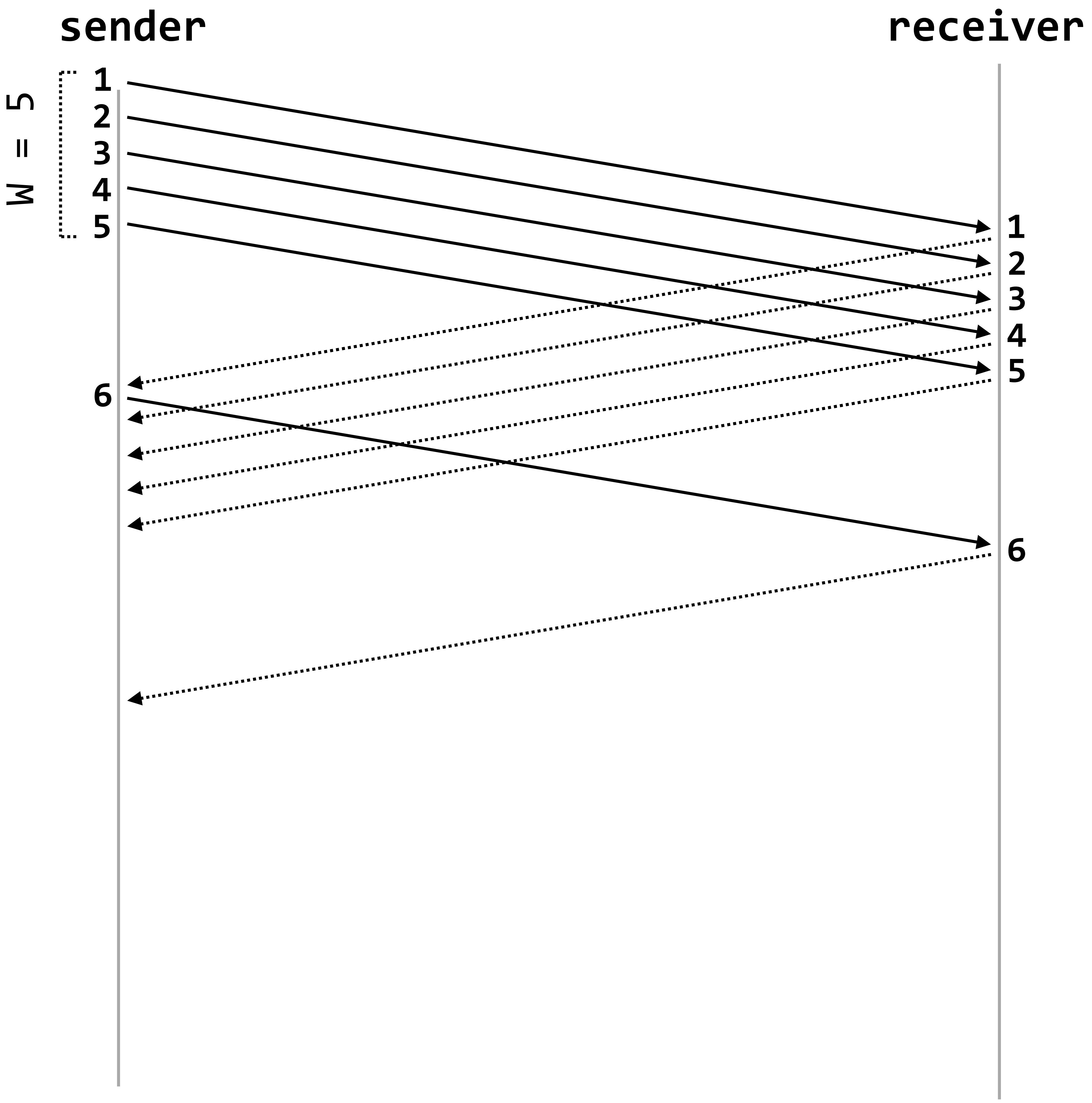
timeouts: used to retransmit packets

note that the sender could also infer loss because it has received multiple ACKs with sequence number 6, but none with sequence number > 7; we'll come back to that

this is known as a **sliding-window protocol**

the **window** of outstanding (un-ACKed) packets **slides** along the sequence number space

reliable transport protocols deliver each byte of data **exactly once, in-order**, to the receiving application



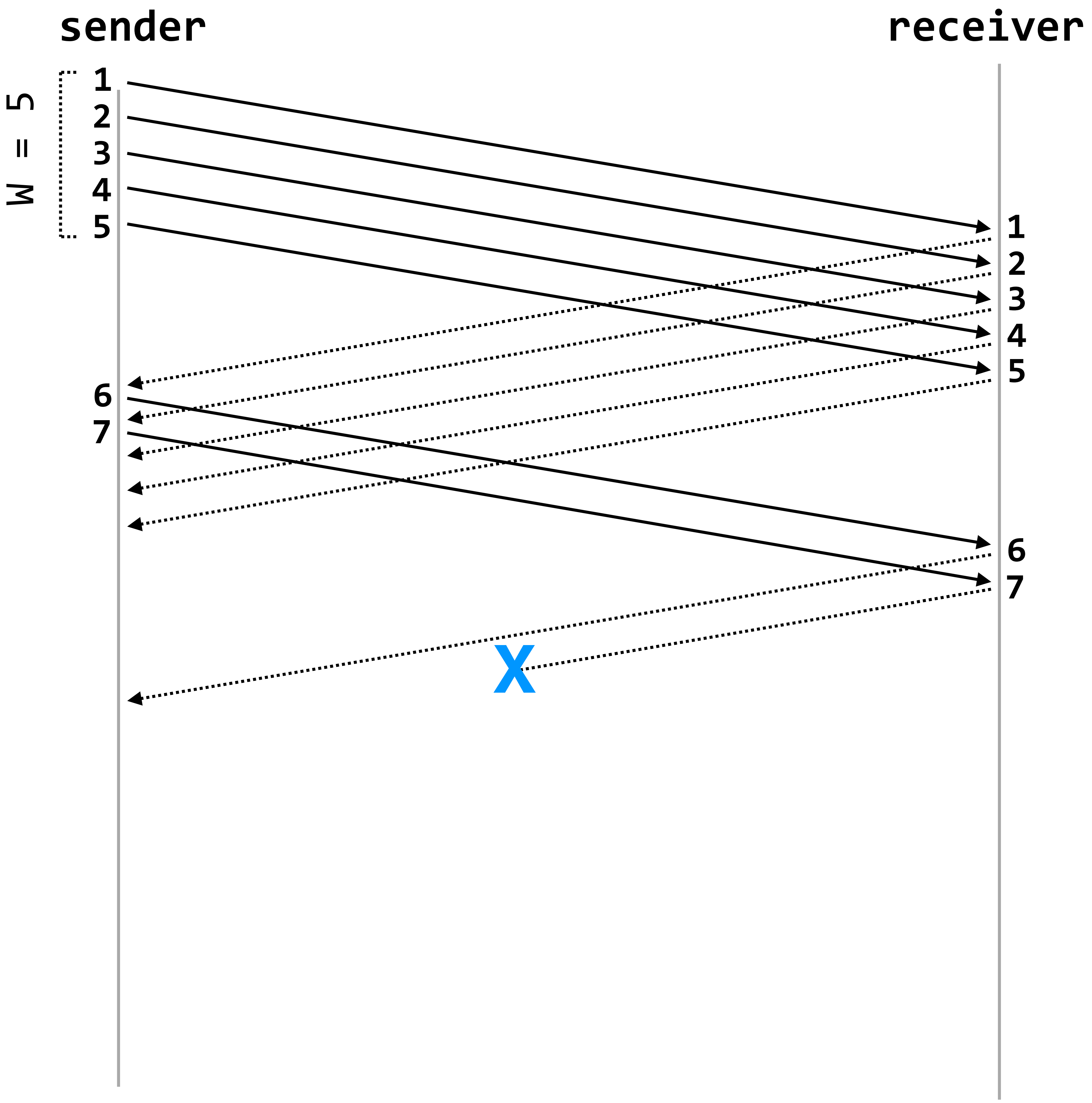
sequence numbers: used to order the packets

acknowledgments ("ACKs"): used to confirm that a packet has been received

an ACK with sequence number k indicates that the receiver has received **all packets up to and including k**

timeouts: used to retransmit packets

reliable transport protocols deliver each byte of data **exactly once, in-order**, to the receiving application



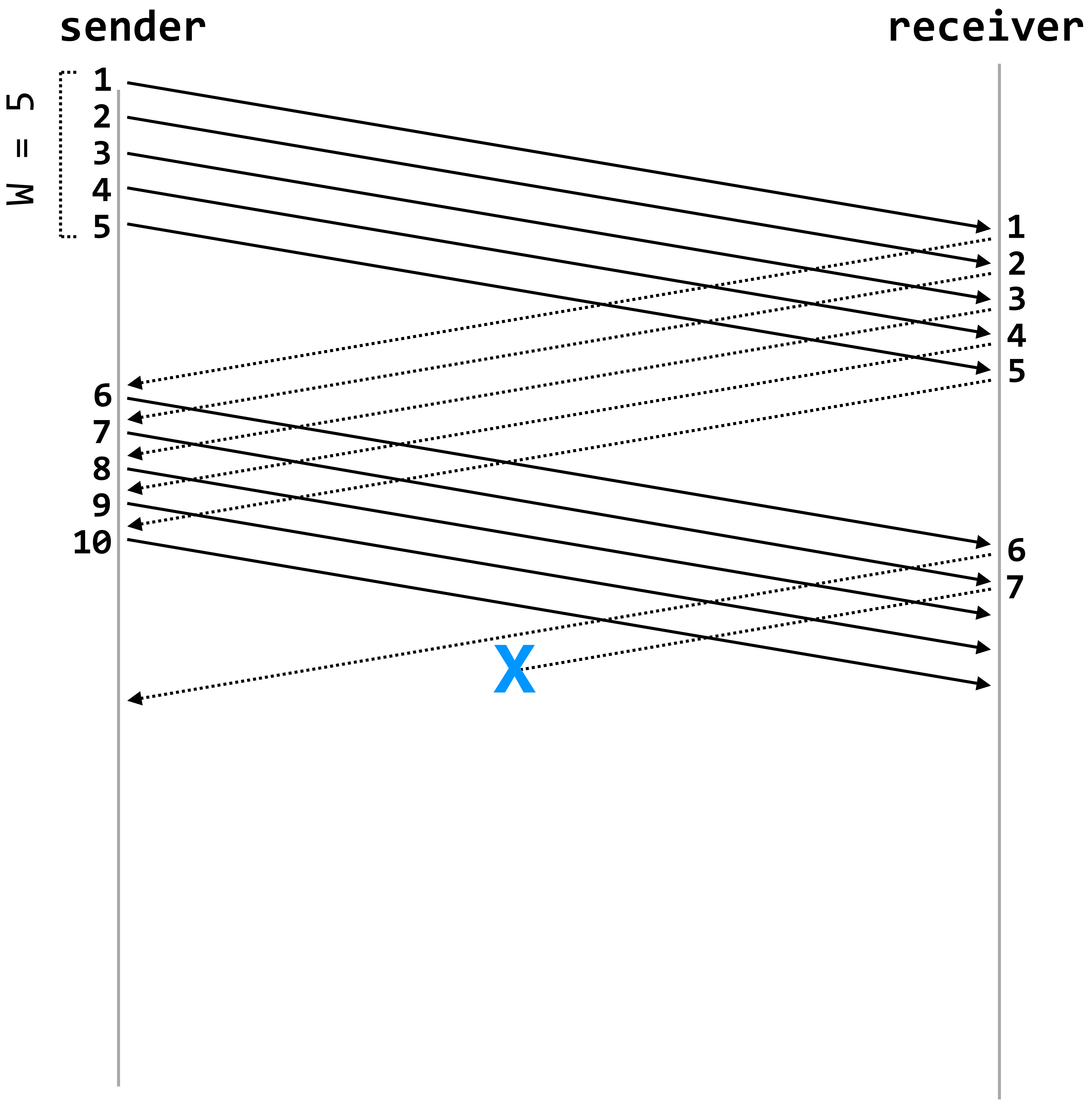
sequence numbers: used to order the packets

acknowledgments ("ACKs"): used to confirm that a packet has been received

an ACK with sequence number k indicates that the receiver has received **all packets up to and including k**

timeouts: used to retransmit packets

reliable transport protocols deliver each byte of data **exactly once, in-order**, to the receiving application



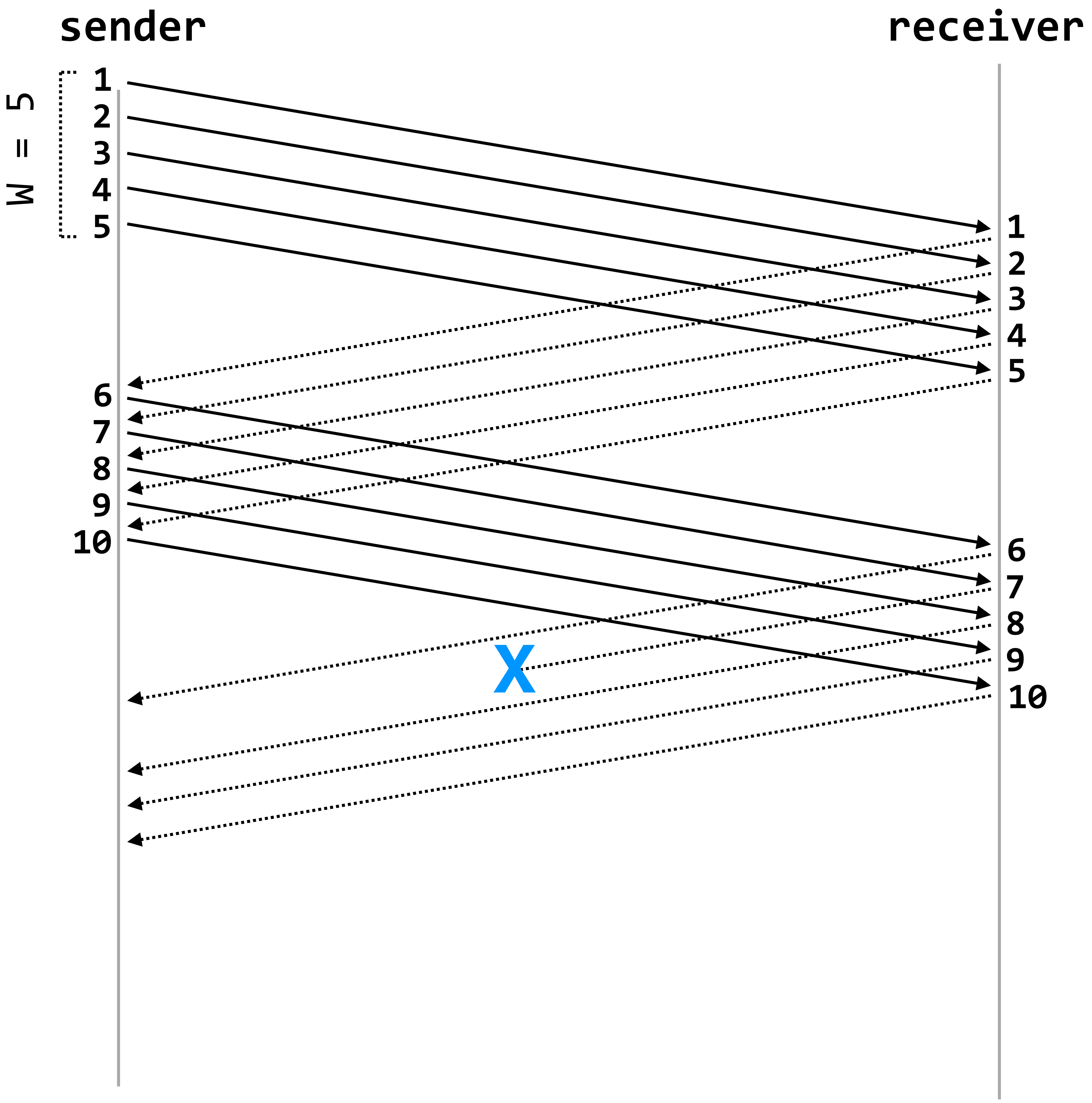
sequence numbers: used to order the packets

acknowledgments ("ACKs"): used to confirm that a packet has been received

an ACK with sequence number k indicates that the receiver has received **all packets up to and including k**

timeouts: used to retransmit packets

reliable transport protocols deliver each byte of data **exactly once, in-order**, to the receiving application



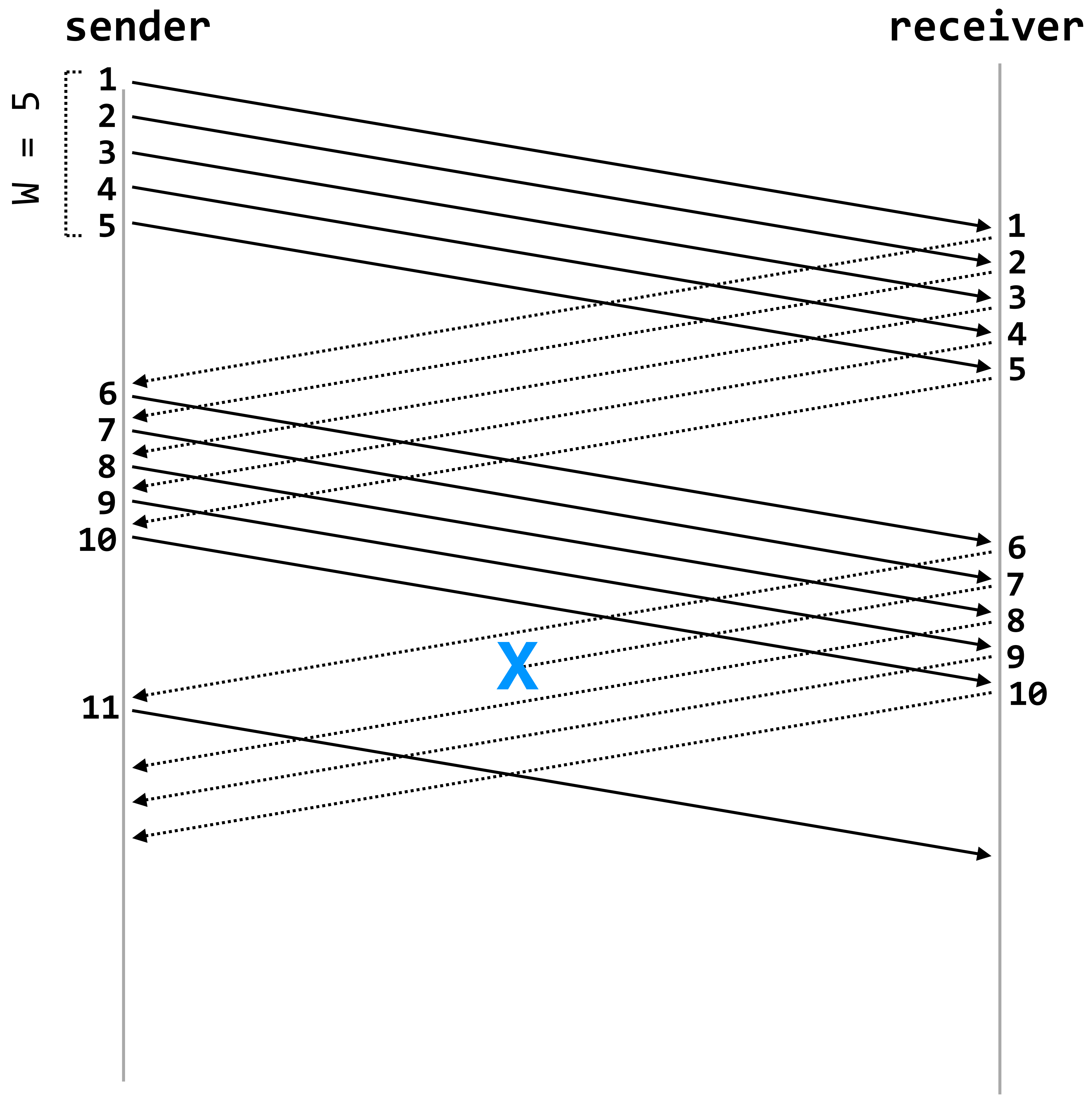
sequence numbers: used to order the packets

acknowledgments ("ACKs"): used to confirm that a packet has been received

an ACK with sequence number k indicates that the receiver has received **all packets up to and including k**

timeouts: used to retransmit packets

reliable transport protocols deliver each byte of data **exactly once, in-order**, to the receiving application



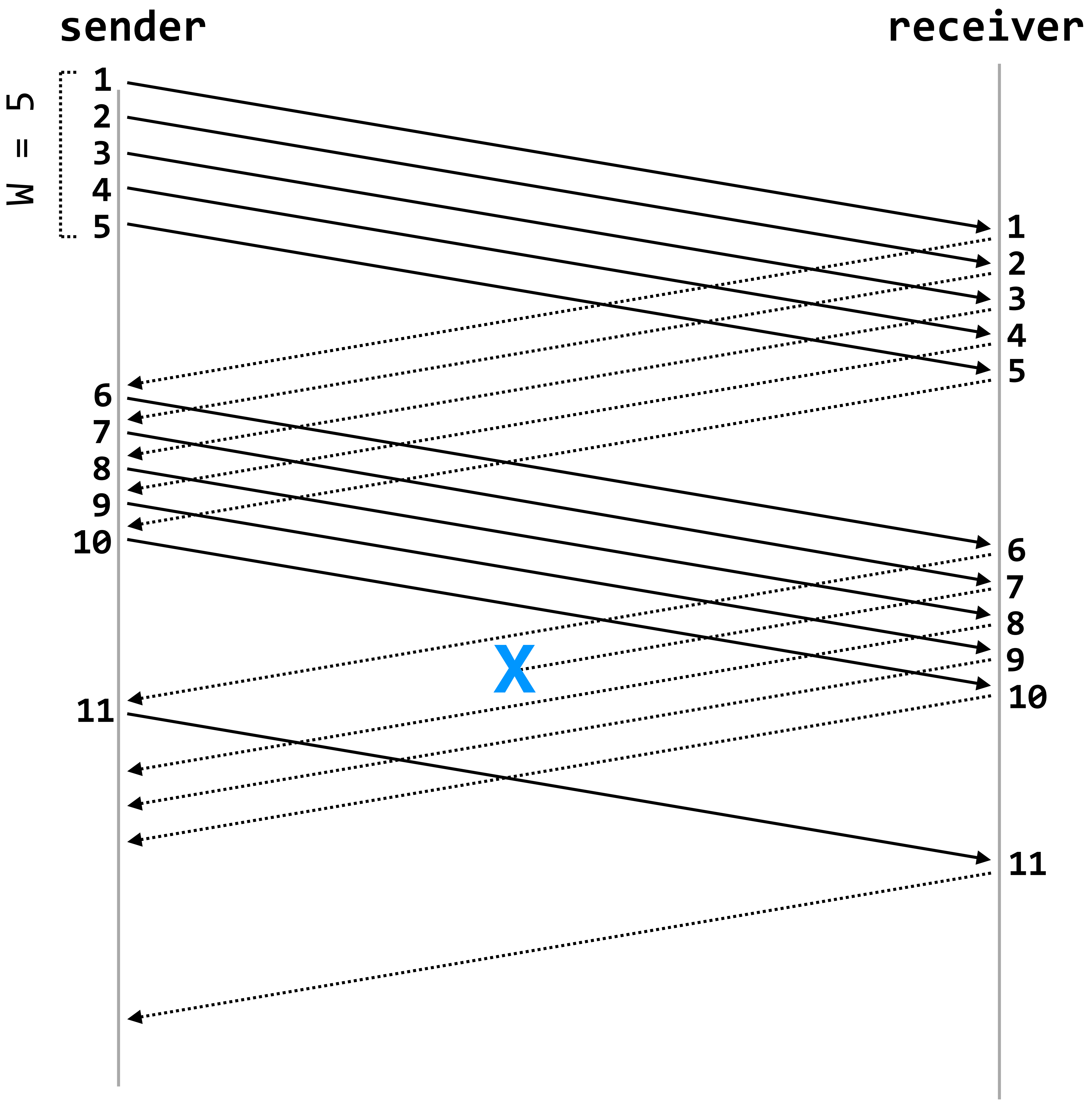
sequence numbers: used to order the packets

acknowledgments ("ACKs"): used to confirm that a packet has been received

an ACK with sequence number k indicates that the receiver has received **all packets up to and including k**

timeouts: used to retransmit packets

reliable transport protocols deliver each byte of data **exactly once, in-order**, to the receiving application



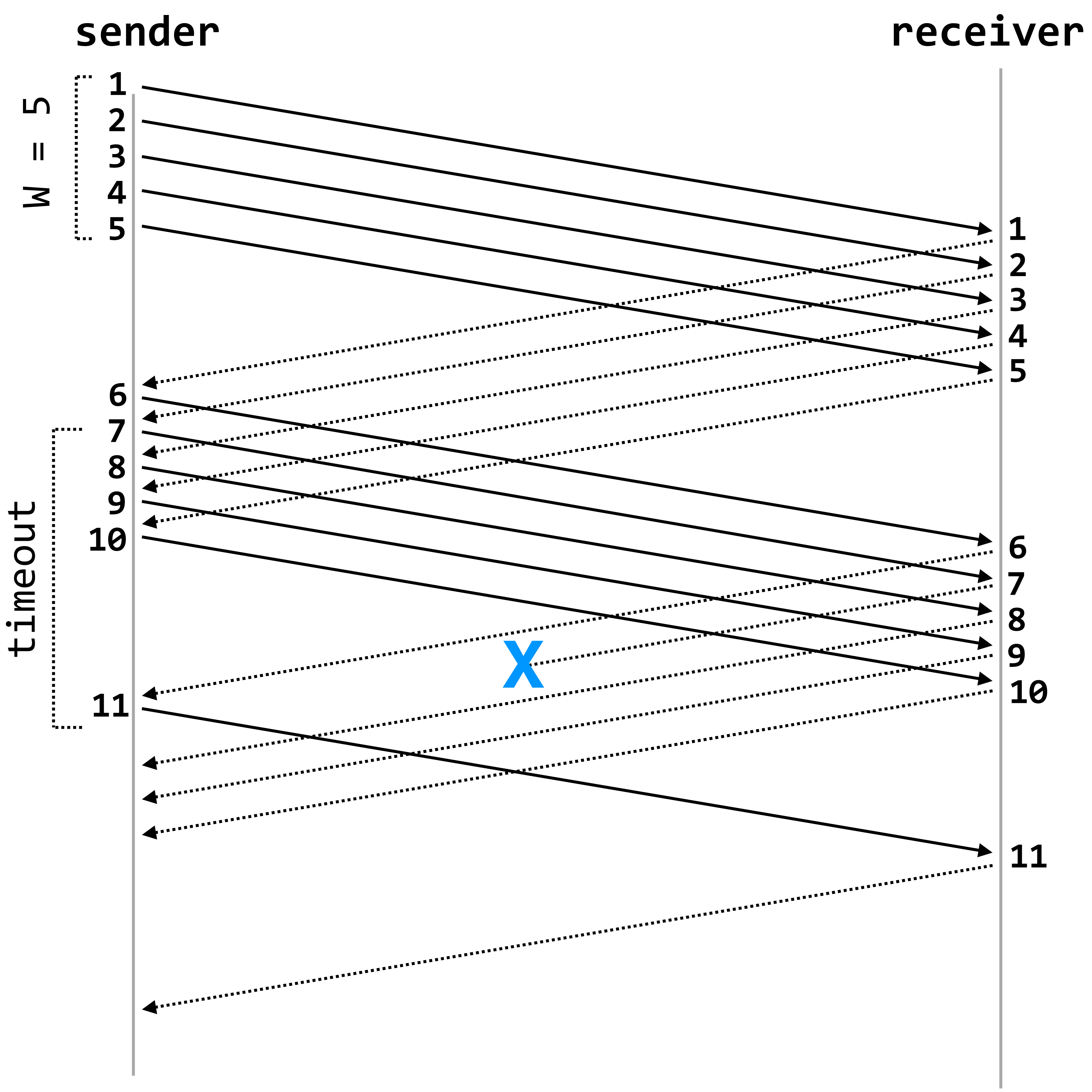
sequence numbers: used to order the packets

acknowledgments ("ACKs"): used to confirm that a packet has been received

an ACK with sequence number k indicates that the receiver has received **all packets up to and including k**

timeouts: used to retransmit packets

reliable transport protocols deliver each byte of data **exactly once, in-order**, to the receiving application



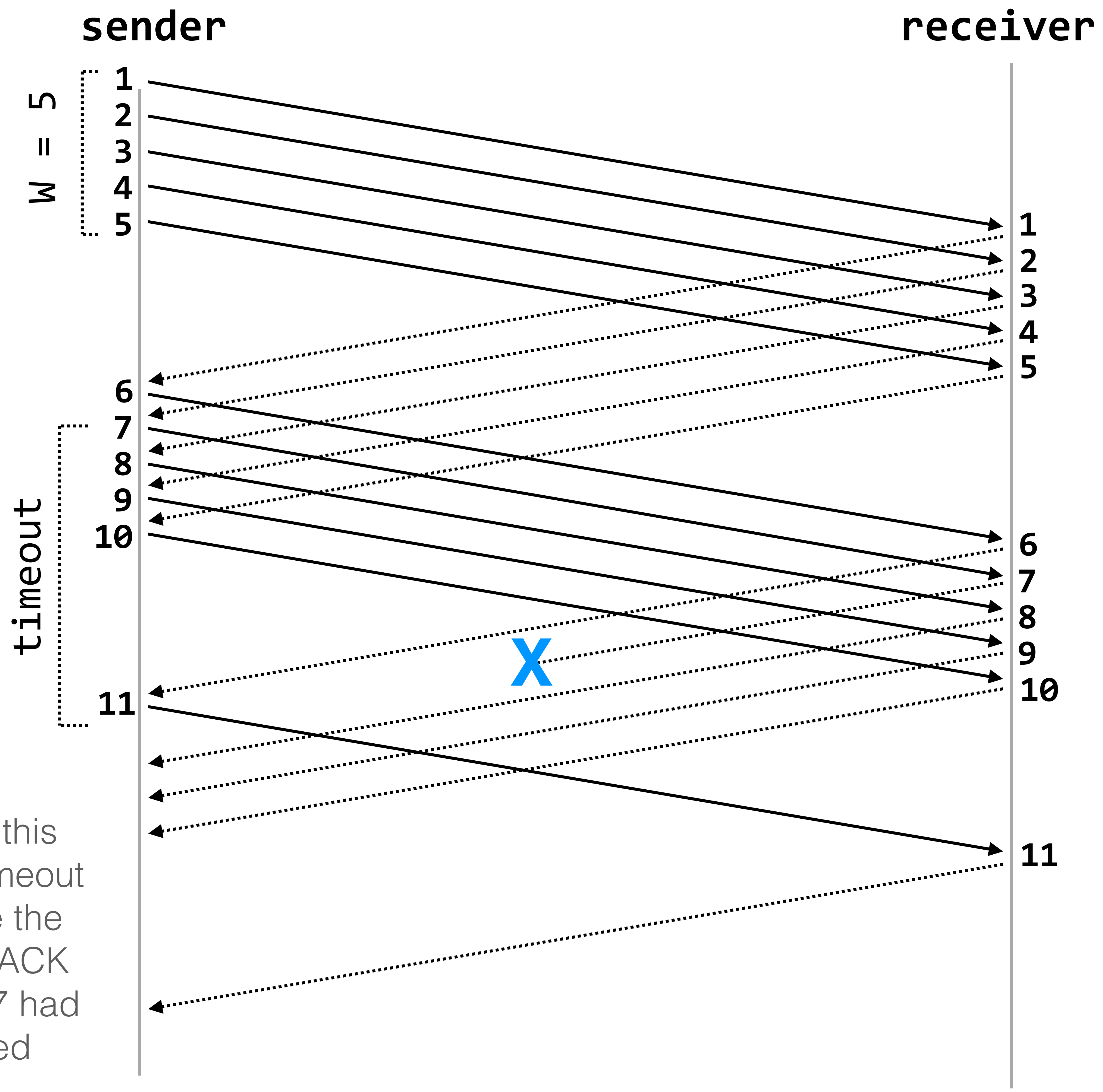
sequence numbers: used to order the packets

acknowledgments ("ACKs"): used to confirm that a packet has been received

an ACK with sequence number k indicates that the receiver has received **all packets up to and including k**

timeouts: used to retransmit packets

reliable transport protocols deliver each byte of data **exactly once, in-order**, to the receiving application



notice that (in this example) the timeout expired before the sender got an ACK indicating that 7 had been received

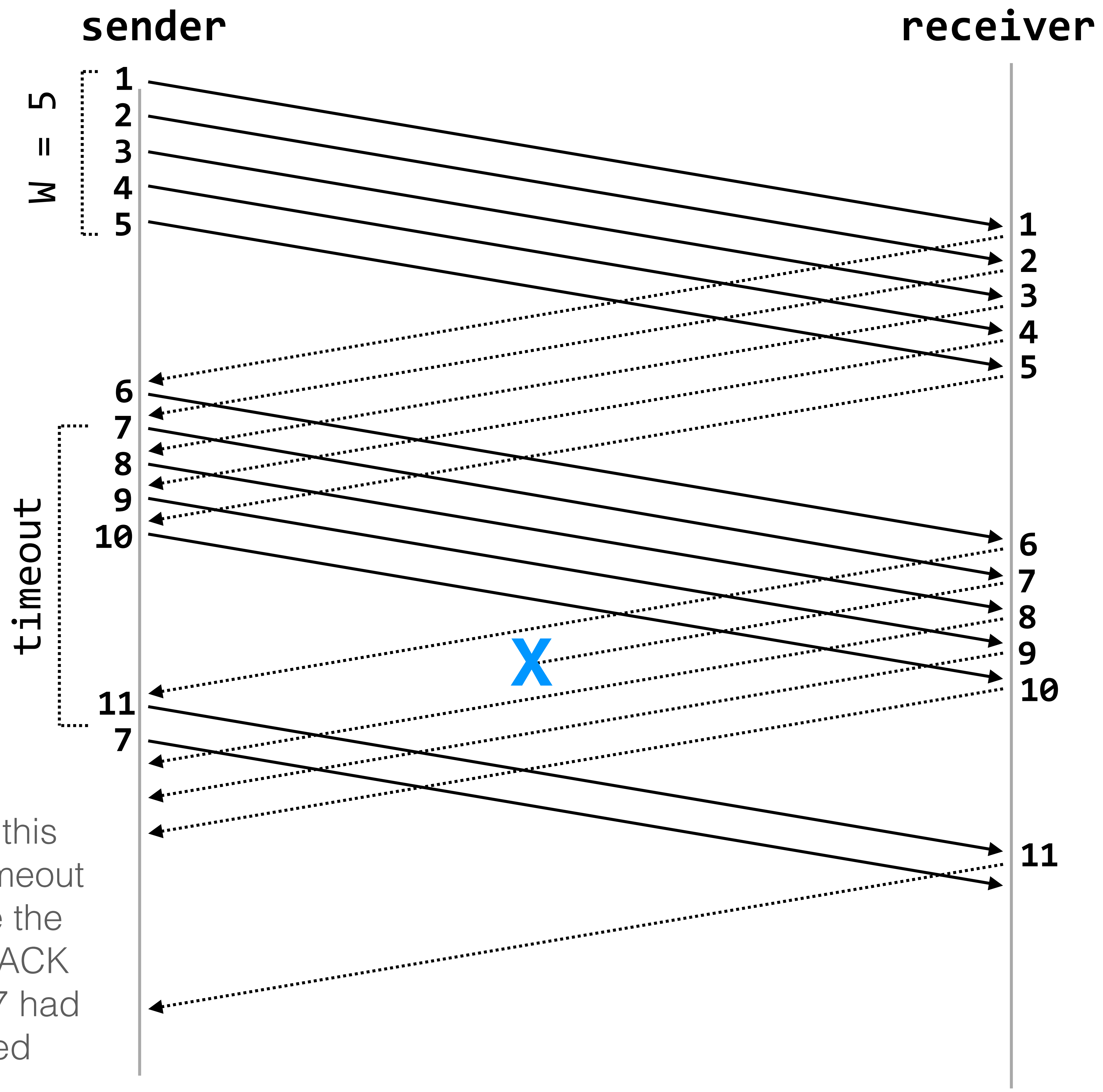
sequence numbers: used to order the packets

acknowledgments ("ACKs"): used to confirm that a packet has been received

an ACK with sequence number k indicates that the receiver has received **all packets up to and including k**

timeouts: used to retransmit packets

reliable transport protocols deliver each byte of data **exactly once, in-order**, to the receiving application



notice that (in this example) the timeout expired before the sender got an ACK indicating that 7 had been received

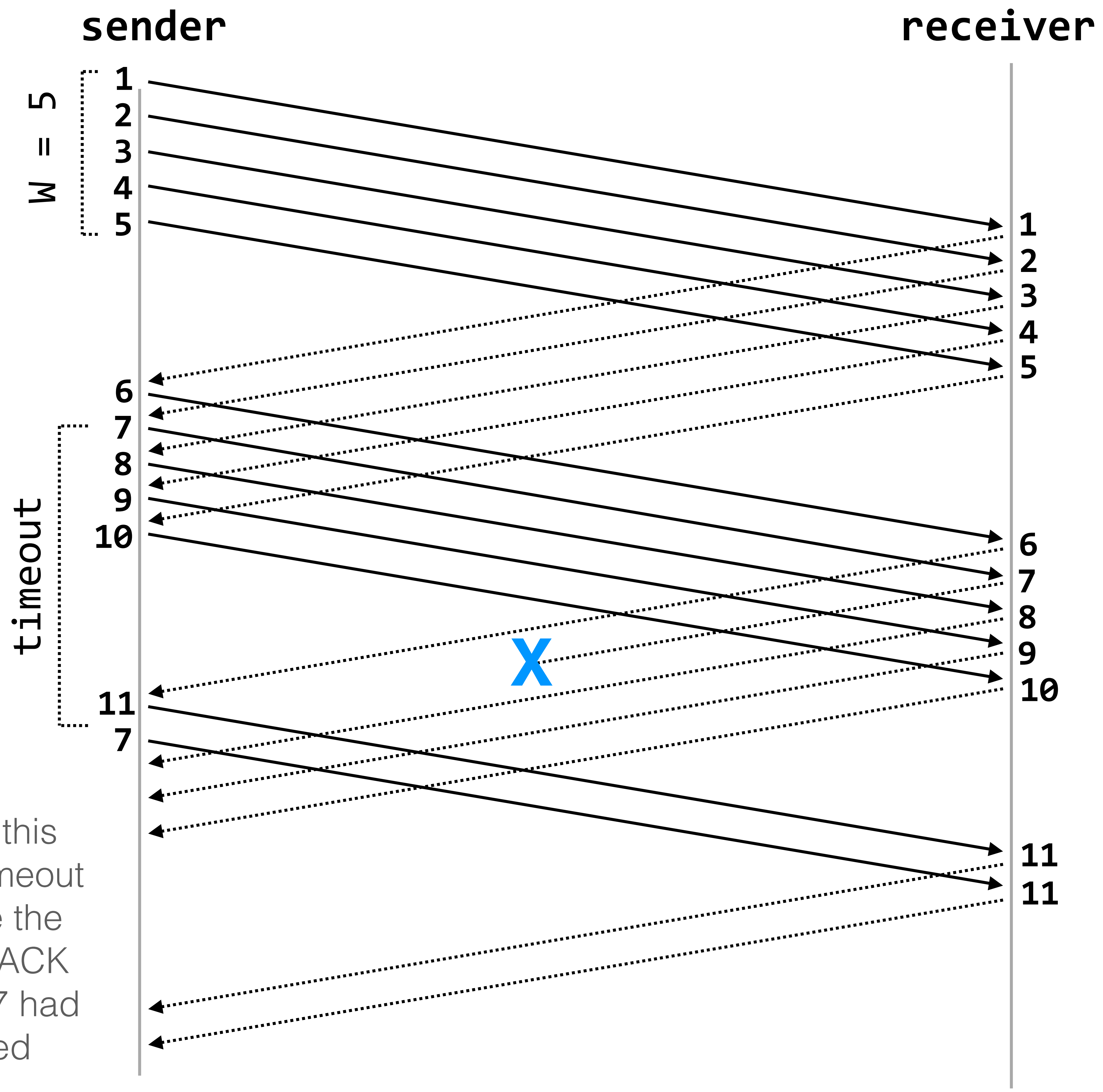
sequence numbers: used to order the packets

acknowledgments ("ACKs"): used to confirm that a packet has been received

an ACK with sequence number k indicates that the receiver has received **all packets up to and including k**

timeouts: used to retransmit packets

reliable transport protocols deliver each byte of data **exactly once, in-order**, to the receiving application



notice that (in this example) the timeout expired before the sender got an ACK indicating that 7 had been received

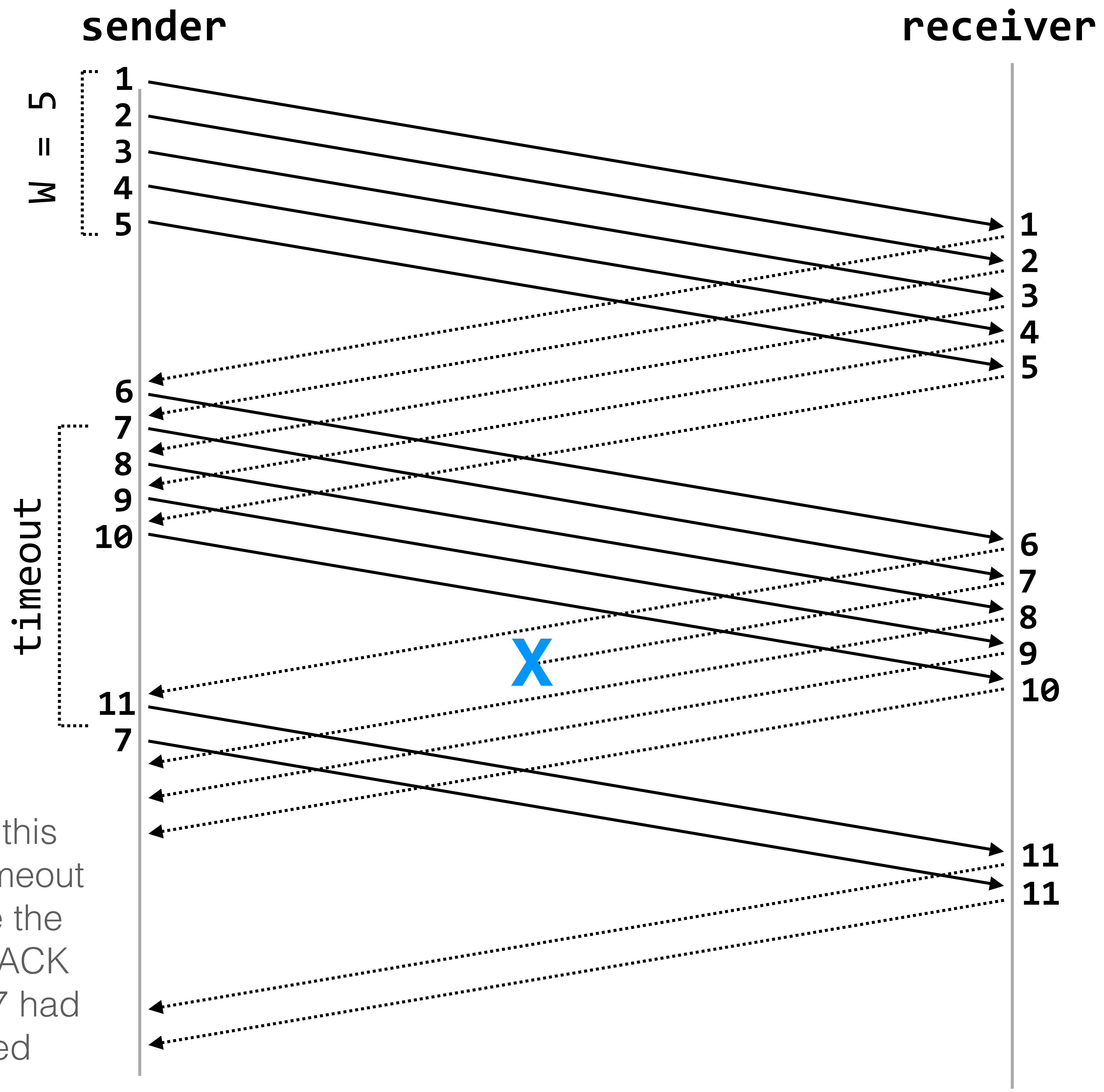
sequence numbers: used to order the packets

acknowledgments ("ACKs"): used to confirm that a packet has been received

an ACK with sequence number k indicates that the receiver has received **all packets up to and including k**

timeouts: used to retransmit packets

reliable transport protocols deliver each byte of data **exactly once, in-order**, to the receiving application



notice that (in this example) the timeout expired before the sender got an ACK indicating that 7 had been received

spurious retransmission: the sender retransmitted a packet that the receiver had already ACKed

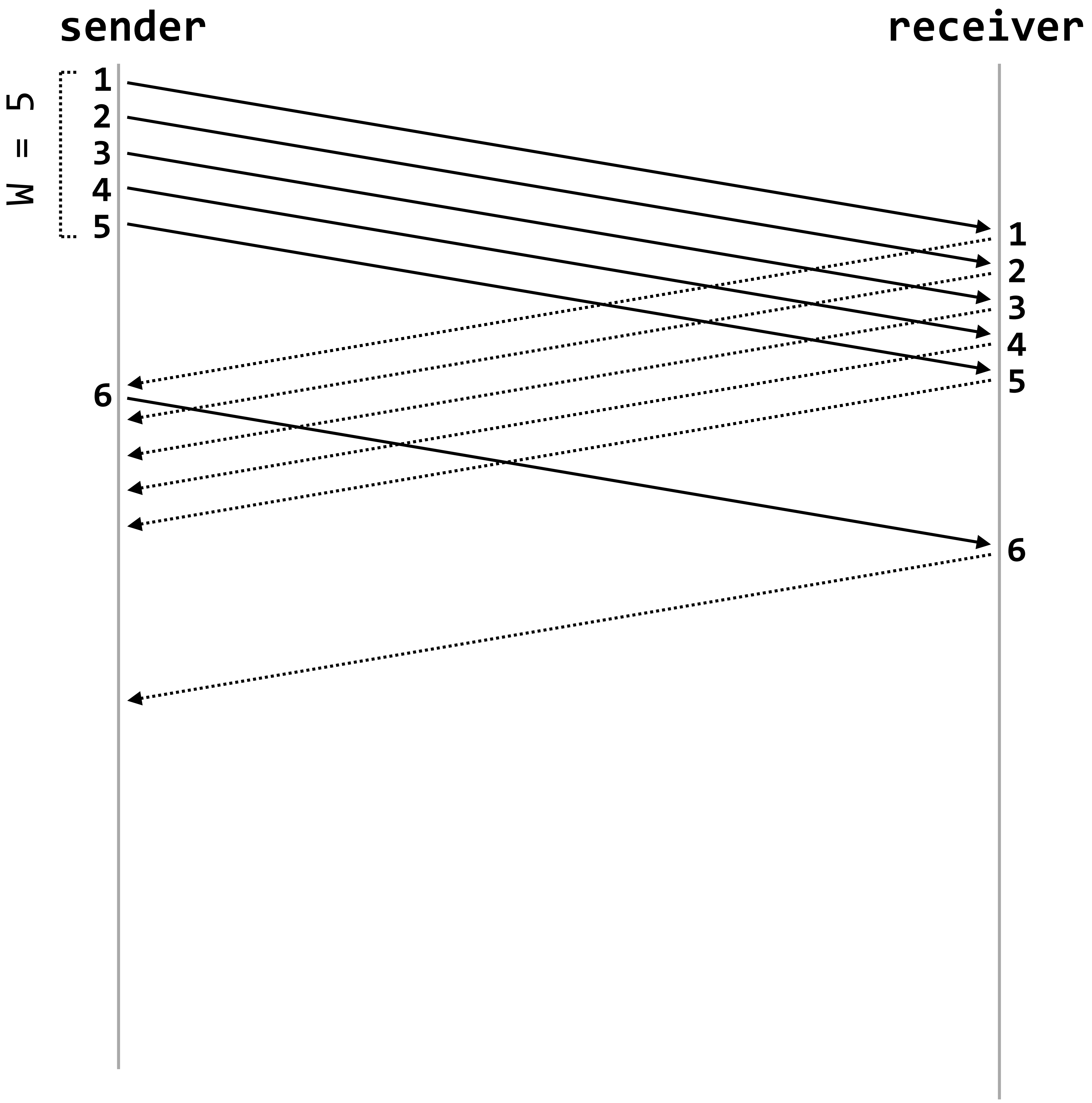
sequence numbers: used to order the packets

acknowledgments ("ACKs"): used to confirm that a packet has been received

an ACK with sequence number k indicates that the receiver has received **all packets up to and including k**

timeouts: used to retransmit packets

reliable transport protocols deliver each byte of data **exactly once, in-order**, to the receiving application



spurious retransmission: the sender retransmitted a packet that the receiver had already ACKed

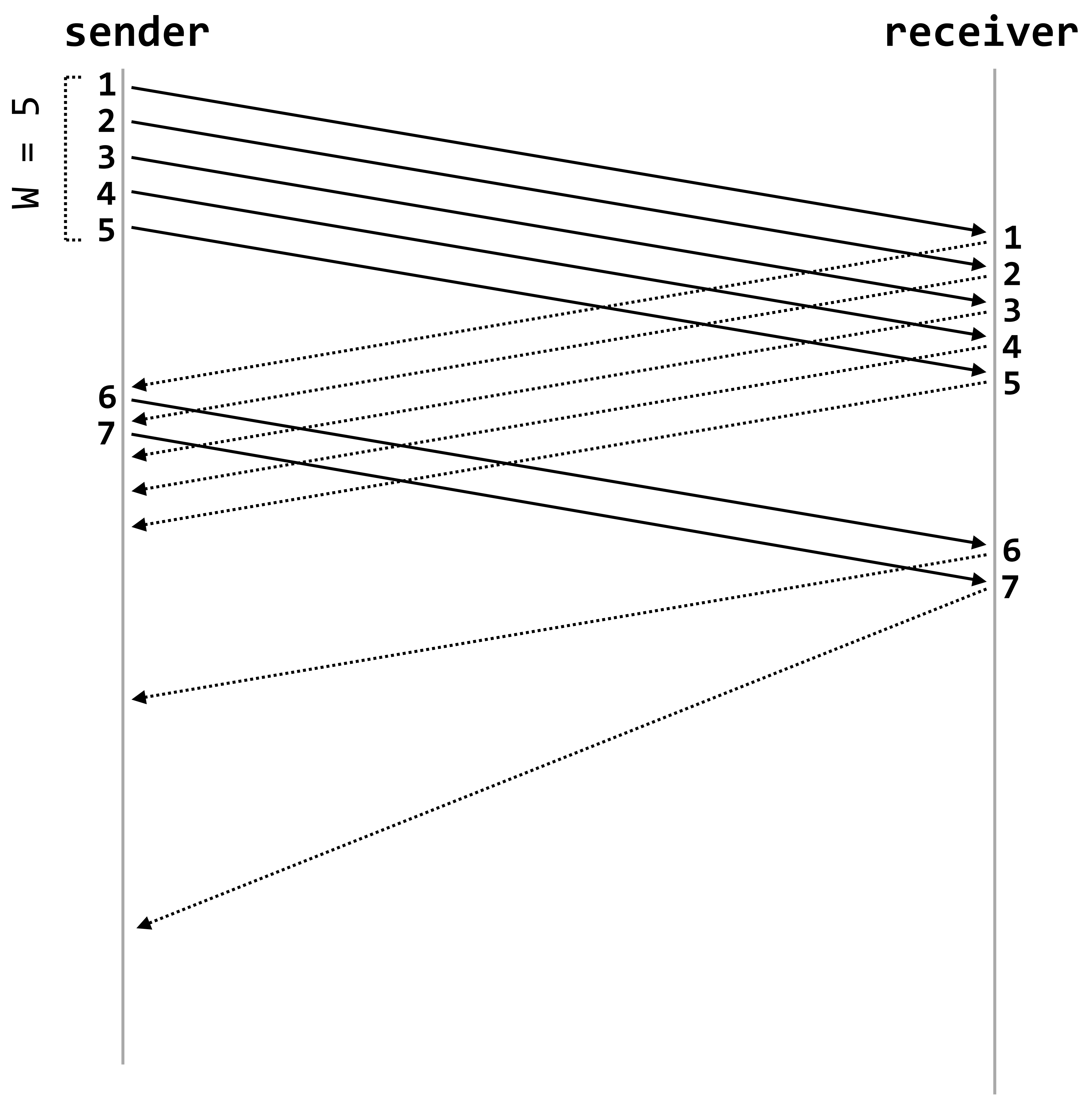
sequence numbers: used to order the packets

acknowledgments ("ACKs"): used to confirm that a packet has been received

an ACK with sequence number k indicates that the receiver has received **all packets up to and including k**

timeouts: used to retransmit packets

reliable transport protocols deliver each byte of data **exactly once, in-order**, to the receiving application



spurious retransmission: the sender retransmitted a packet that the receiver had already ACKed

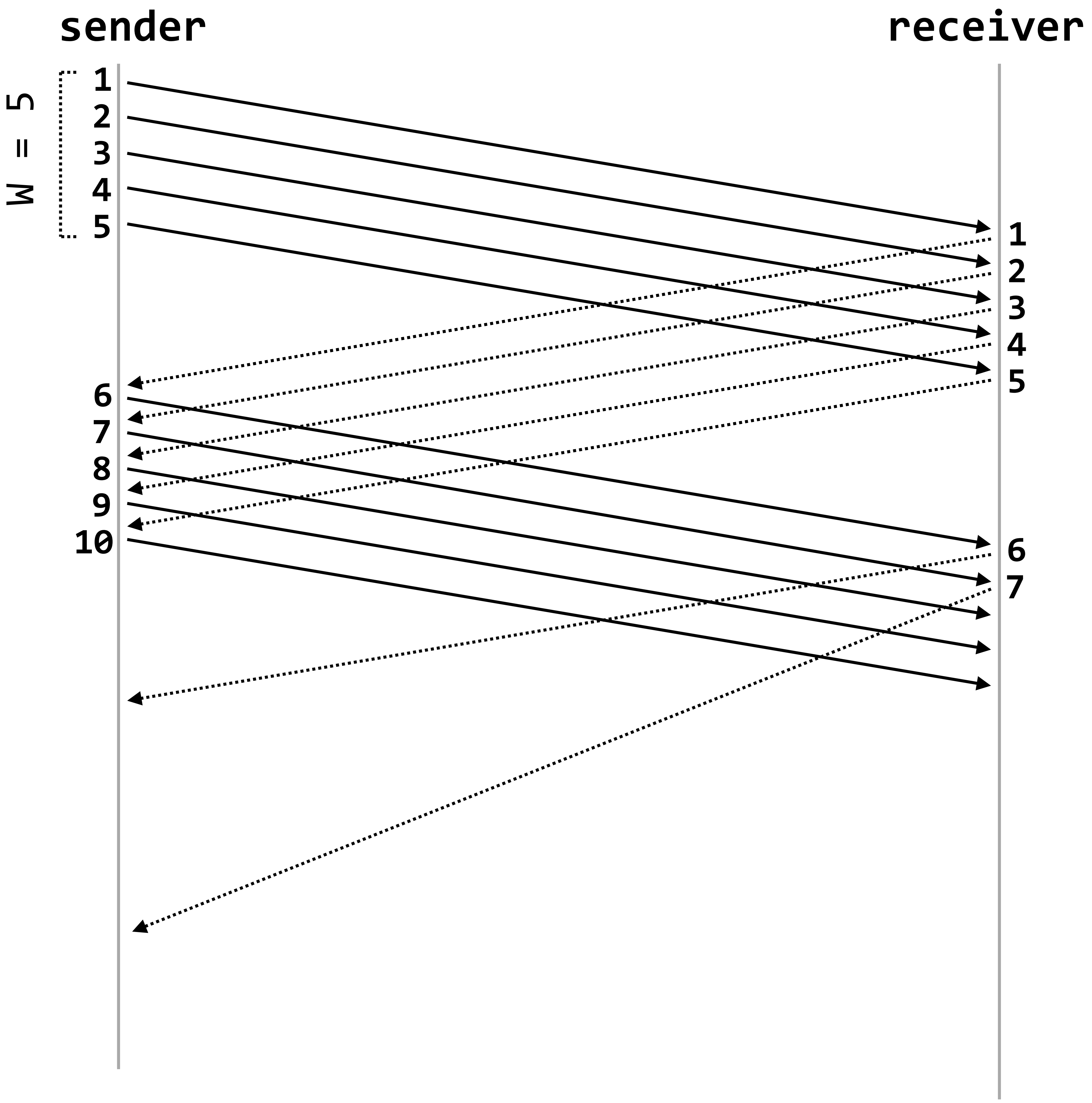
sequence numbers: used to order the packets

acknowledgments ("ACKs"): used to confirm that a packet has been received

an ACK with sequence number k indicates that the receiver has received **all packets up to and including k**

timeouts: used to retransmit packets

reliable transport protocols deliver each byte of data **exactly once, in-order**, to the receiving application



spurious retransmission: the sender retransmitted a packet that the receiver had already ACKed

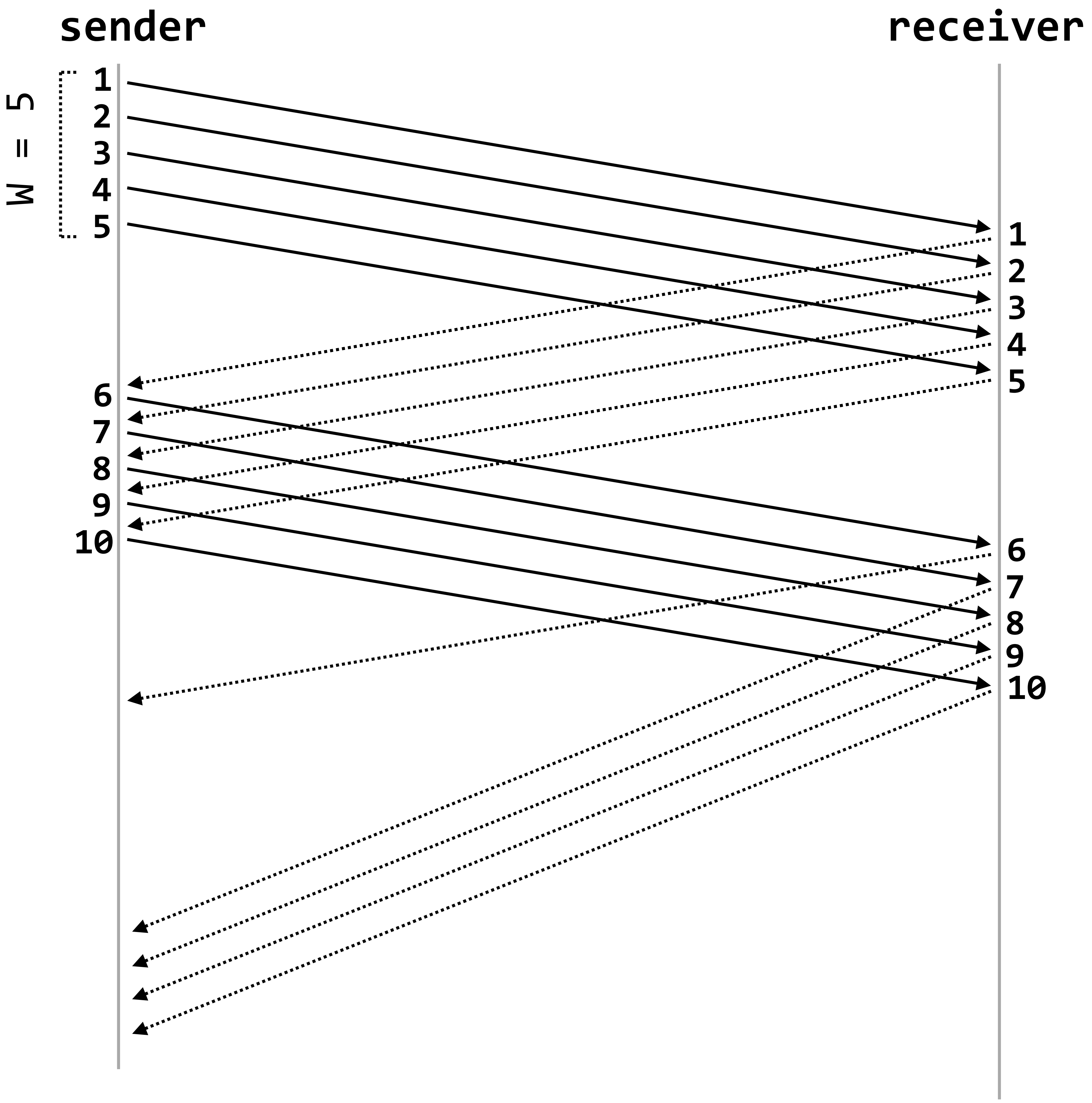
sequence numbers: used to order the packets

acknowledgments ("ACKs"): used to confirm that a packet has been received

an ACK with sequence number k indicates that the receiver has received **all packets up to and including k**

timeouts: used to retransmit packets

reliable transport protocols deliver each byte of data **exactly once, in-order**, to the receiving application



sequence numbers: used to order the packets

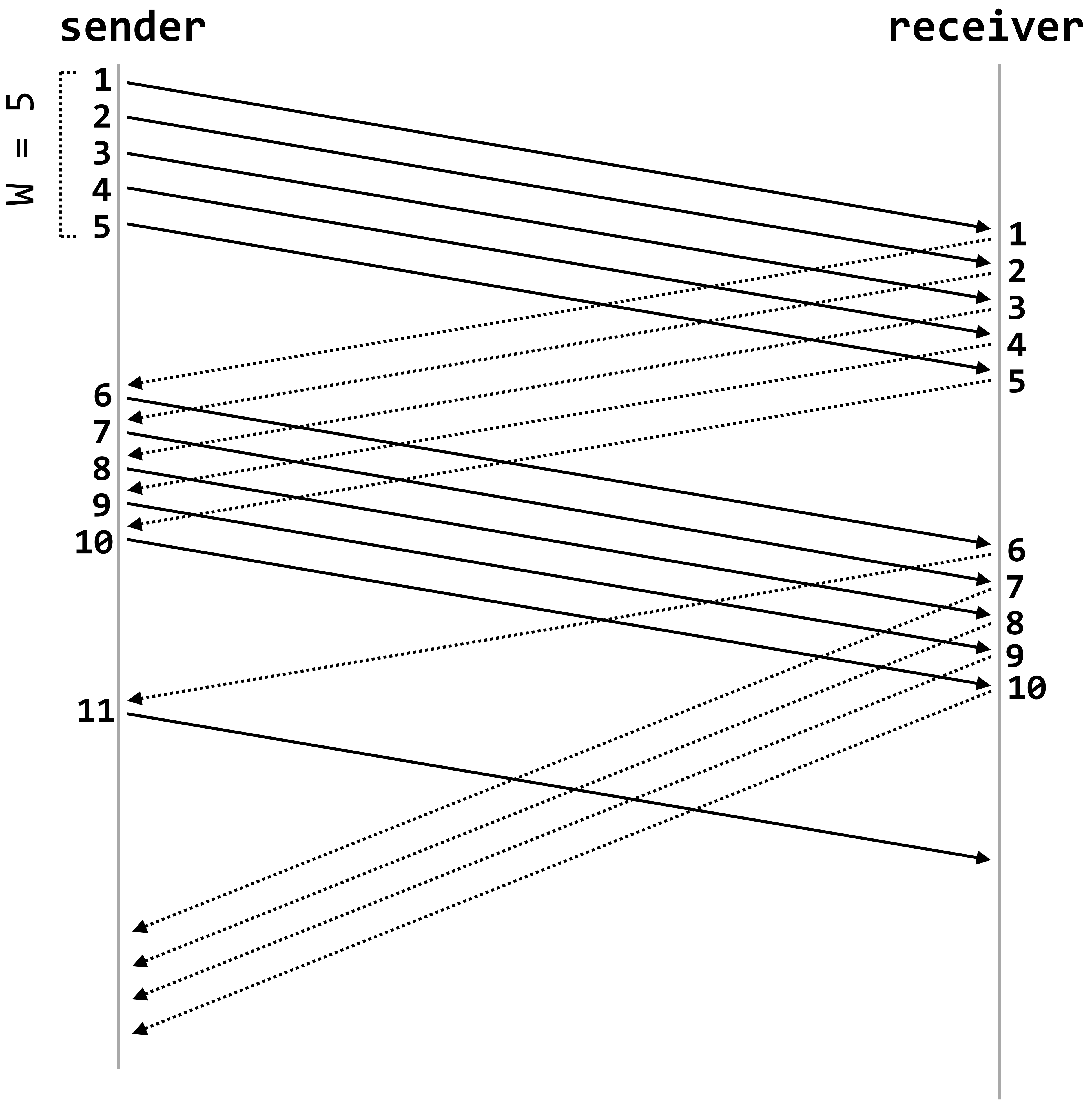
acknowledgments ("ACKs"): used to confirm that a packet has been received

an ACK with sequence number k indicates that the receiver has received **all packets up to and including k**

timeouts: used to retransmit packets

spurious retransmission: the sender retransmitted a packet that the receiver had already ACKed

reliable transport protocols deliver each byte of data **exactly once, in-order**, to the receiving application



spurious retransmission: the sender retransmitted a packet that the receiver had already ACKed

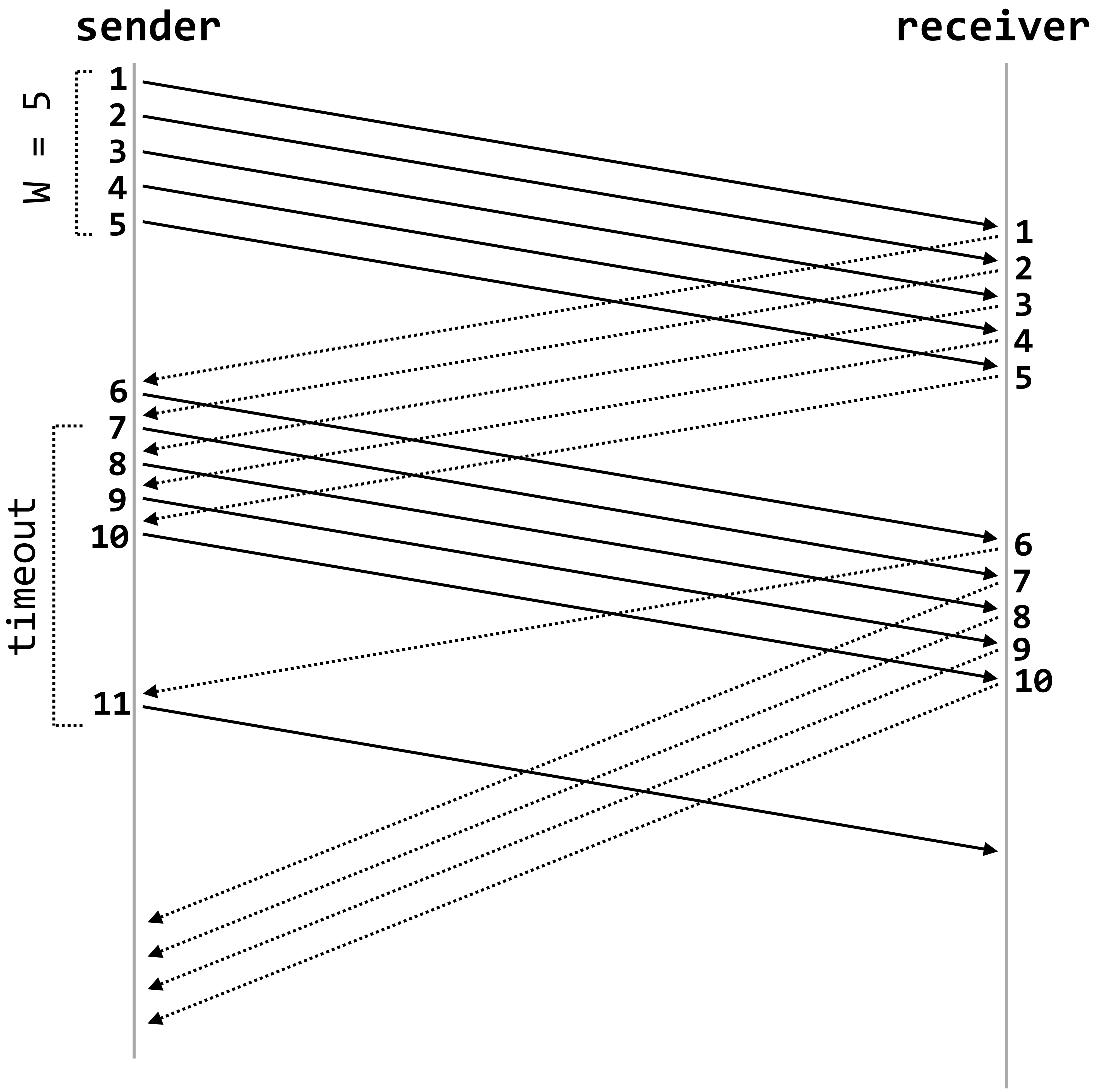
sequence numbers: used to order the packets

acknowledgments ("ACKs"): used to confirm that a packet has been received

an ACK with sequence number k indicates that the receiver has received **all packets up to and including k**

timeouts: used to retransmit packets

reliable transport protocols deliver each byte of data **exactly once, in-order**, to the receiving application



spurious retransmission: the sender retransmitted a packet that the receiver had already ACKed

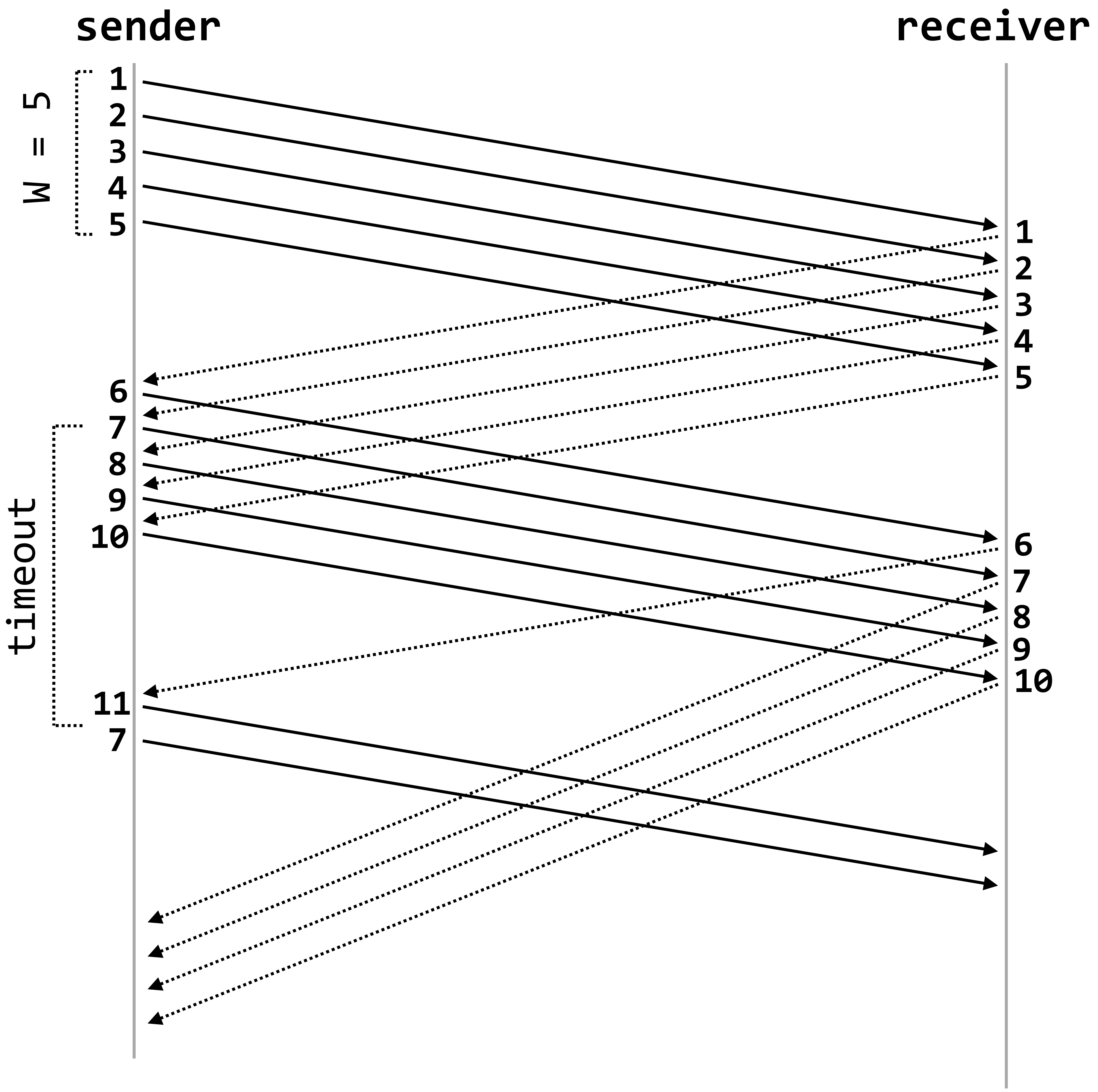
sequence numbers: used to order the packets

acknowledgments ("ACKs"): used to confirm that a packet has been received

an ACK with sequence number k indicates that the receiver has received **all packets up to and including k**

timeouts: used to retransmit packets

reliable transport protocols deliver each byte of data **exactly once, in-order**, to the receiving application



spurious retransmission: the sender retransmitted a packet that the receiver had already ACKed

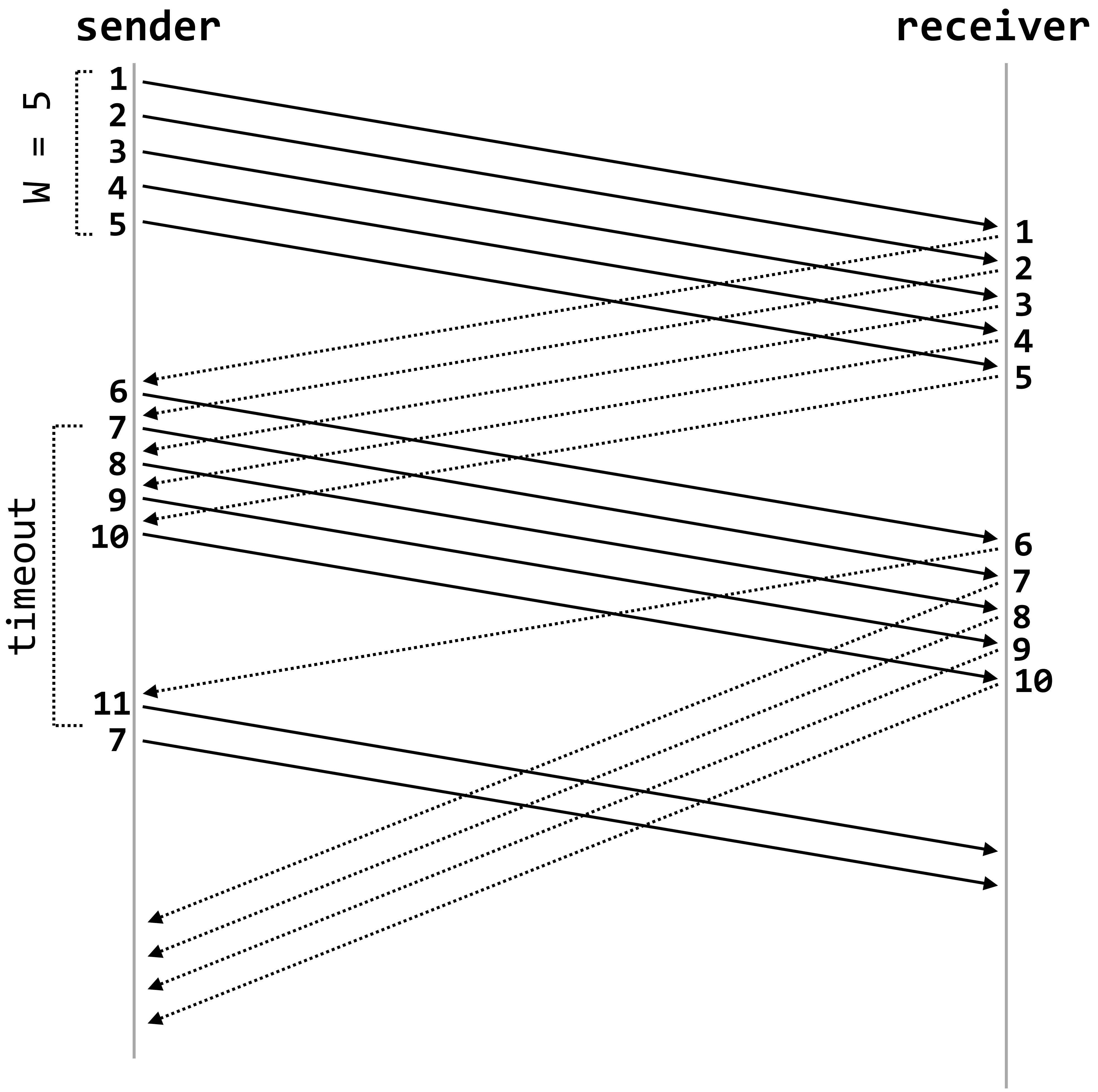
sequence numbers: used to order the packets

acknowledgments ("ACKs"): used to confirm that a packet has been received

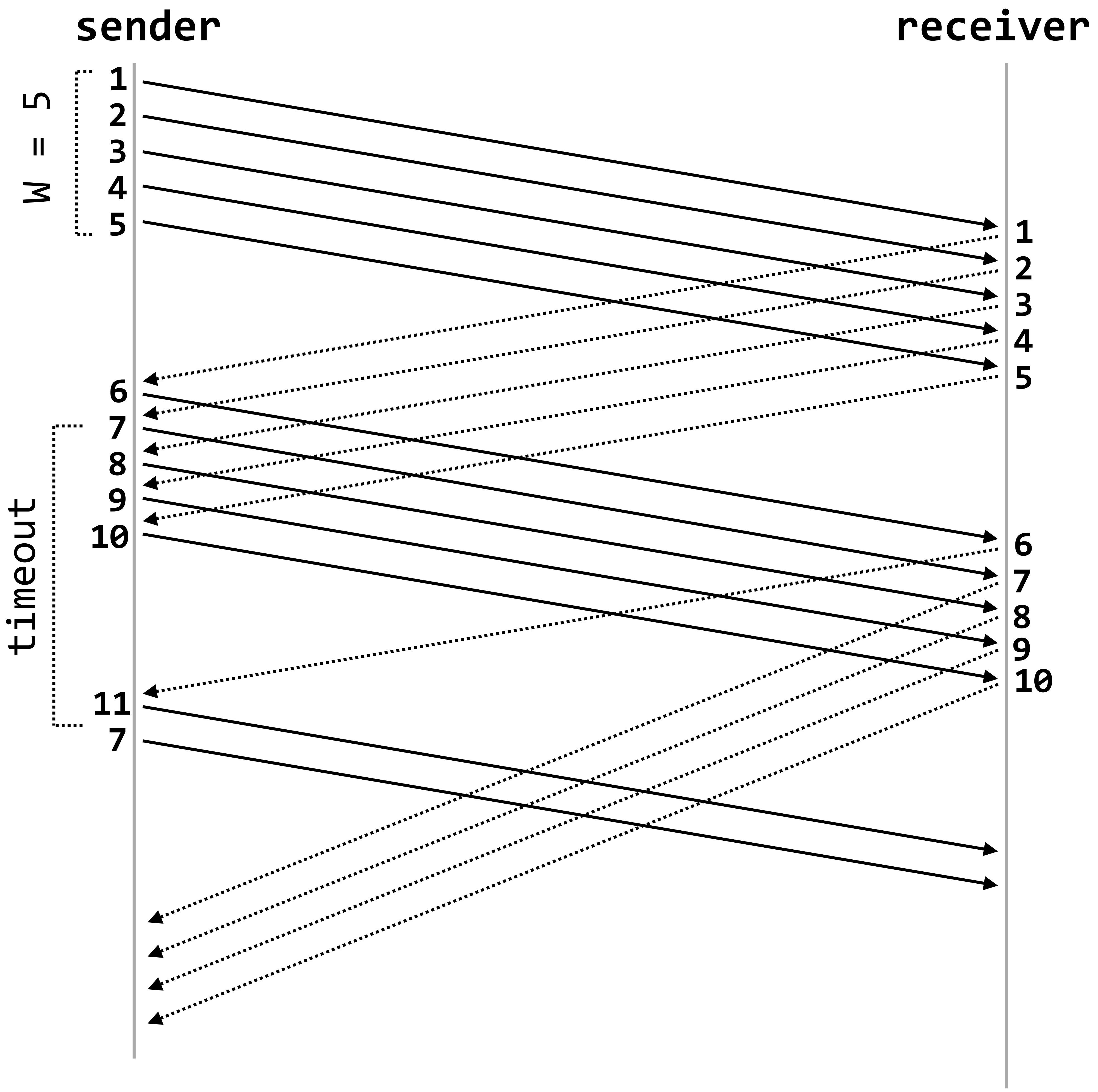
an ACK with sequence number k indicates that the receiver has received **all packets up to and including k**

timeouts: used to retransmit packets

reliable transport protocols deliver each byte of data **exactly once, in-order**, to the receiving application

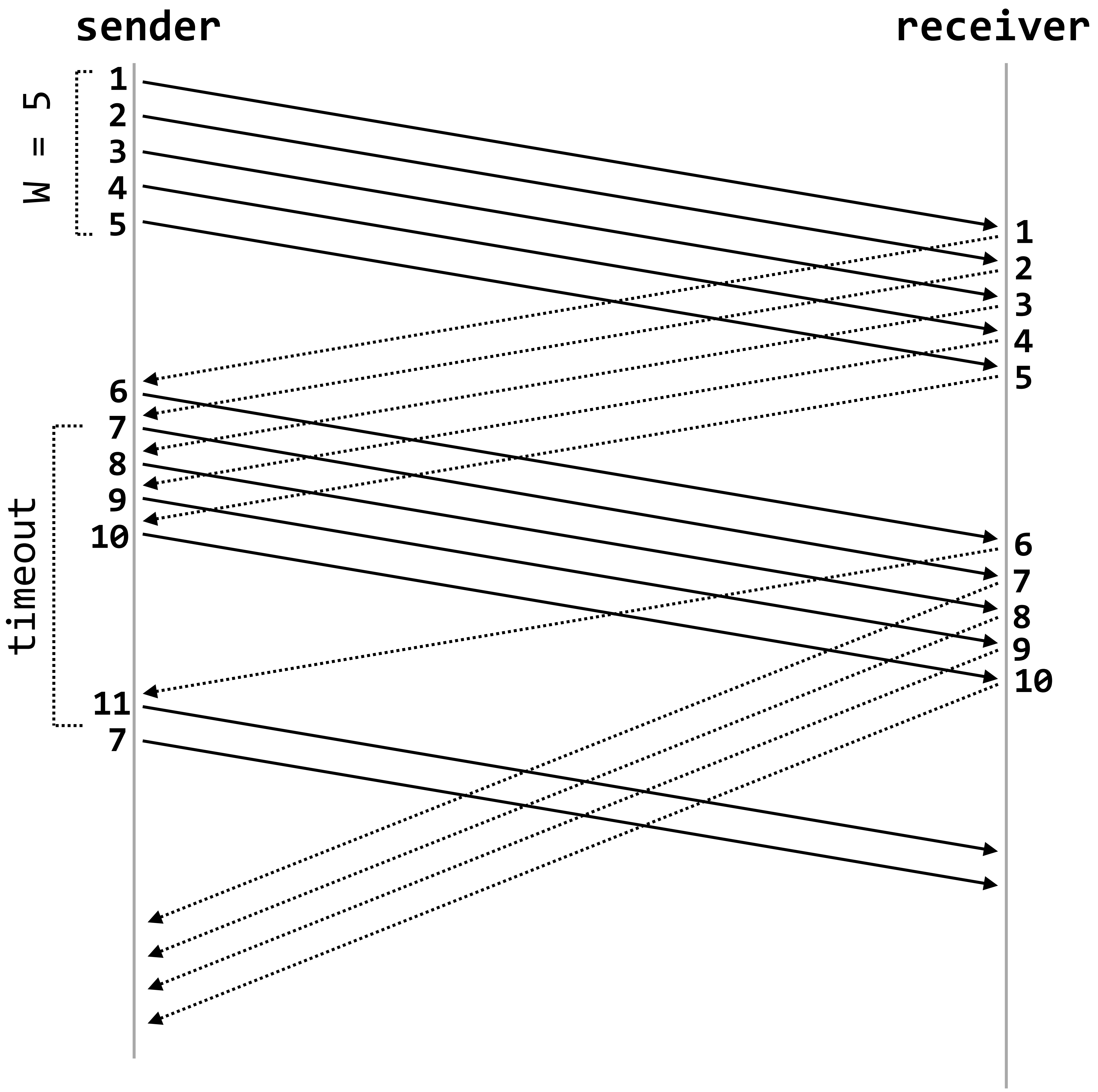


reliable transport protocols deliver each byte of data **exactly once, in-order**, to the receiving application



question: what should `W` be?

reliable transport protocols deliver each byte of data **exactly once, in-order**, to the receiving application



question: what should `W` be?

how can a single reliable sender, using a sliding-window protocol, set its window size to **maximize utilization — but prevent congestion and unfairness** — given that there are many other end points using the network, all with different, changing demands?

congestion control: controlling the source rates to achieve **efficiency** and **fairness**

congestion control: controlling the source rates to achieve **efficiency** and **fairness**

efficiency: minimize drops, minimize delay, maximize bottleneck utilization

congestion control: controlling the source rates to achieve **efficiency** and **fairness**

efficiency: minimize drops, minimize delay, maximize bottleneck utilization

fairness: under infinite offered load, split bandwidth evenly among all sources sharing a bottleneck

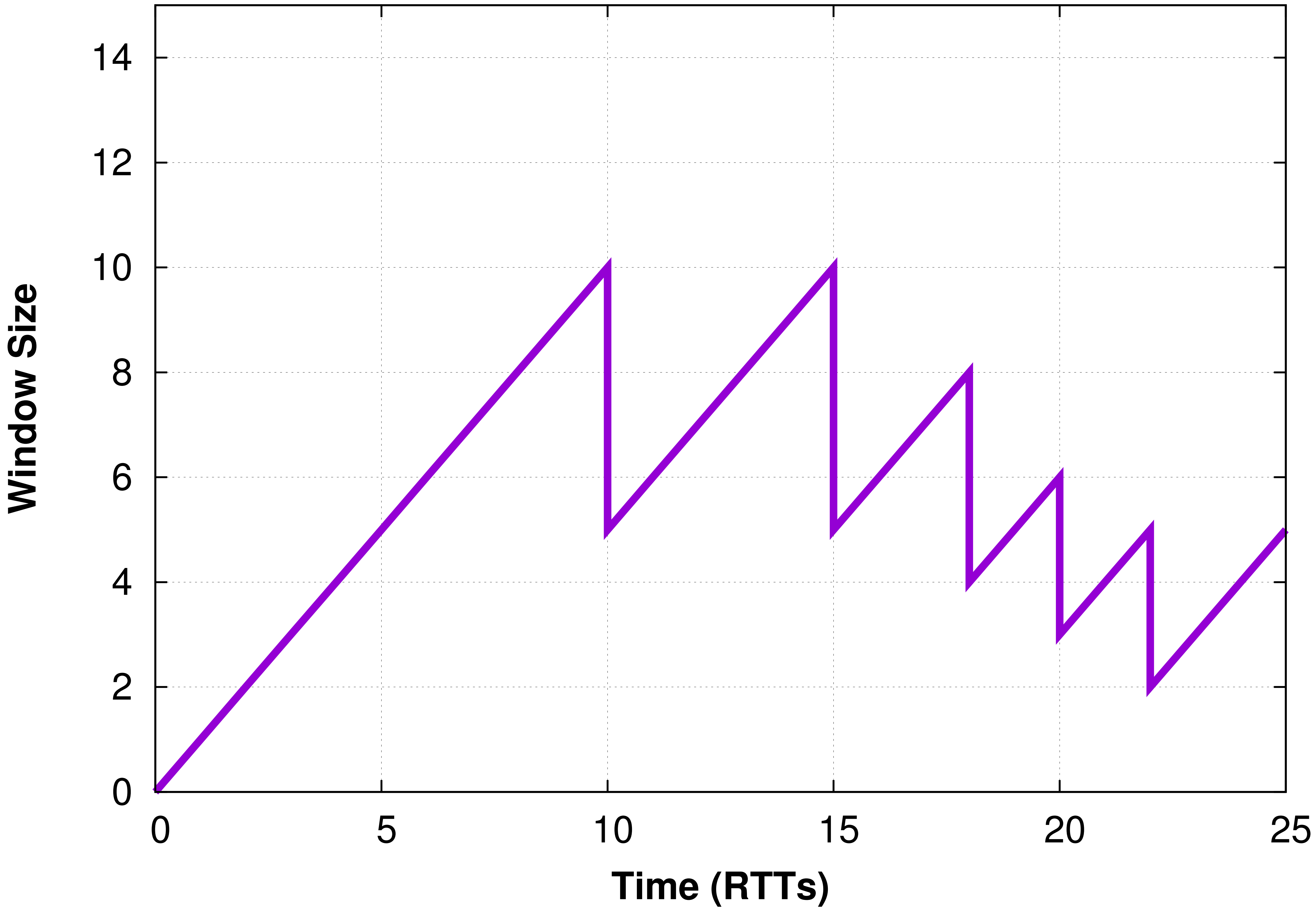
congestion control: controlling the source rates to achieve **efficiency** and **fairness**

efficiency: minimize drops, minimize delay, maximize bottleneck utilization

fairness: under infinite offered load, split bandwidth evenly among all sources sharing a bottleneck

AIMD: every RTT, if there is no loss,
`W = W + 1`; else, `W = W/2`

congestion control: controlling the source rates to achieve **efficiency** and **fairness**

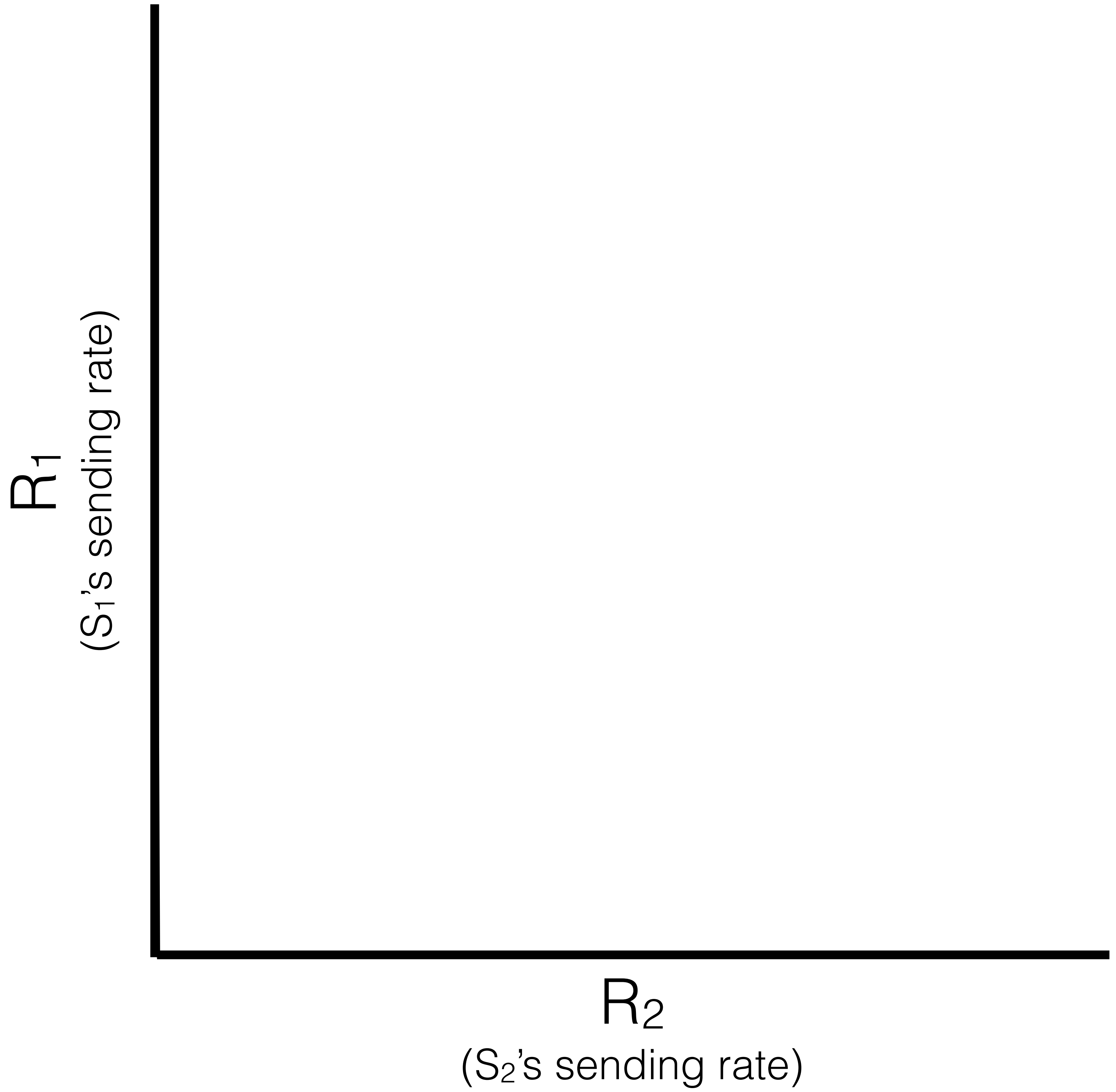


efficiency: minimize drops, minimize delay, maximize bottleneck utilization

fairness: under infinite offered load, split bandwidth evenly among all sources sharing a bottleneck

AIMD: every RTT, if there is no loss, `W = W + 1`; else, `W = W/2`

congestion control: controlling the source rates to achieve **efficiency** and **fairness**

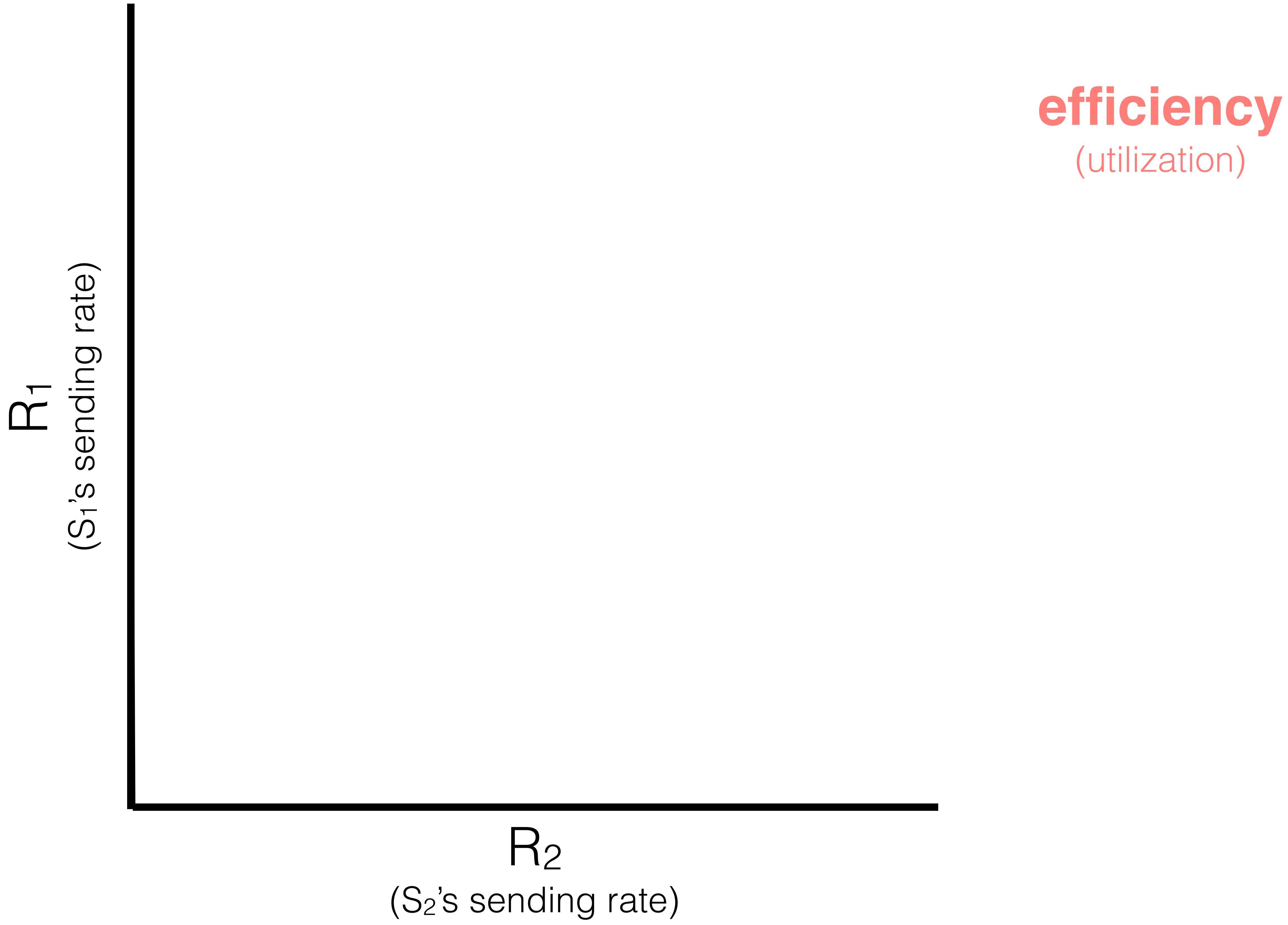


efficiency: minimize drops, minimize delay, maximize bottleneck utilization

fairness: under infinite offered load, split bandwidth evenly among all sources sharing a bottleneck

AIMD: every RTT, if there is no loss, `W = W + 1`; else, `W = W/2`

congestion control: controlling the source rates to achieve **efficiency** and **fairness**

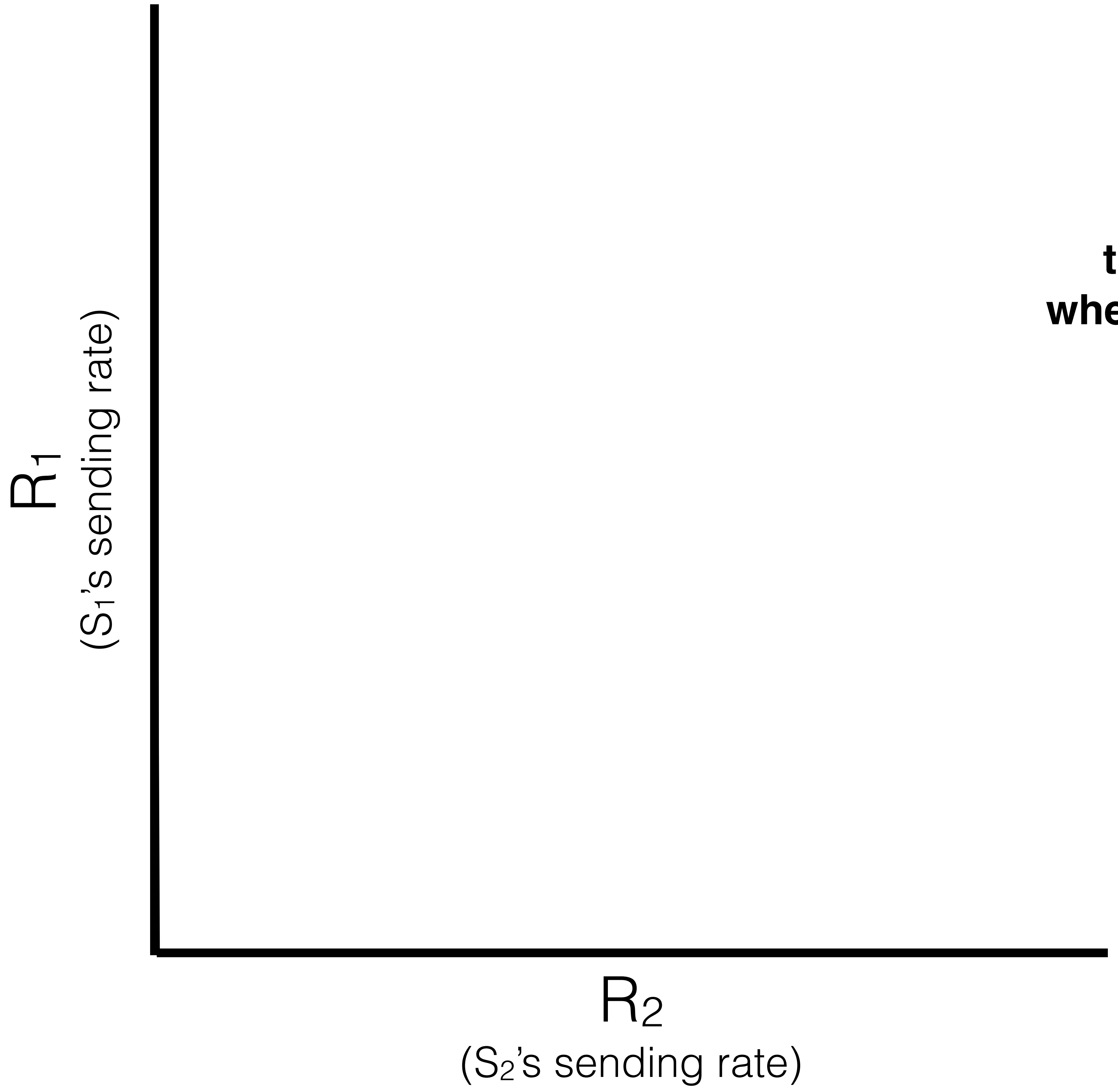


efficiency: minimize drops, minimize delay, maximize bottleneck utilization

fairness: under infinite offered load, split bandwidth evenly among all sources sharing a bottleneck

AIMD: every RTT, if there is no loss, `W = W + 1`; else, `W = W/2`

congestion control: controlling the source rates to achieve **efficiency** and **fairness**



efficiency

(utilization)

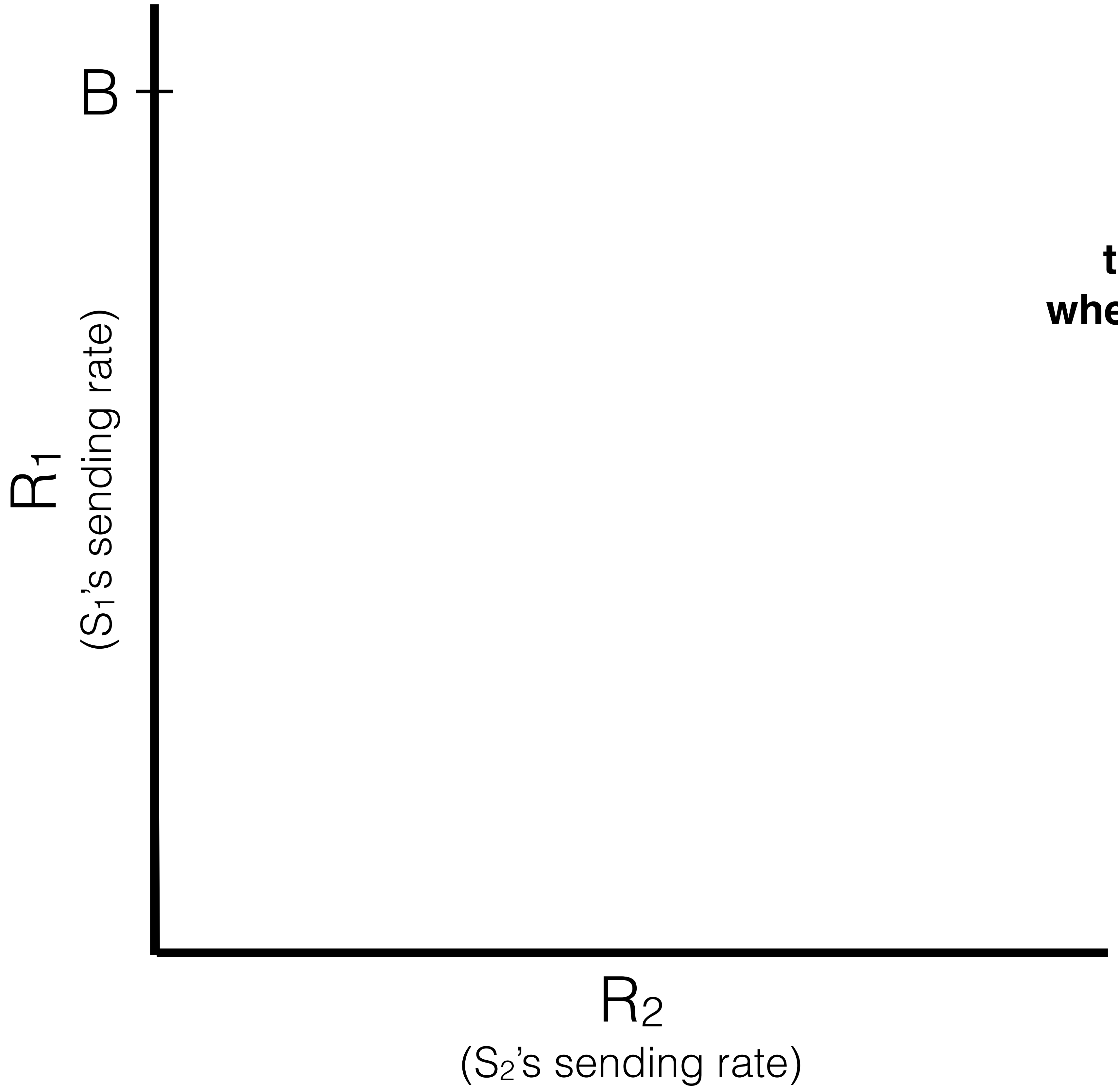
the network is fully utilized when the bottleneck link is "full"

efficiency: minimize drops, minimize delay, maximize bottleneck utilization

fairness: under infinite offered load, split bandwidth evenly among all sources sharing a bottleneck

AIMD: every RTT, if there is no loss, `W = W + 1`; else, `W = W/2`

congestion control: controlling the source rates to achieve **efficiency** and **fairness**



efficiency

(utilization)

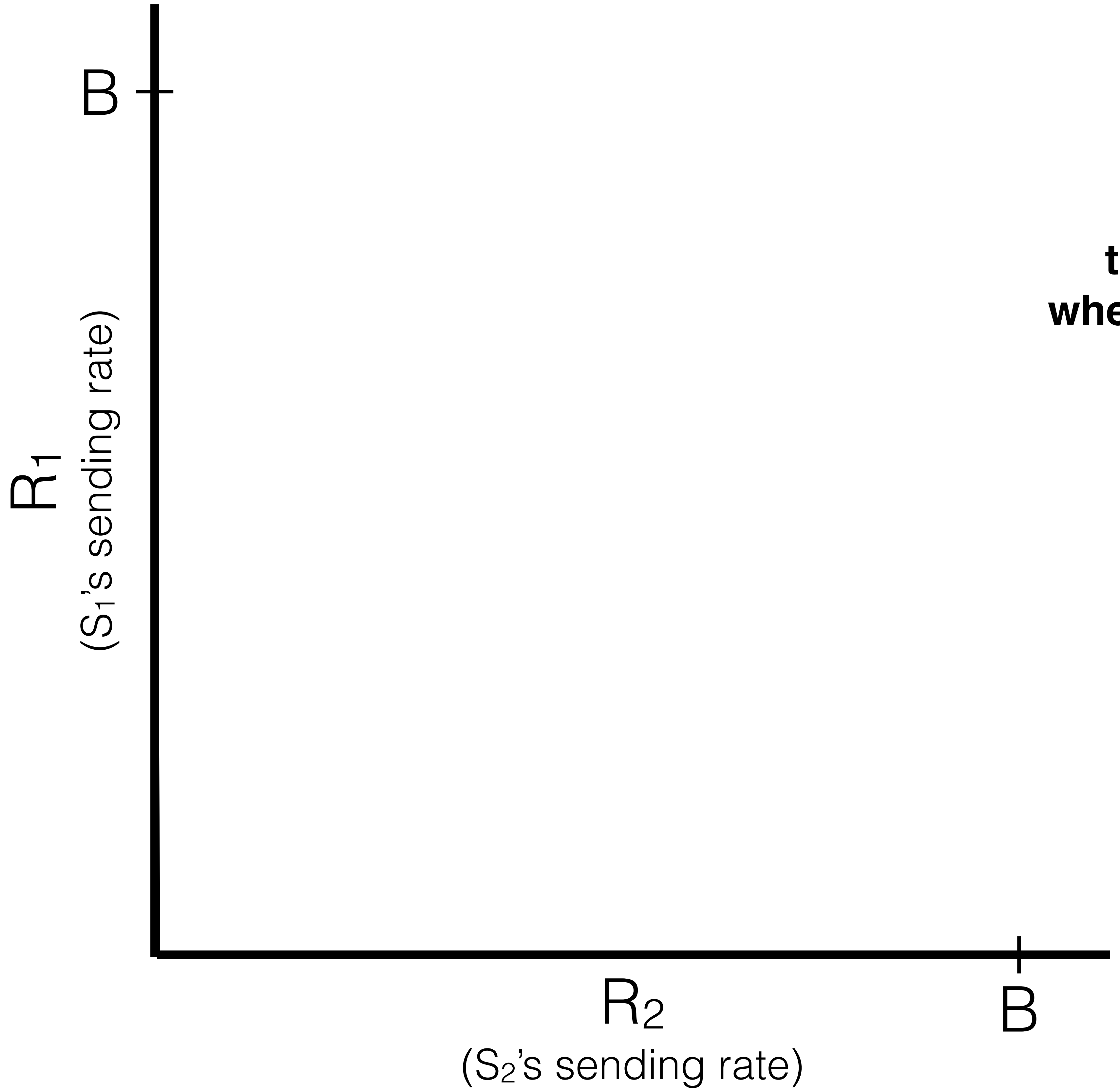
the network is fully utilized when the bottleneck link is "full"

efficiency: minimize drops, minimize delay, maximize bottleneck utilization

fairness: under infinite offered load, split bandwidth evenly among all sources sharing a bottleneck

AIMD: every RTT, if there is no loss, `W = W + 1`; else, `W = W/2`

congestion control: controlling the source rates to achieve **efficiency** and **fairness**



efficiency

(utilization)

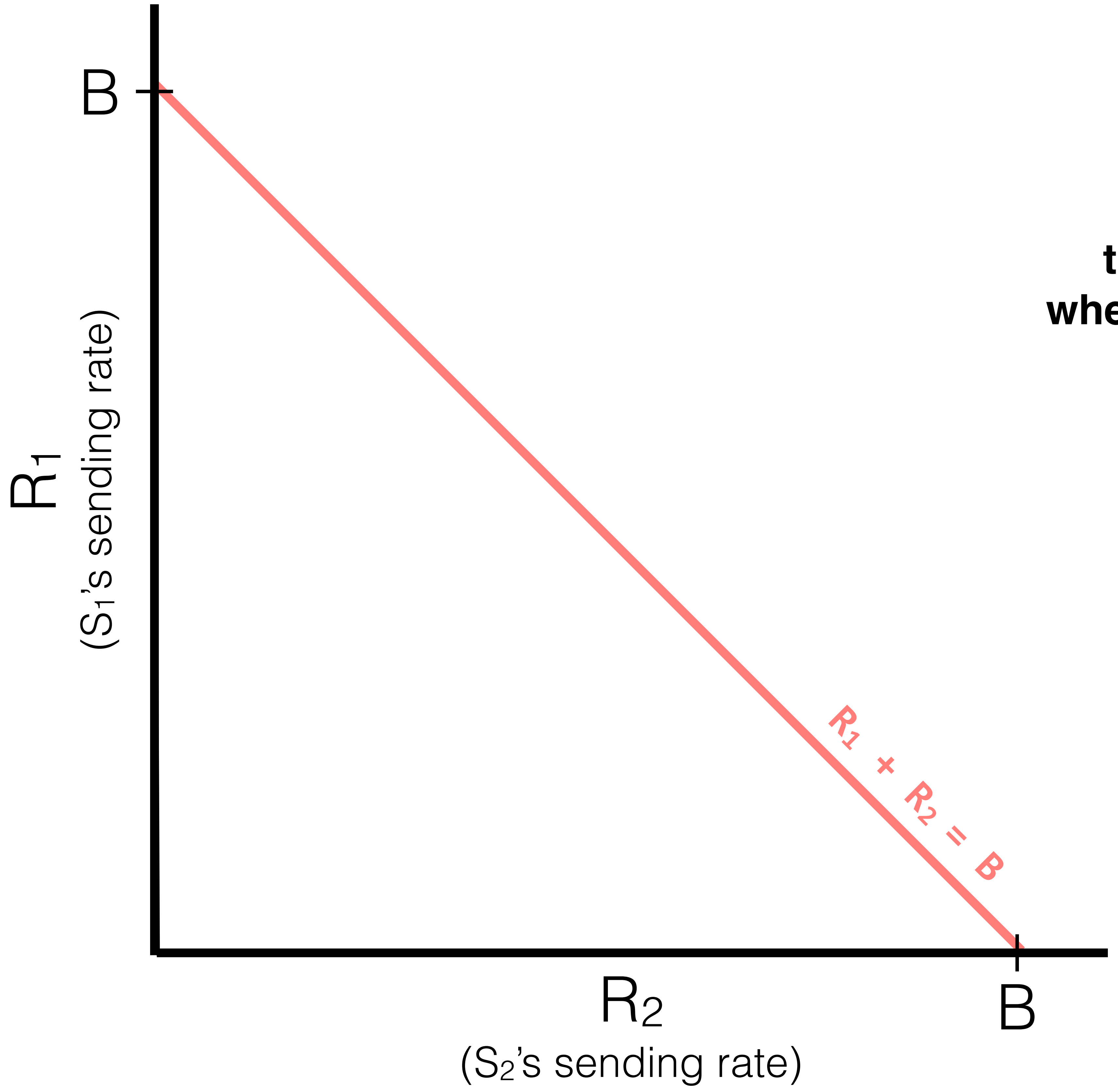
the network is fully utilized when the bottleneck link is "full"

efficiency: minimize drops, minimize delay, maximize bottleneck utilization

fairness: under infinite offered load, split bandwidth evenly among all sources sharing a bottleneck

AIMD: every RTT, if there is no loss, `W = W + 1`; else, `W = W/2`

congestion control: controlling the source rates to achieve **efficiency** and **fairness**



efficiency

(utilization)

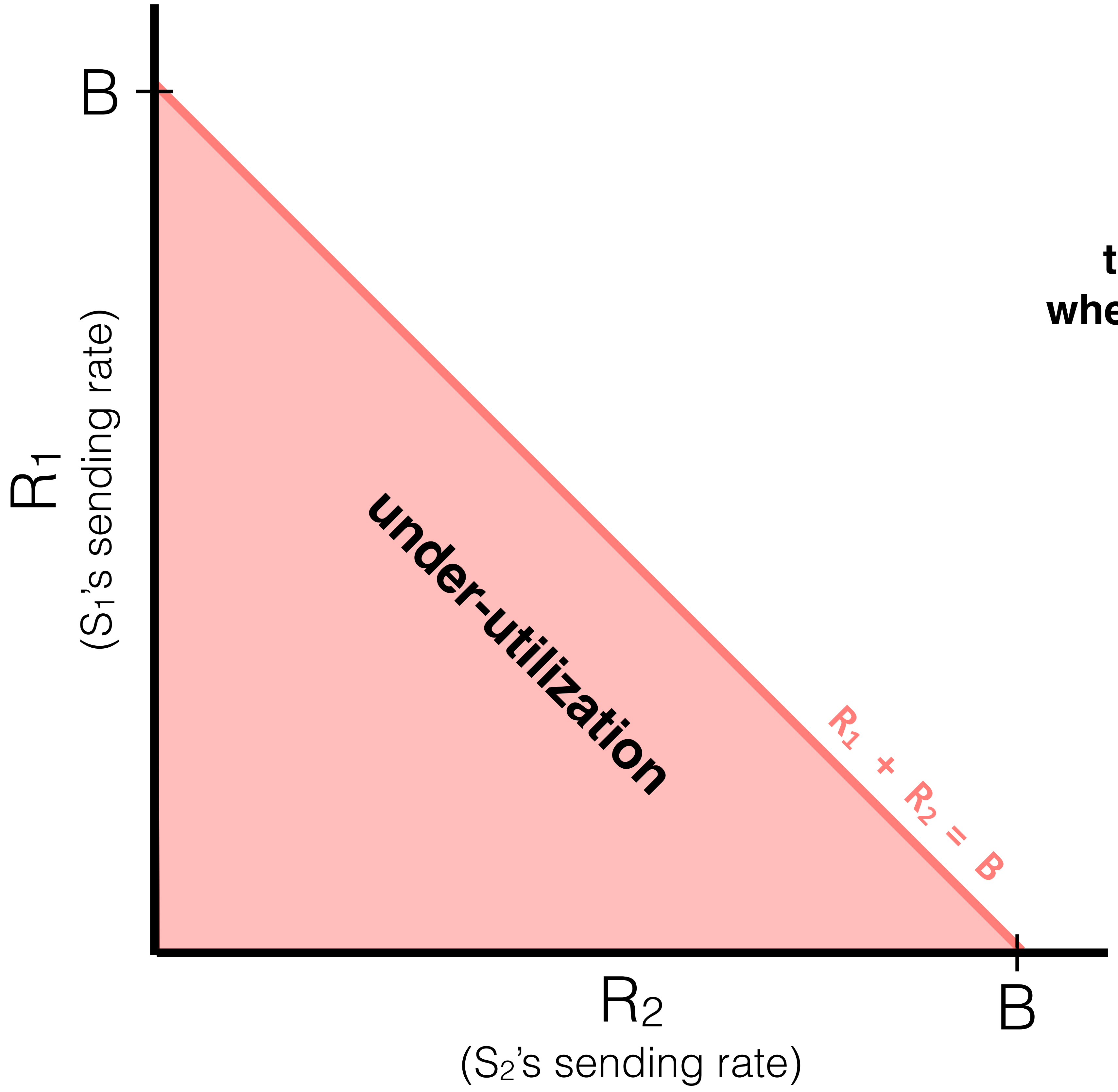
the network is fully utilized when the bottleneck link is "full"

efficiency: minimize drops, minimize delay, maximize bottleneck utilization

fairness: under infinite offered load, split bandwidth evenly among all sources sharing a bottleneck

AIMD: every RTT, if there is no loss, `W = W + 1`; else, `W = W/2`

congestion control: controlling the source rates to achieve **efficiency** and **fairness**



efficiency

(utilization)

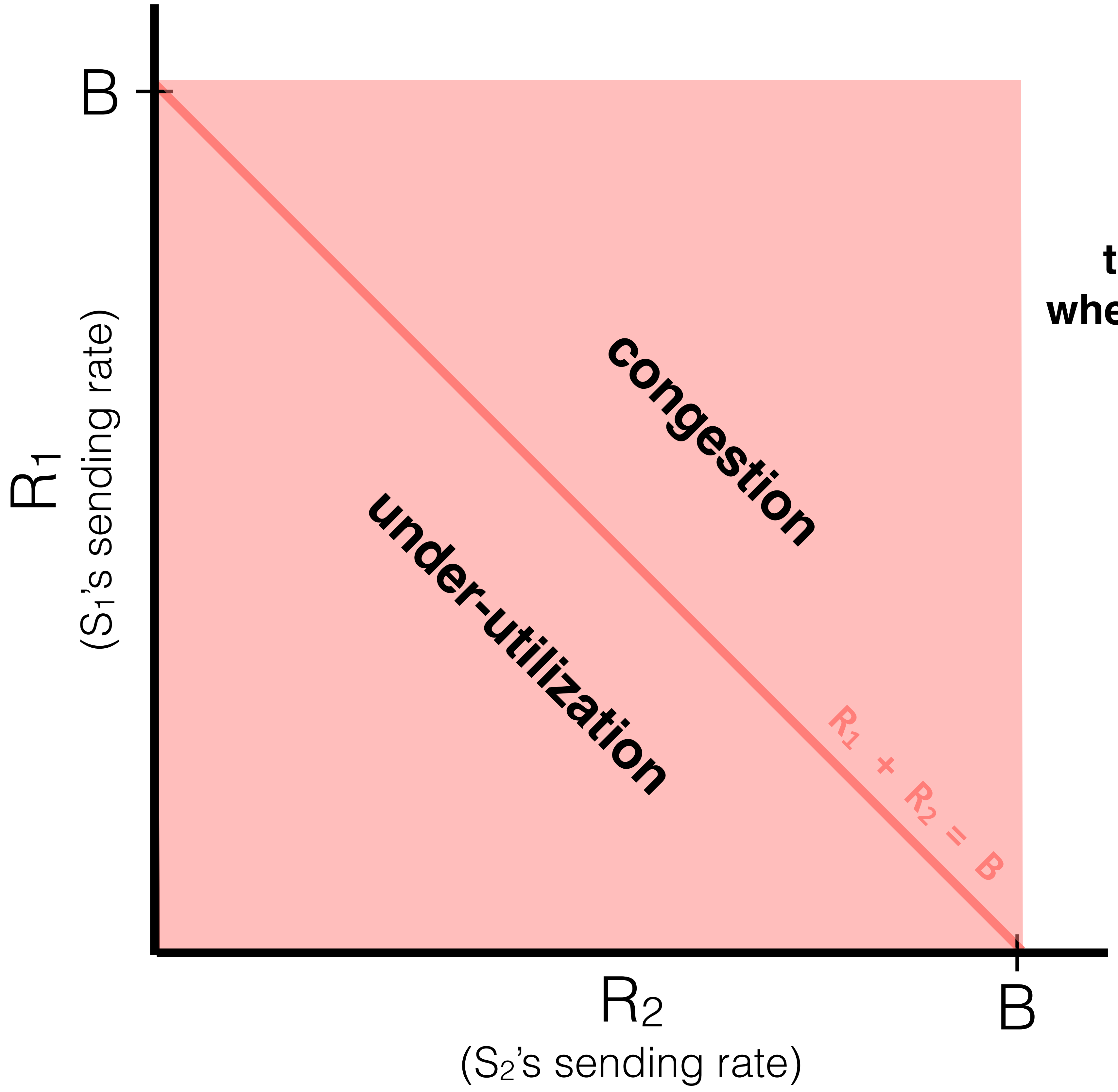
the network is fully utilized when the bottleneck link is "full"

efficiency: minimize drops, minimize delay, maximize bottleneck utilization

fairness: under infinite offered load, split bandwidth evenly among all sources sharing a bottleneck

AIMD: every RTT, if there is no loss, `W = W + 1`; else, `W = W/2`

congestion control: controlling the source rates to achieve **efficiency** and **fairness**



efficiency

(utilization)

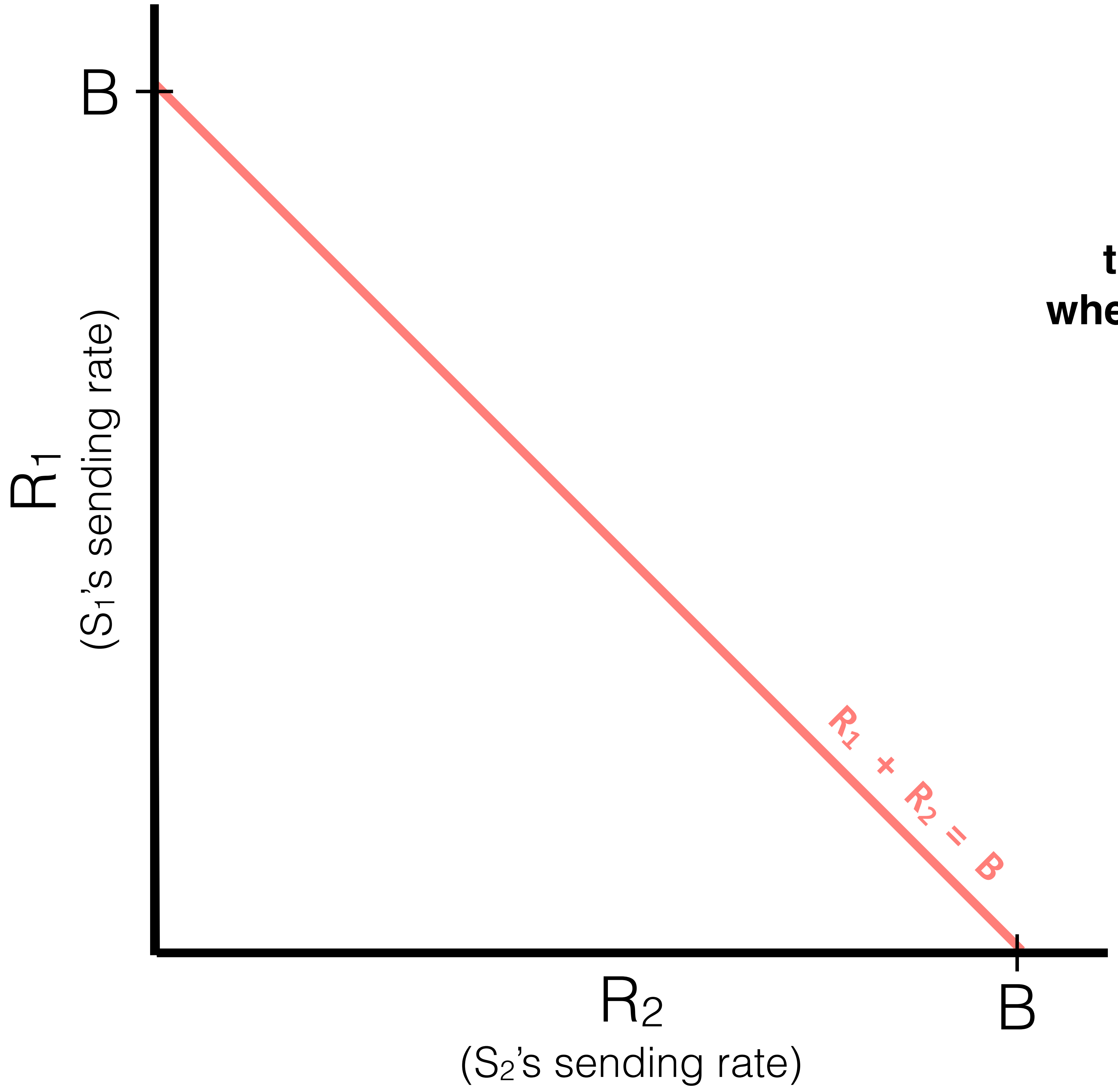
the network is fully utilized when the bottleneck link is "full"

efficiency: minimize drops, minimize delay, maximize bottleneck utilization

fairness: under infinite offered load, split bandwidth evenly among all sources sharing a bottleneck

AIMD: every RTT, if there is no loss, `W = W + 1`; else, `W = W/2`

congestion control: controlling the source rates to achieve **efficiency** and **fairness**



efficiency

(utilization)

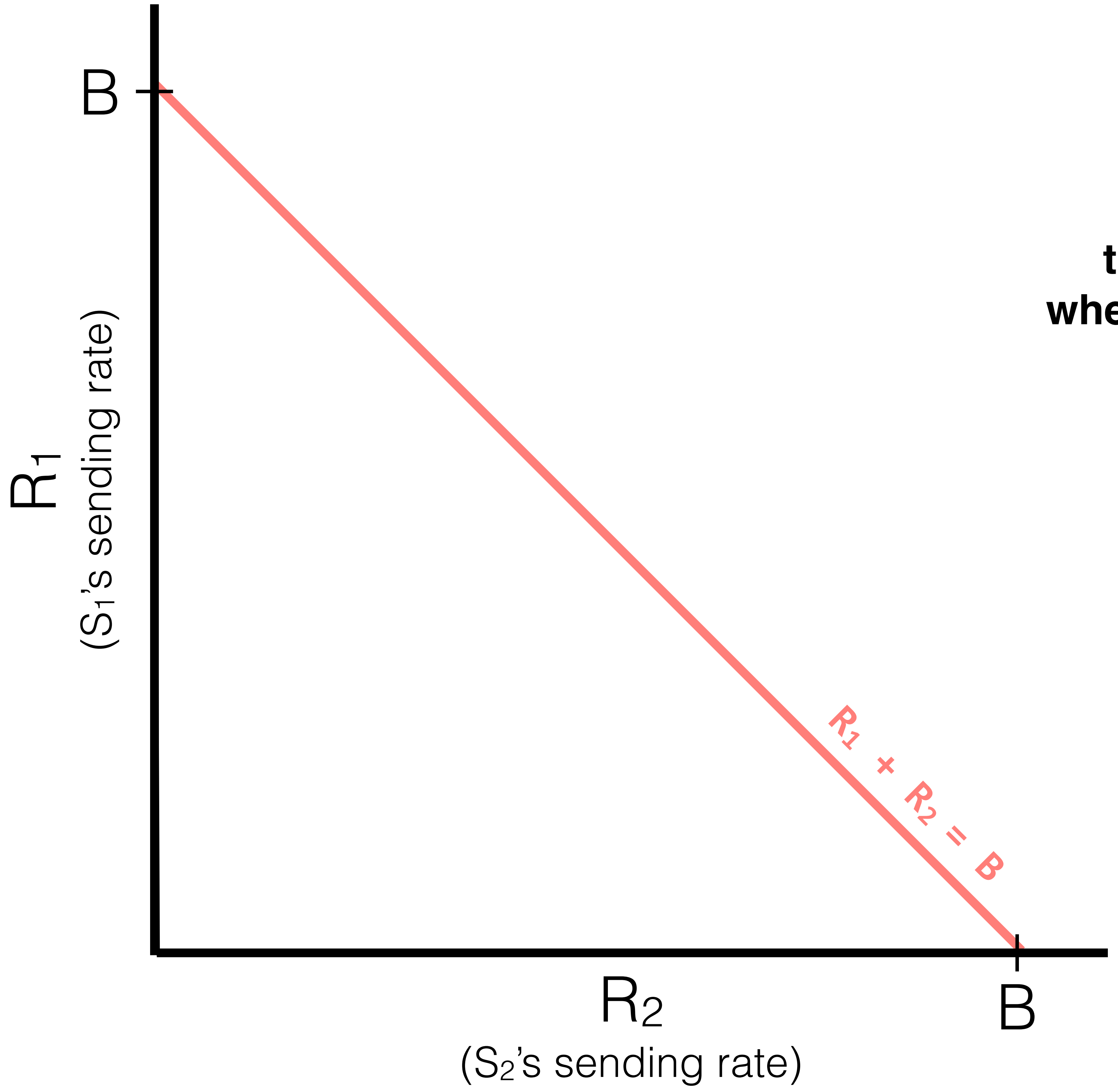
the network is fully utilized when the bottleneck link is "full"

efficiency: minimize drops, minimize delay, maximize bottleneck utilization

fairness: under infinite offered load, split bandwidth evenly among all sources sharing a bottleneck

AIMD: every RTT, if there is no loss, `W = W + 1`; else, `W = W/2`

congestion control: controlling the source rates to achieve **efficiency** and **fairness**



efficiency

(utilization)

the network is fully utilized when the bottleneck link is "full"

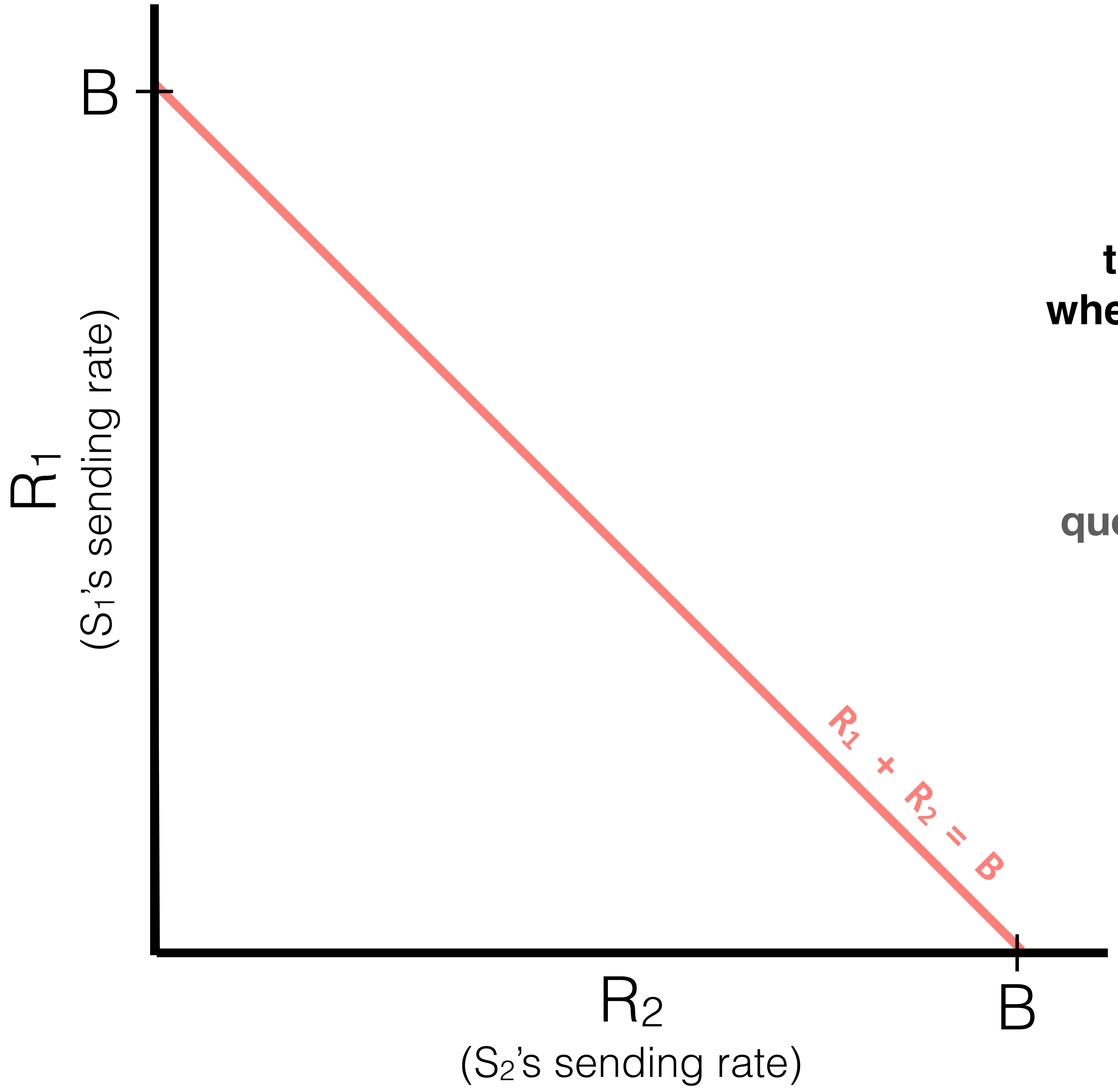
fairness

efficiency: minimize drops, minimize delay, maximize bottleneck utilization

fairness: under infinite offered load, split bandwidth evenly among all sources sharing a bottleneck

AIMD: every RTT, if there is no loss, `W = W + 1`; else, `W = W/2`

congestion control: controlling the source rates to achieve **efficiency** and **fairness**



efficiency

(utilization)

the network is fully utilized when the bottleneck link is "full"

fairness

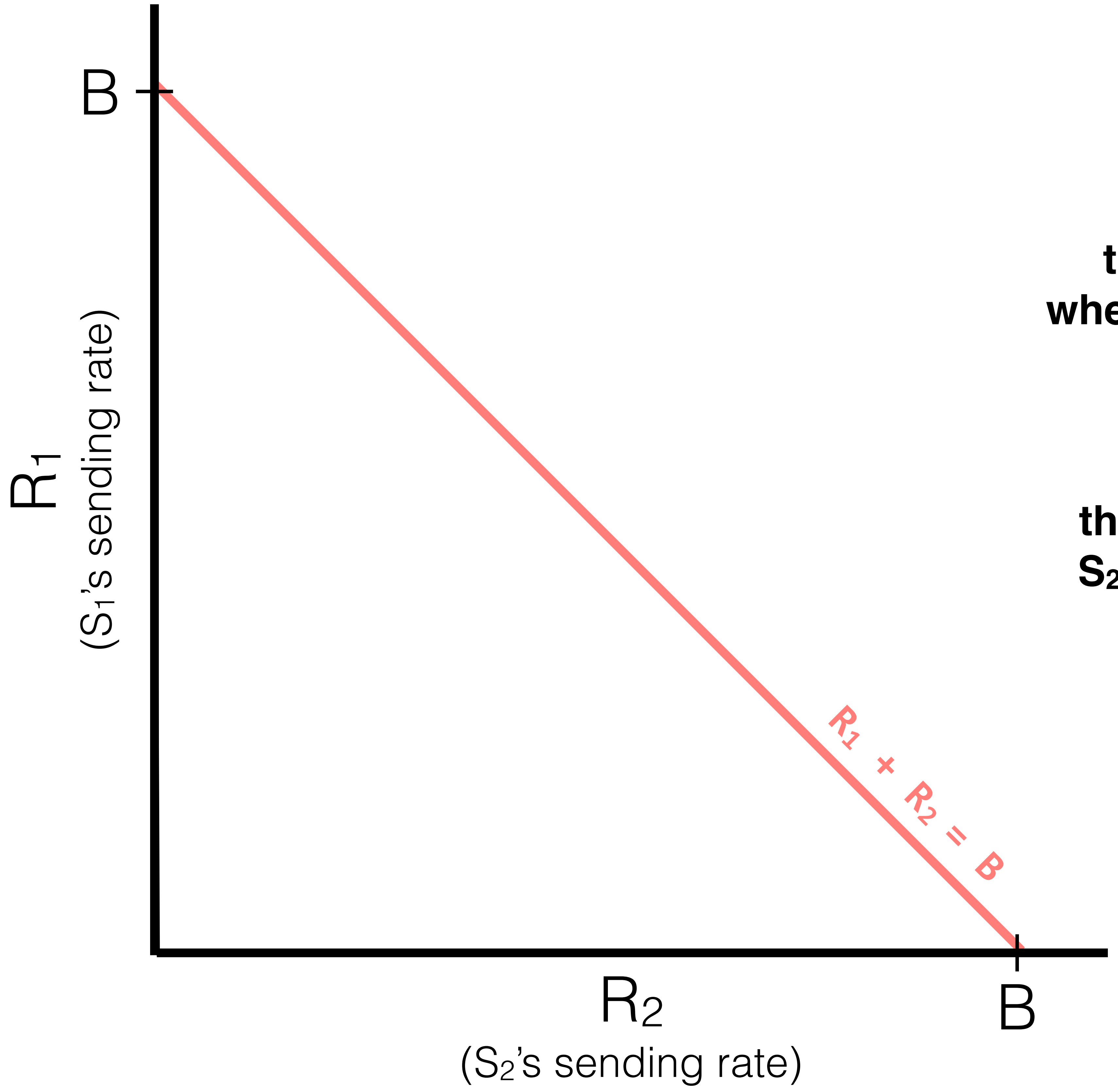
question: what line on this graph would represent fairness?

efficiency: minimize drops, minimize delay, maximize bottleneck utilization

fairness: under infinite offered load, split bandwidth evenly among all sources sharing a bottleneck

AIMD: every RTT, if there is no loss, `W = W + 1`; else, `W = W/2`

congestion control: controlling the source rates to achieve **efficiency** and **fairness**



efficiency

(utilization)

the network is fully utilized when the bottleneck link is "full"

fairness

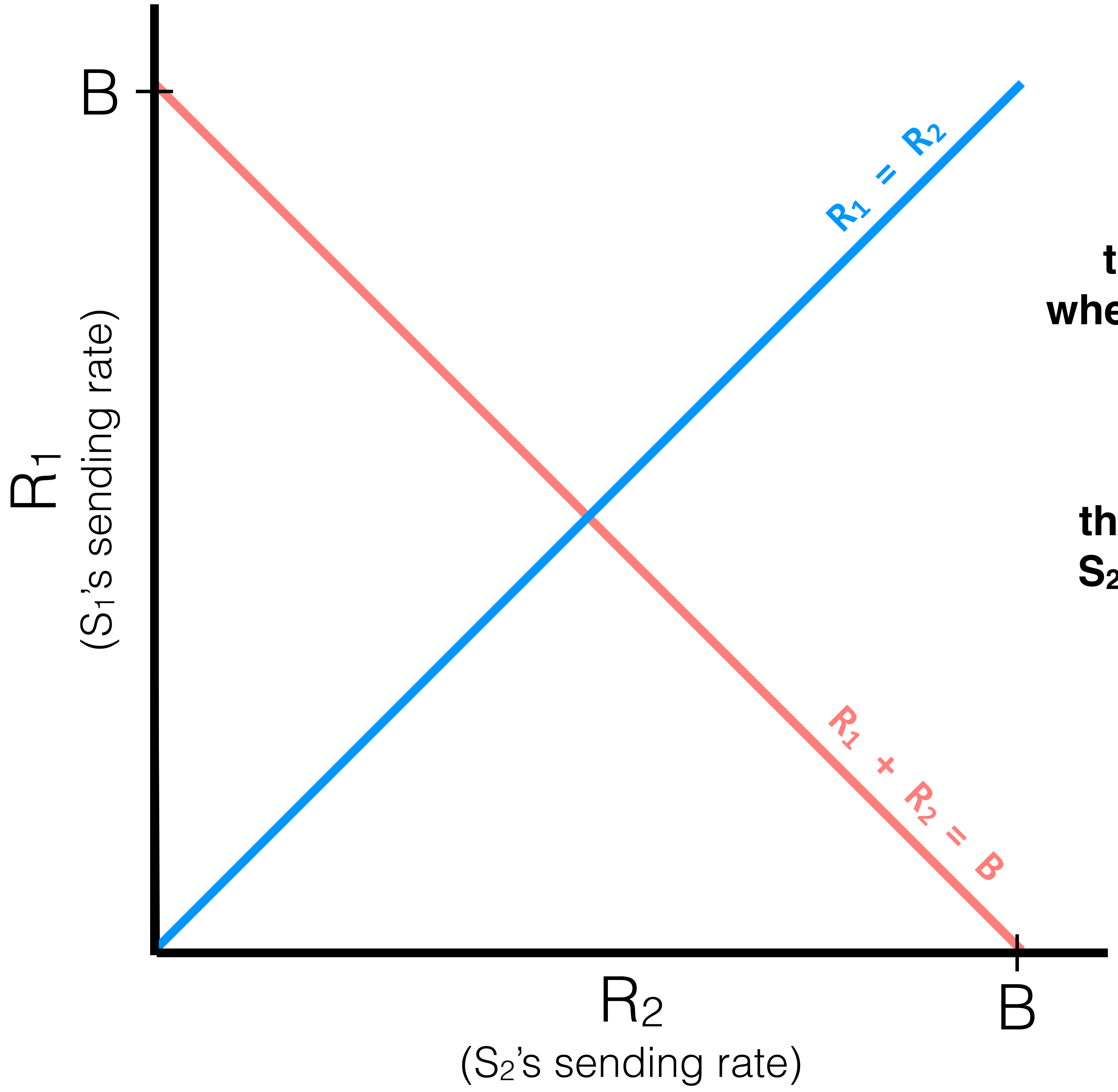
the network is fair when S_1 and S_2 are sending at the same rate

efficiency: minimize drops, minimize delay, maximize bottleneck utilization

fairness: under infinite offered load, split bandwidth evenly among all sources sharing a bottleneck

AIMD: every RTT, if there is no loss, `W = W + 1`; else, `W = W/2`

congestion control: controlling the source rates to achieve **efficiency** and **fairness**



efficiency

(utilization)

the network is fully utilized when the bottleneck link is "full"

fairness

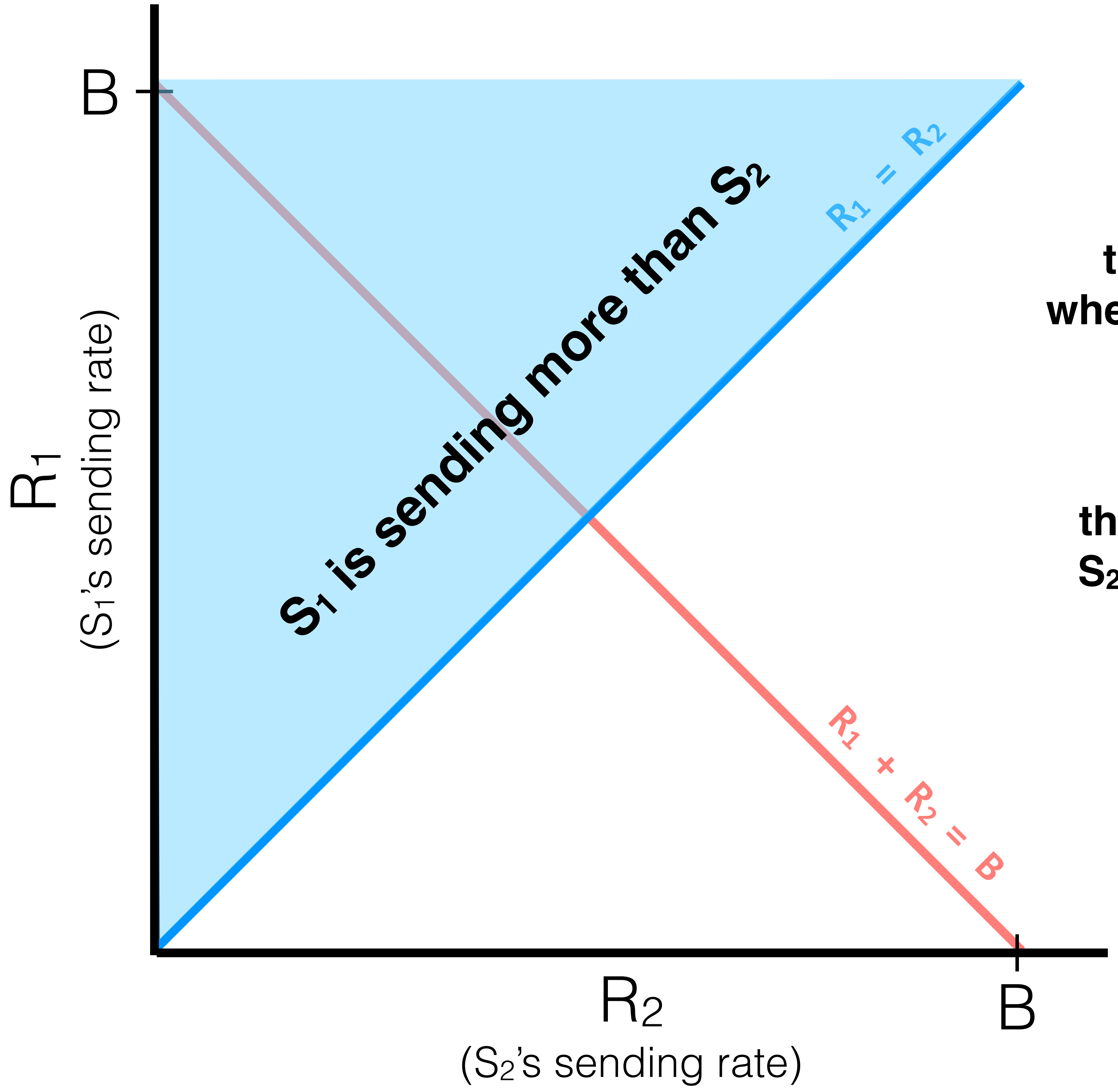
the network is fair when S_1 and S_2 are sending at the same rate

efficiency: minimize drops, minimize delay, maximize bottleneck utilization

fairness: under infinite offered load, split bandwidth evenly among all sources sharing a bottleneck

AIMD: every RTT, if there is no loss, `W = W + 1`; else, `W = W/2`

congestion control: controlling the source rates to achieve **efficiency** and **fairness**



efficiency

(utilization)

the network is fully utilized when the bottleneck link is "full"

fairness

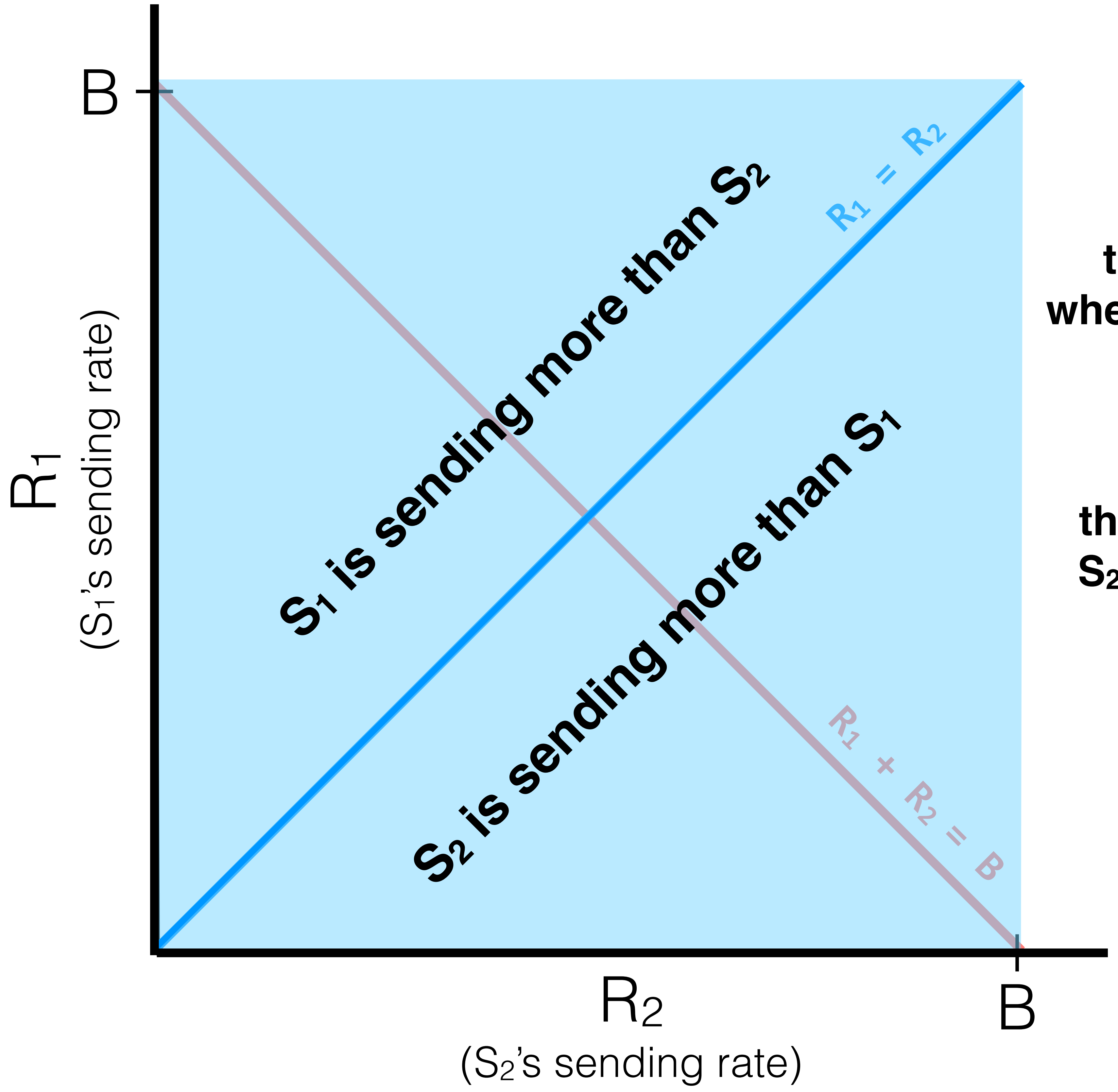
the network is fair when S_1 and S_2 are sending at the same rate

efficiency: minimize drops, minimize delay, maximize bottleneck utilization

fairness: under infinite offered load, split bandwidth evenly among all sources sharing a bottleneck

AIMD: every RTT, if there is no loss, `W = W + 1`; else, `W = W/2`

congestion control: controlling the source rates to achieve **efficiency** and **fairness**



efficiency

(utilization)

the network is fully utilized when the bottleneck link is "full"

fairness

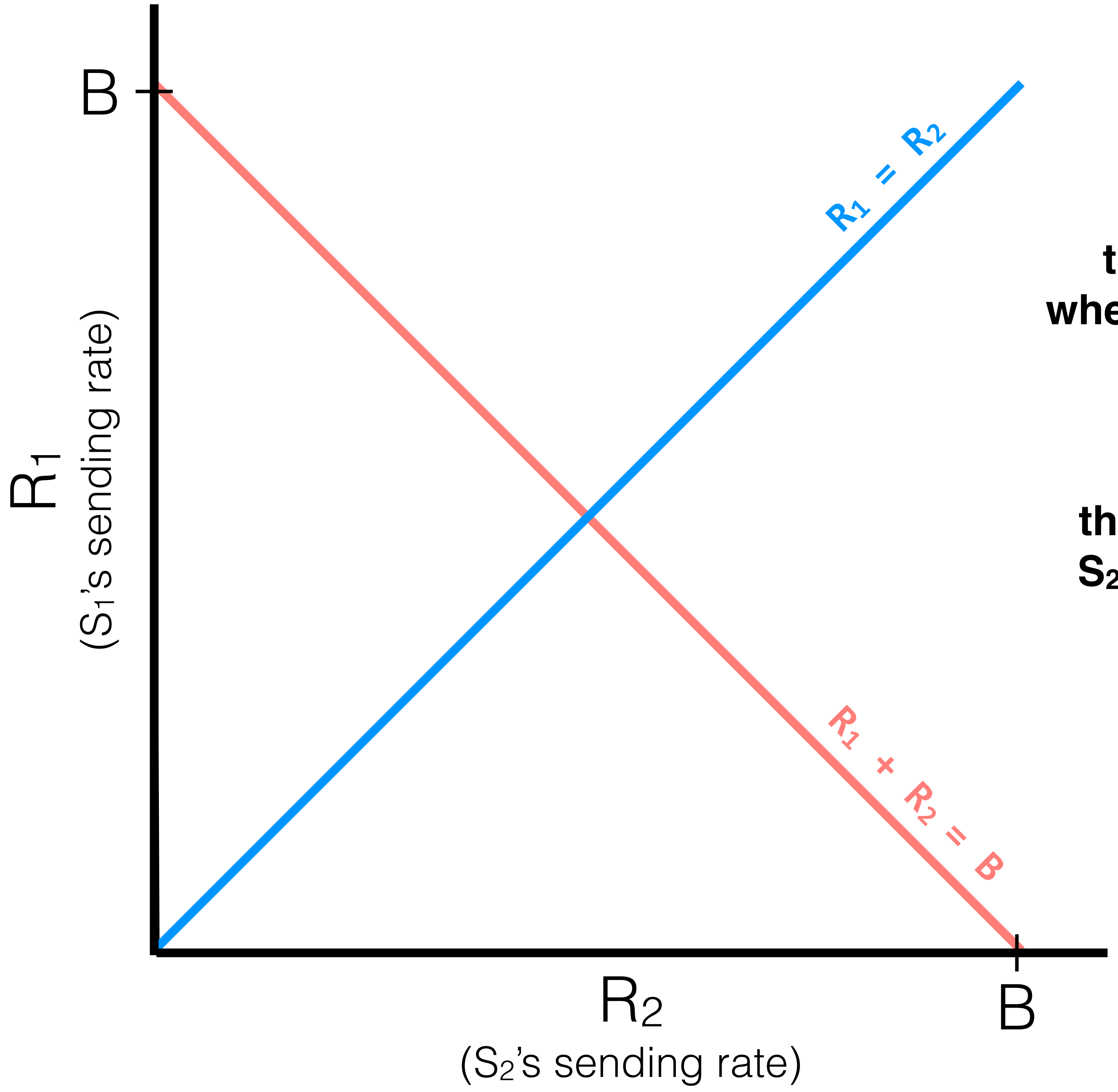
the network is fair when S_1 and S_2 are sending at the same rate

efficiency: minimize drops, minimize delay, maximize bottleneck utilization

fairness: under infinite offered load, split bandwidth evenly among all sources sharing a bottleneck

AIMD: every RTT, if there is no loss, `W = W + 1`; else, `W = W/2`

congestion control: controlling the source rates to achieve **efficiency** and **fairness**



efficiency

(utilization)

the network is fully utilized when the bottleneck link is "full"

fairness

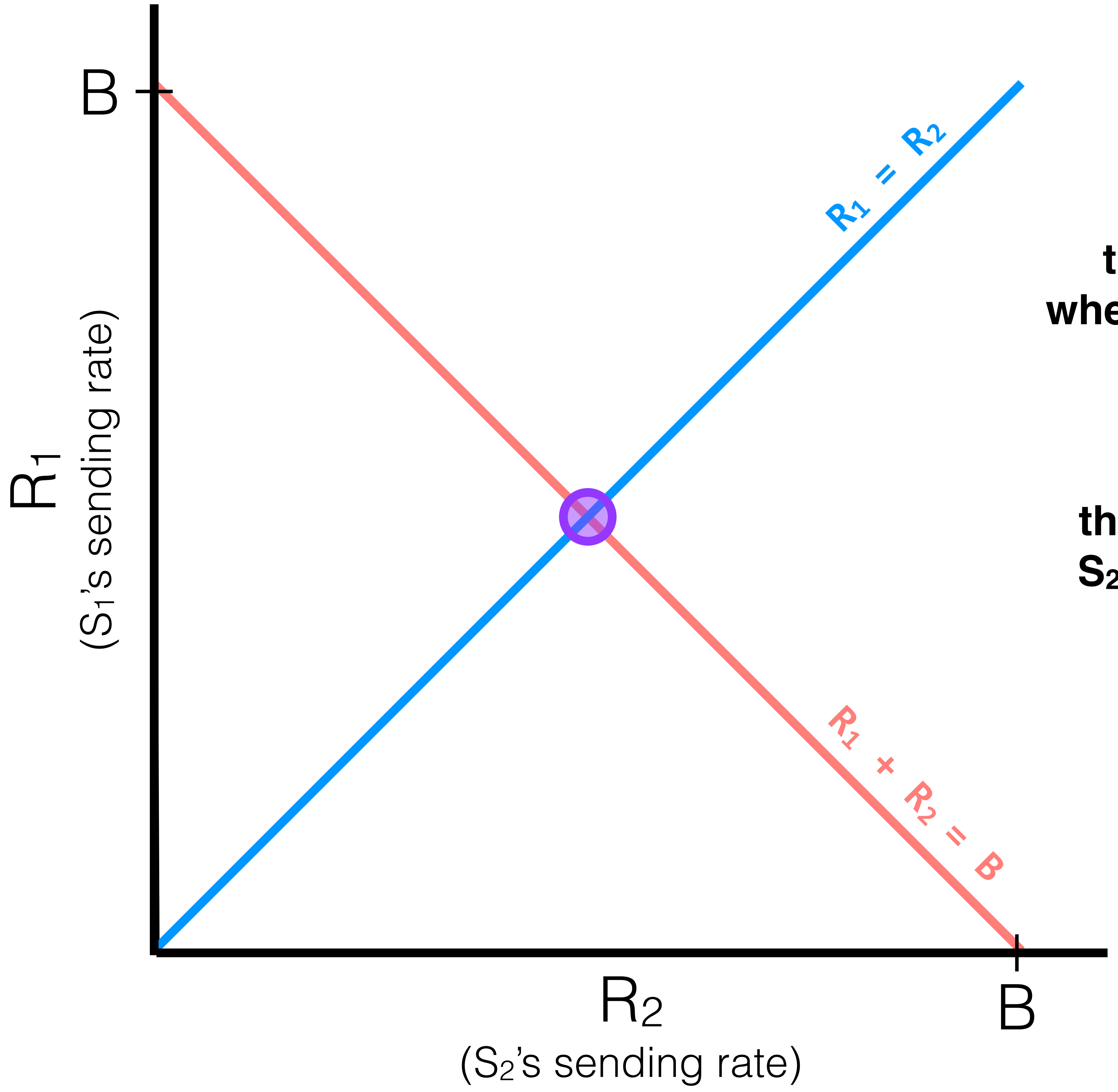
the network is fair when S_1 and S_2 are sending at the same rate

efficiency: minimize drops, minimize delay, maximize bottleneck utilization

fairness: under infinite offered load, split bandwidth evenly among all sources sharing a bottleneck

AIMD: every RTT, if there is no loss, `W = W + 1`; else, `W = W/2`

congestion control: controlling the source rates to achieve **efficiency** and **fairness**



efficiency

(utilization)

the network is fully utilized when the bottleneck link is "full"

fairness

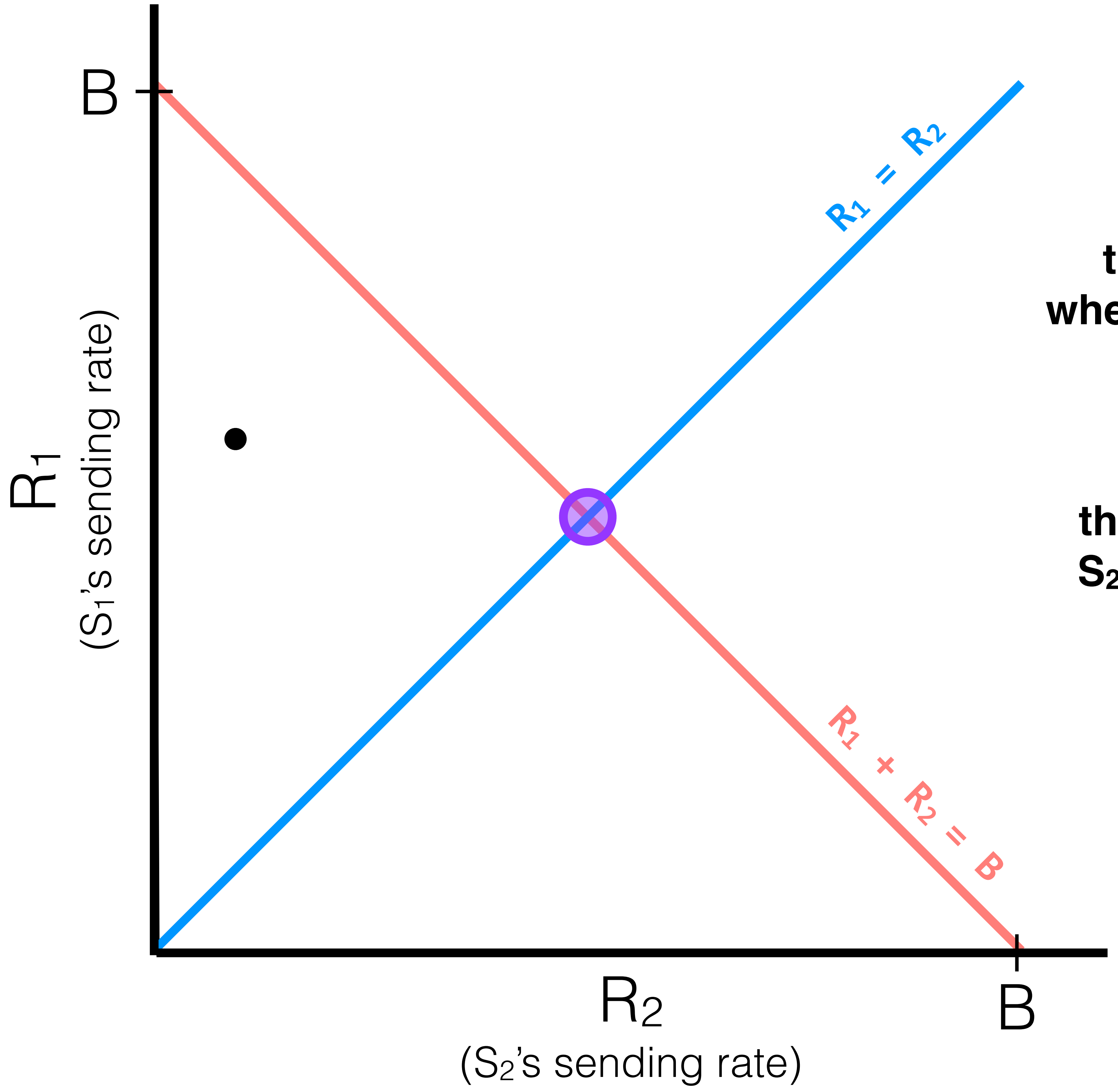
the network is fair when S_1 and S_2 are sending at the same rate

efficiency: minimize drops, minimize delay, maximize bottleneck utilization

fairness: under infinite offered load, split bandwidth evenly among all sources sharing a bottleneck

AIMD: every RTT, if there is no loss, `W = W + 1`; else, `W = W/2`

congestion control: controlling the source rates to achieve **efficiency** and **fairness**



efficiency

(utilization)

the network is fully utilized when the bottleneck link is "full"

fairness

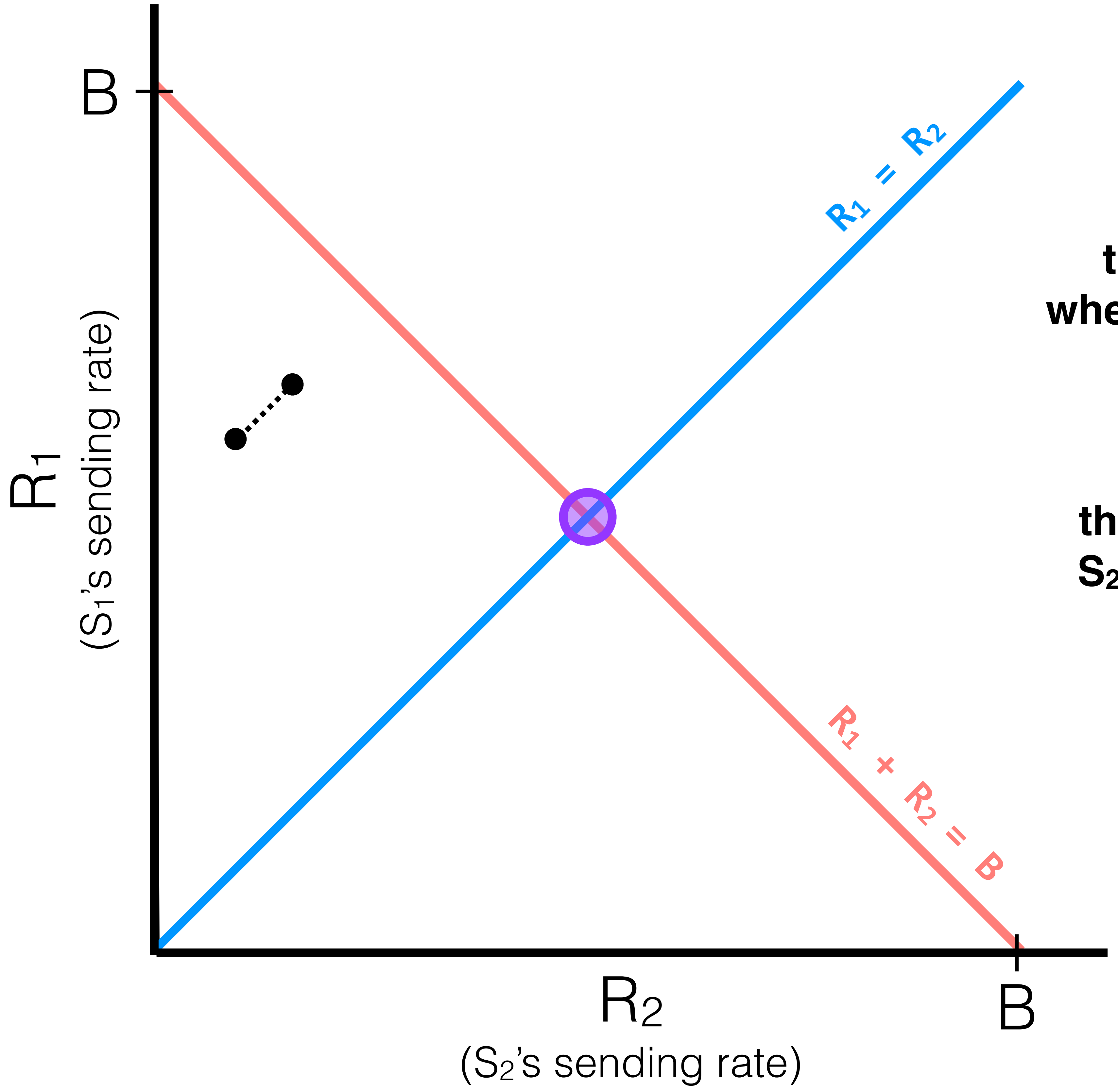
the network is fair when S_1 and S_2 are sending at the same rate

efficiency: minimize drops, minimize delay, maximize bottleneck utilization

fairness: under infinite offered load, split bandwidth evenly among all sources sharing a bottleneck

AIMD: every RTT, if there is no loss, `W = W + 1`; else, `W = W/2`

congestion control: controlling the source rates to achieve **efficiency** and **fairness**



efficiency

(utilization)

the network is fully utilized when the bottleneck link is "full"

fairness

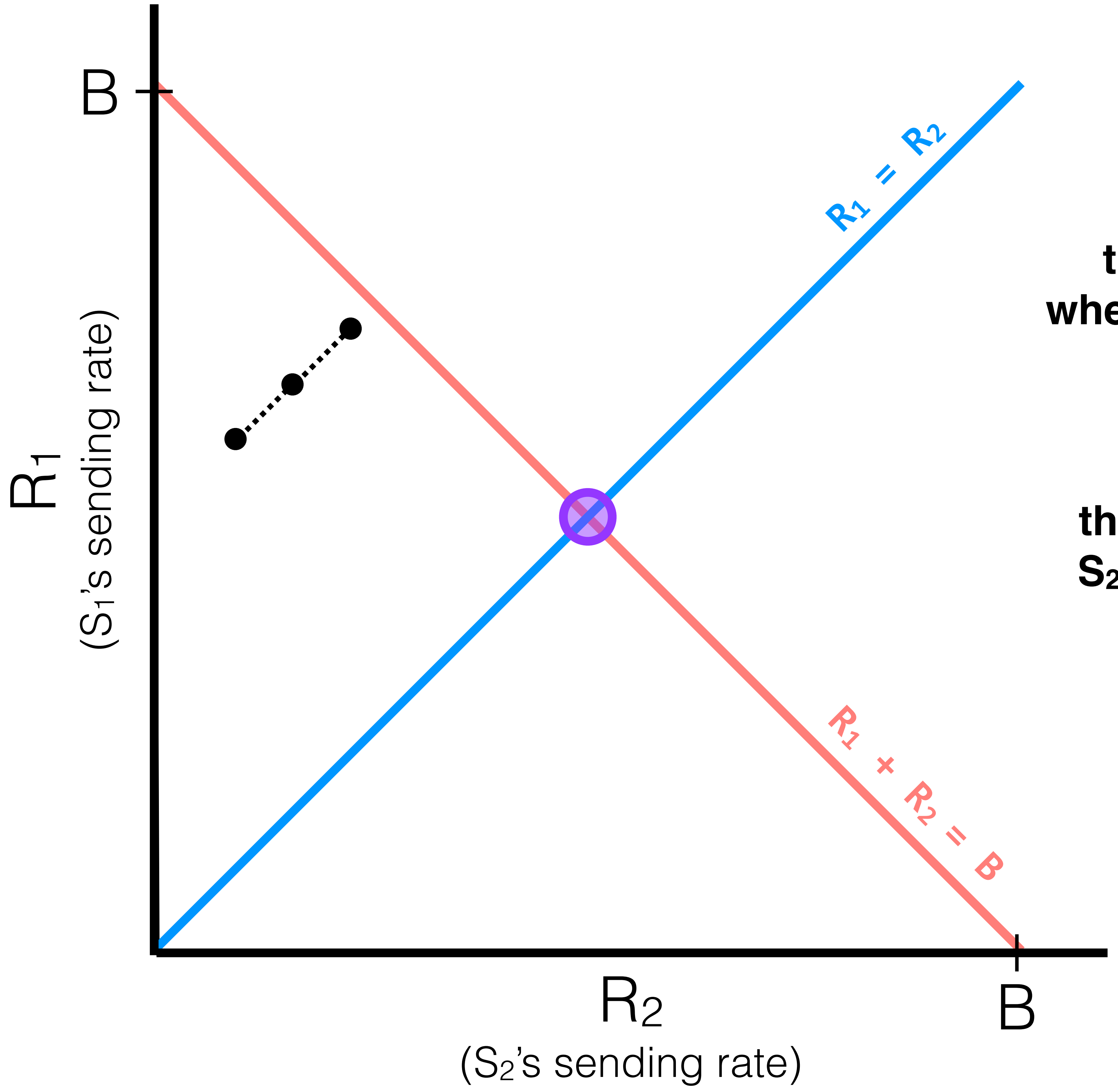
the network is fair when S_1 and S_2 are sending at the same rate

efficiency: minimize drops, minimize delay, maximize bottleneck utilization

fairness: under infinite offered load, split bandwidth evenly among all sources sharing a bottleneck

AIMD: every RTT, if there is no loss, `W = W + 1`; else, `W = W/2`

congestion control: controlling the source rates to achieve **efficiency** and **fairness**



efficiency

(utilization)

the network is fully utilized when the bottleneck link is "full"

fairness

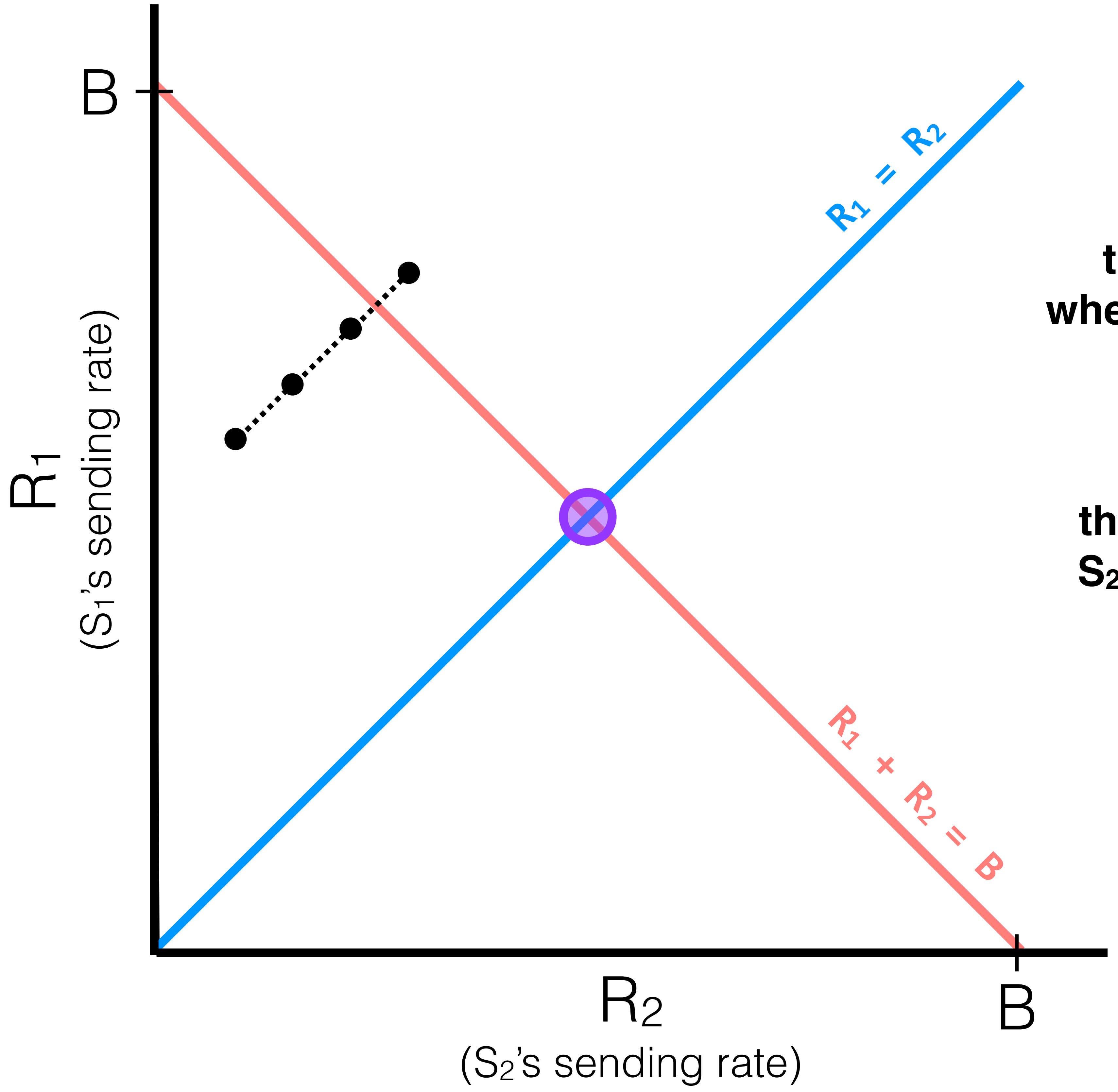
the network is fair when S_1 and S_2 are sending at the same rate

efficiency: minimize drops, minimize delay, maximize bottleneck utilization

fairness: under infinite offered load, split bandwidth evenly among all sources sharing a bottleneck

AIMD: every RTT, if there is no loss, `W = W + 1`; else, `W = W/2`

congestion control: controlling the source rates to achieve **efficiency** and **fairness**



efficiency

(utilization)

the network is fully utilized when the bottleneck link is "full"

fairness

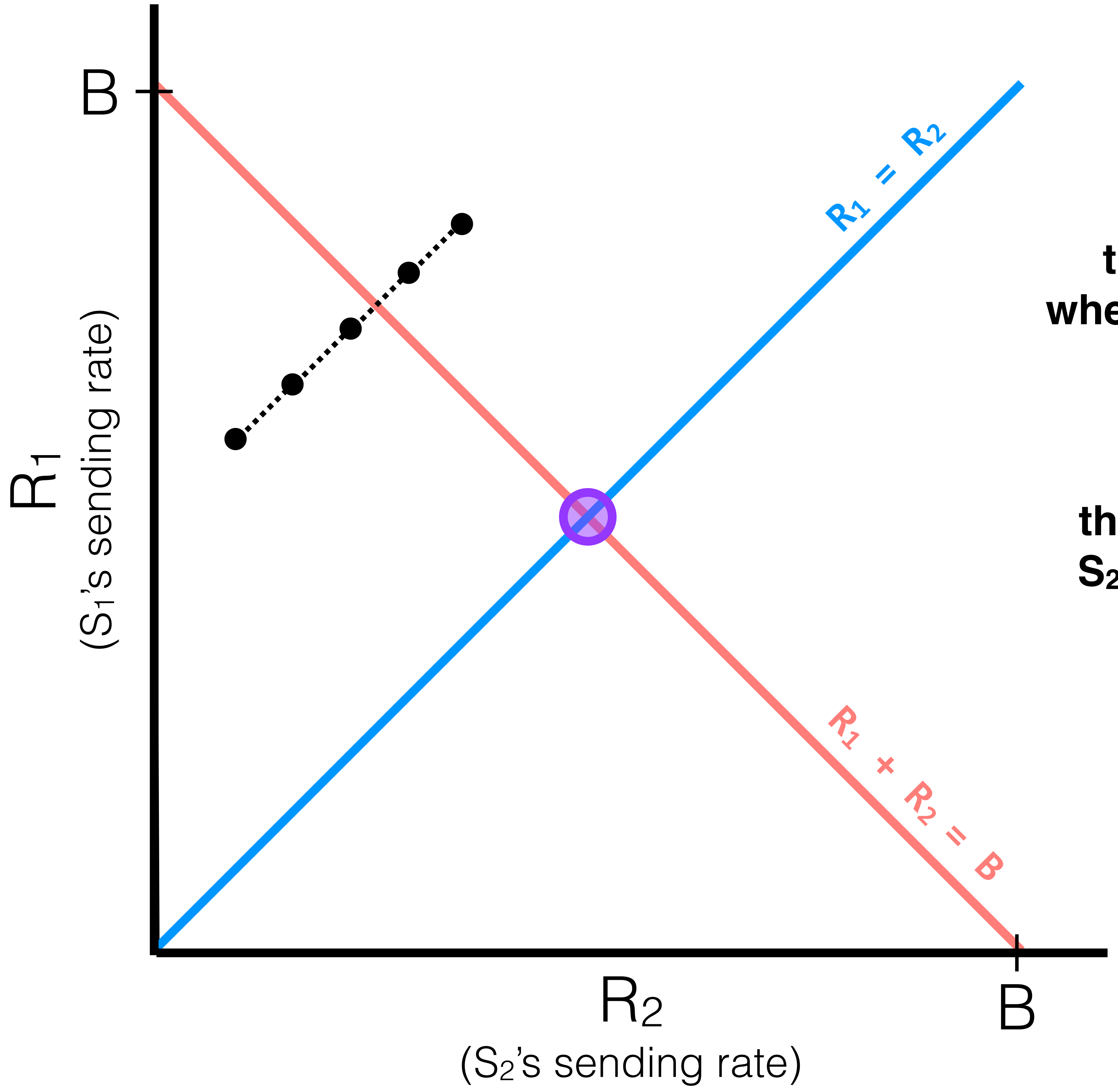
the network is fair when S_1 and S_2 are sending at the same rate

efficiency: minimize drops, minimize delay, maximize bottleneck utilization

fairness: under infinite offered load, split bandwidth evenly among all sources sharing a bottleneck

AIMD: every RTT, if there is no loss, `W = W + 1`; else, `W = W/2`

congestion control: controlling the source rates to achieve **efficiency** and **fairness**



efficiency

(utilization)

the network is fully utilized when the bottleneck link is "full"

fairness

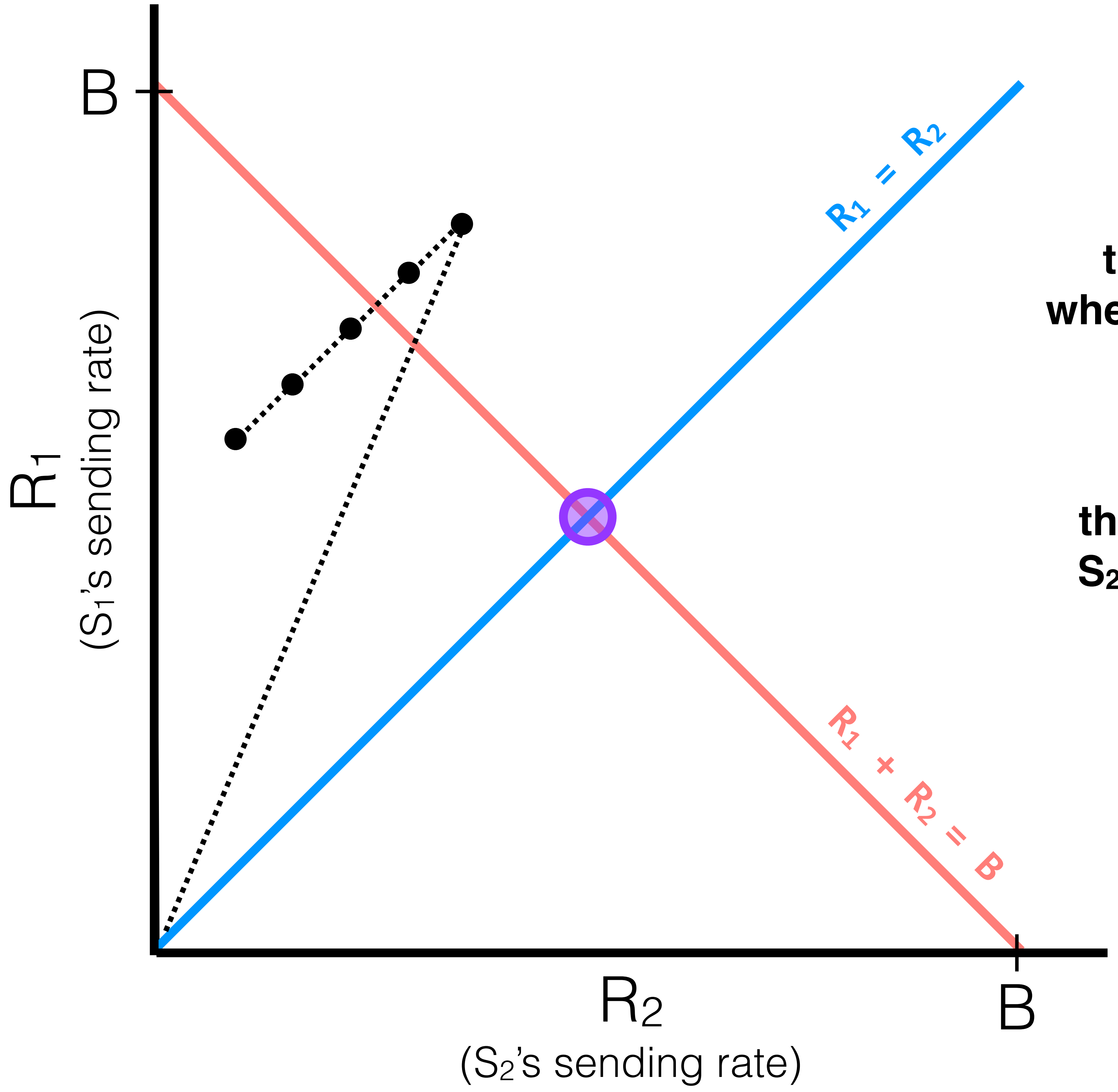
the network is fair when S_1 and S_2 are sending at the same rate

efficiency: minimize drops, minimize delay, maximize bottleneck utilization

fairness: under infinite offered load, split bandwidth evenly among all sources sharing a bottleneck

AIMD: every RTT, if there is no loss, `W = W + 1`; else, `W = W/2`

congestion control: controlling the source rates to achieve **efficiency** and **fairness**



efficiency

(utilization)

the network is fully utilized when the bottleneck link is "full"

fairness

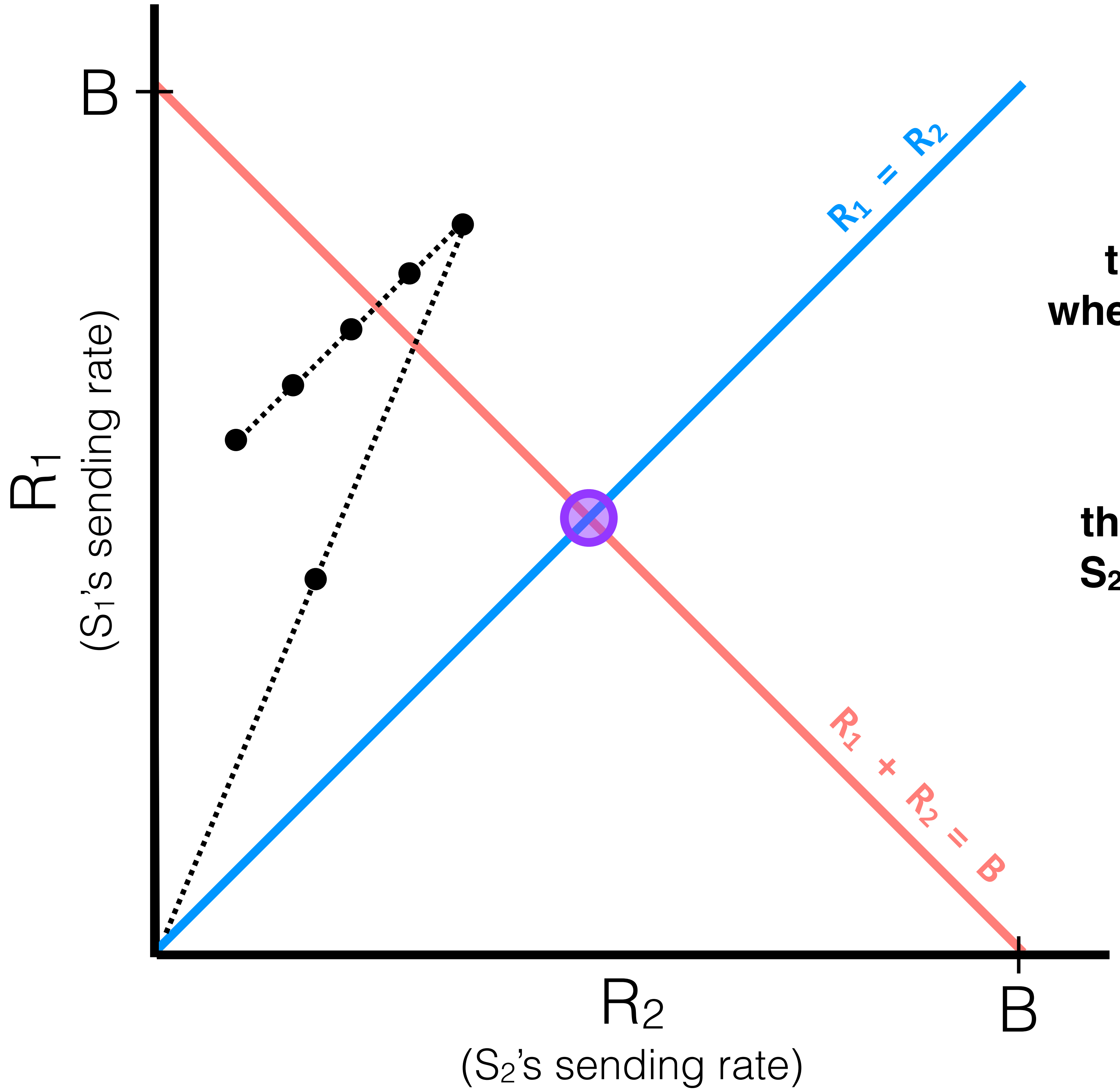
the network is fair when S_1 and S_2 are sending at the same rate

efficiency: minimize drops, minimize delay, maximize bottleneck utilization

fairness: under infinite offered load, split bandwidth evenly among all sources sharing a bottleneck

AIMD: every RTT, if there is no loss, `W = W + 1`; else, `W = W/2`

congestion control: controlling the source rates to achieve **efficiency** and **fairness**



efficiency

(utilization)

the network is fully utilized when the bottleneck link is "full"

fairness

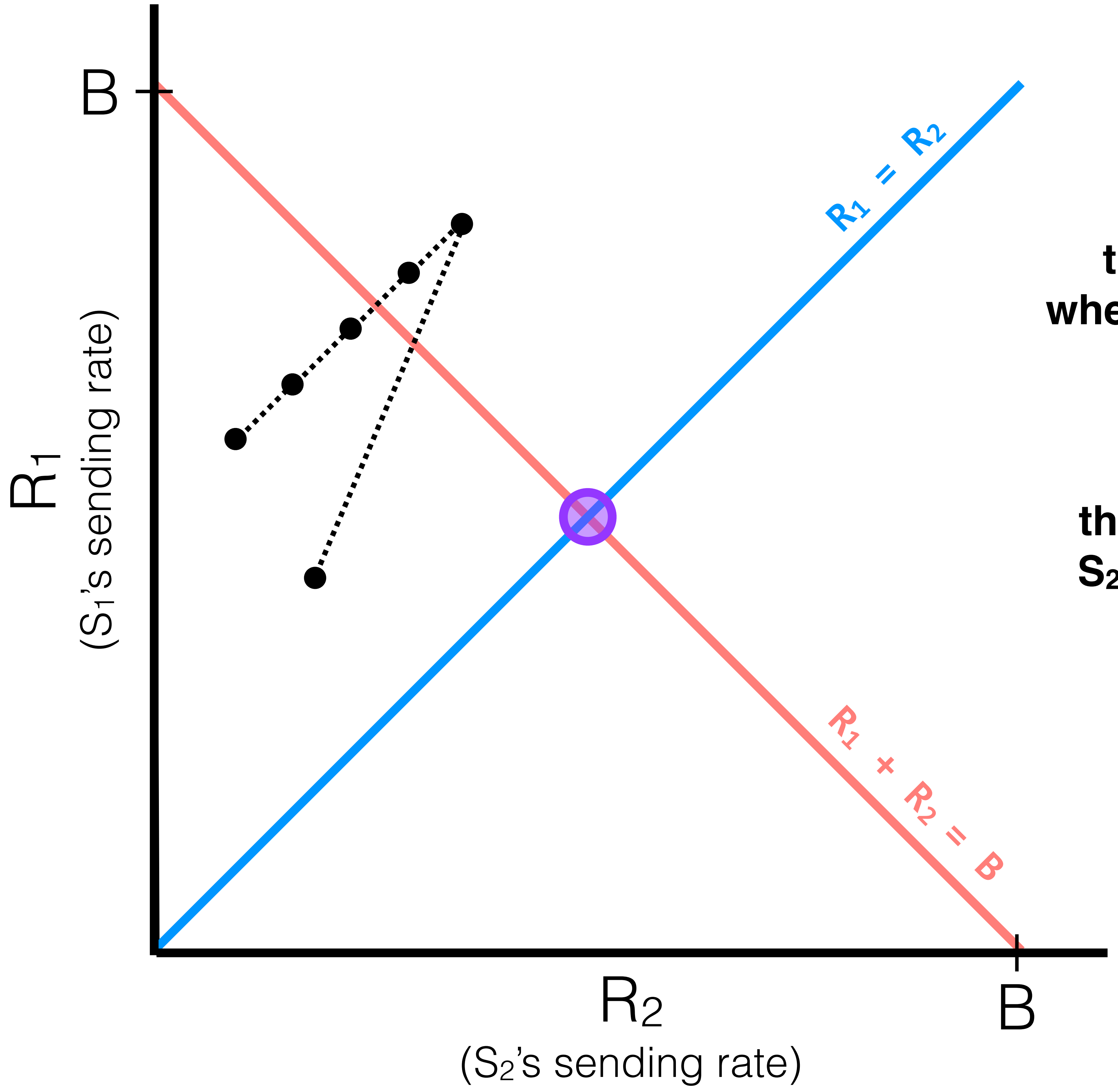
the network is fair when S_1 and S_2 are sending at the same rate

efficiency: minimize drops, minimize delay, maximize bottleneck utilization

fairness: under infinite offered load, split bandwidth evenly among all sources sharing a bottleneck

AIMD: every RTT, if there is no loss, `W = W + 1`; else, `W = W/2`

congestion control: controlling the source rates to achieve **efficiency** and **fairness**



efficiency

(utilization)

the network is fully utilized when the bottleneck link is "full"

fairness

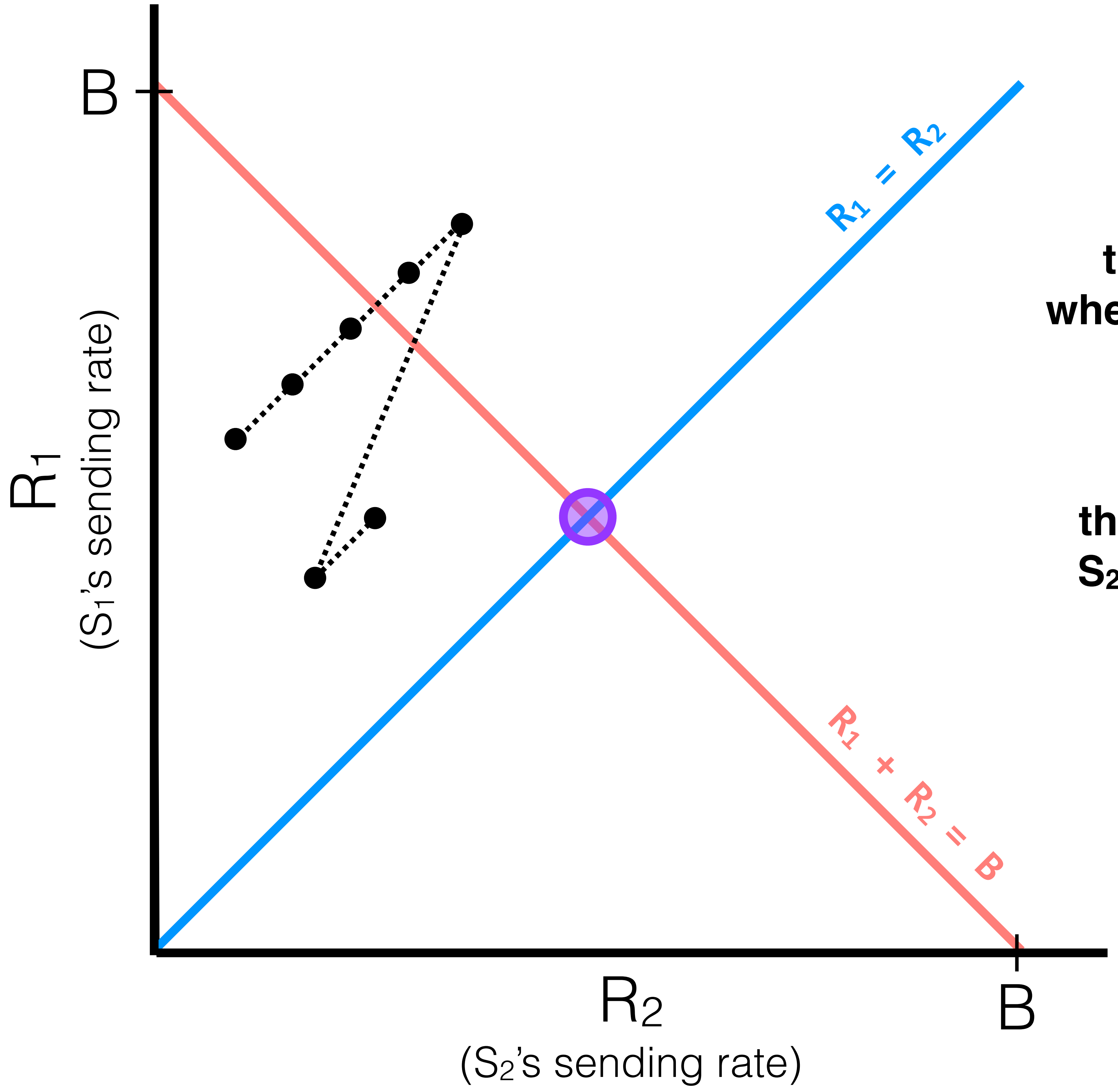
the network is fair when S_1 and S_2 are sending at the same rate

efficiency: minimize drops, minimize delay, maximize bottleneck utilization

fairness: under infinite offered load, split bandwidth evenly among all sources sharing a bottleneck

AIMD: every RTT, if there is no loss, `W = W + 1`; else, `W = W/2`

congestion control: controlling the source rates to achieve **efficiency** and **fairness**



efficiency

(utilization)

the network is fully utilized when the bottleneck link is "full"

fairness

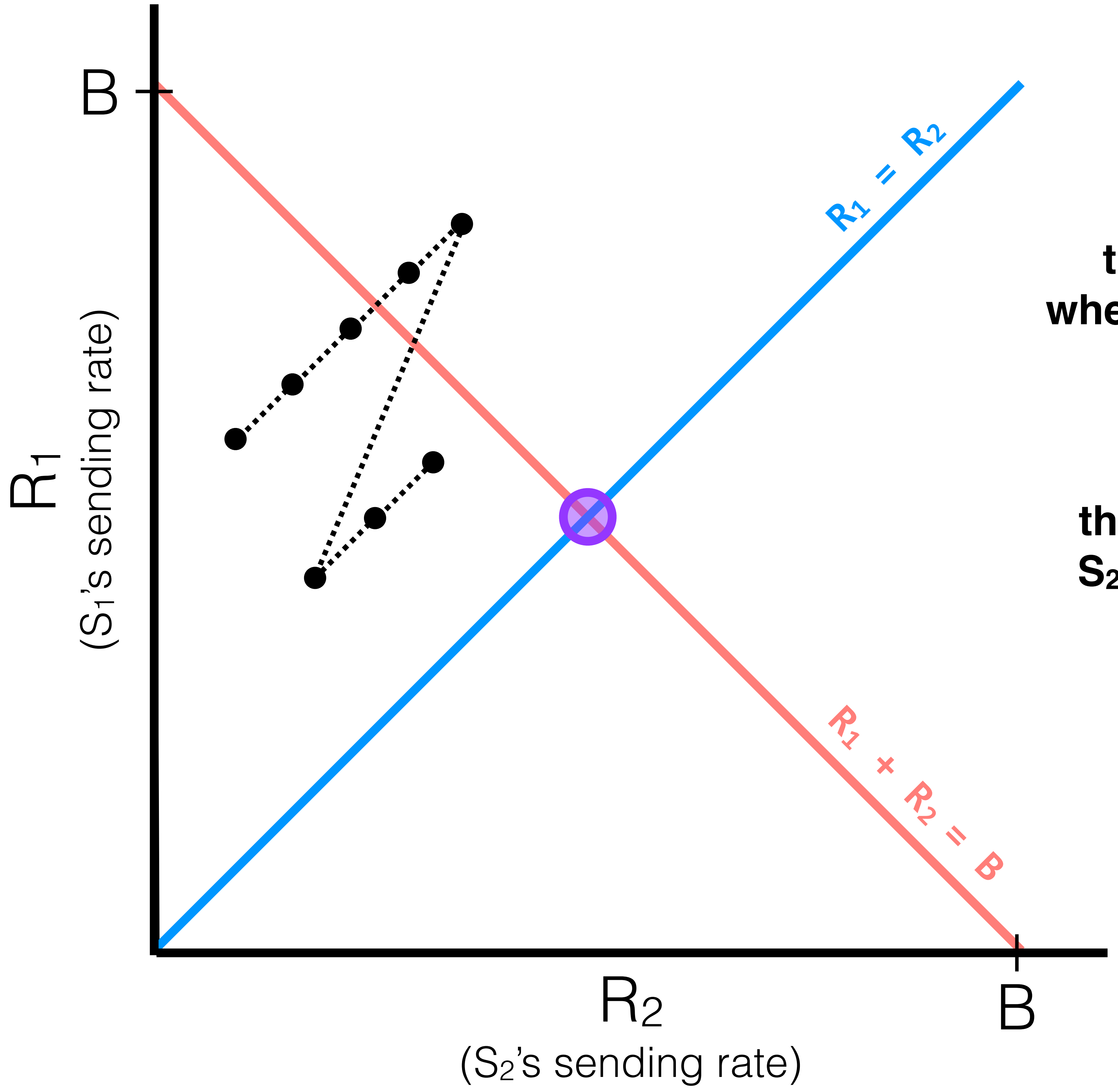
the network is fair when S_1 and S_2 are sending at the same rate

efficiency: minimize drops, minimize delay, maximize bottleneck utilization

fairness: under infinite offered load, split bandwidth evenly among all sources sharing a bottleneck

AIMD: every RTT, if there is no loss, `W = W + 1`; else, `W = W/2`

congestion control: controlling the source rates to achieve **efficiency** and **fairness**



efficiency

(utilization)

the network is fully utilized when the bottleneck link is "full"

fairness

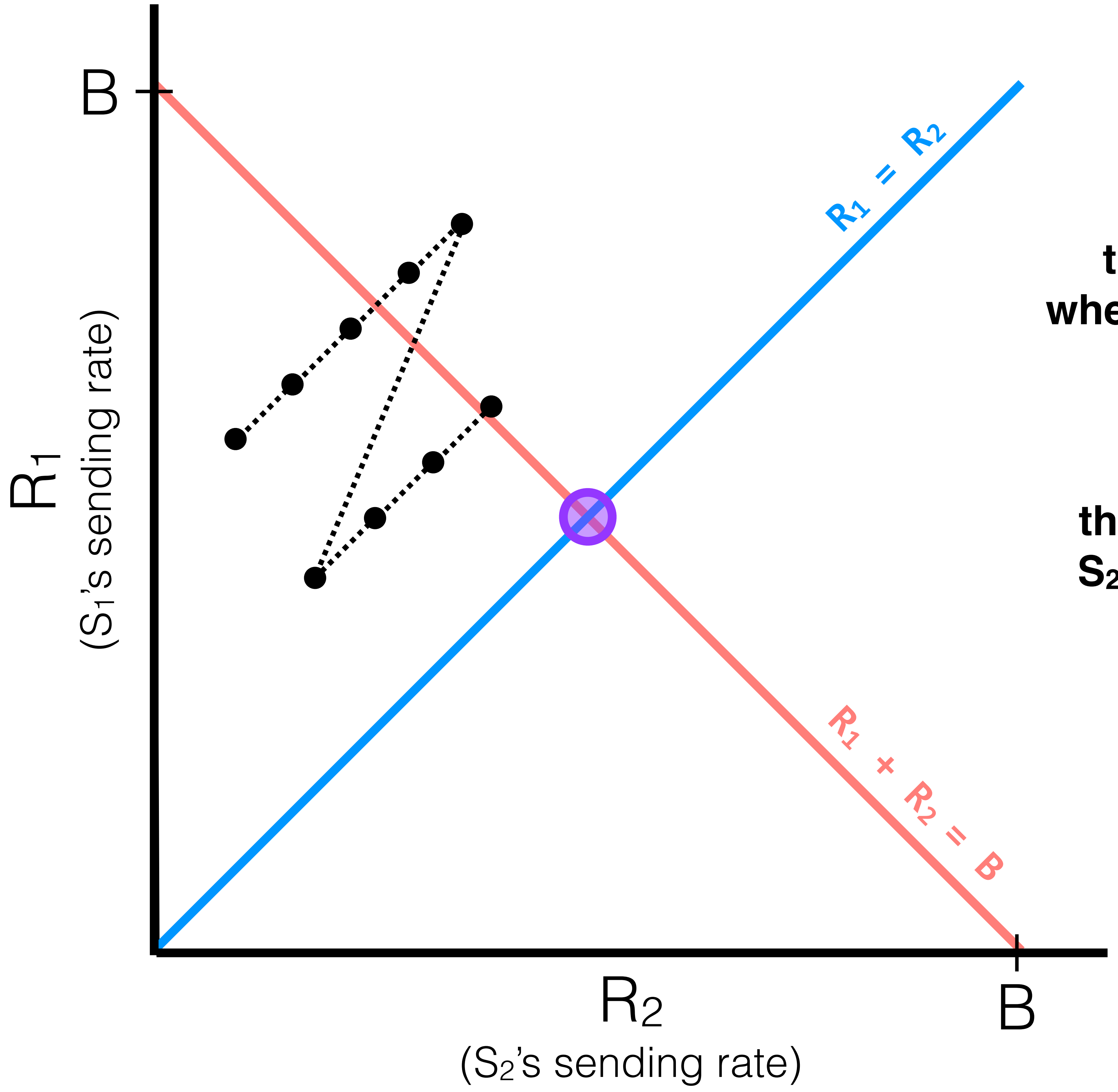
the network is fair when S_1 and S_2 are sending at the same rate

efficiency: minimize drops, minimize delay, maximize bottleneck utilization

fairness: under infinite offered load, split bandwidth evenly among all sources sharing a bottleneck

AIMD: every RTT, if there is no loss, `W = W + 1`; else, `W = W/2`

congestion control: controlling the source rates to achieve **efficiency** and **fairness**



efficiency

(utilization)

the network is fully utilized when the bottleneck link is "full"

fairness

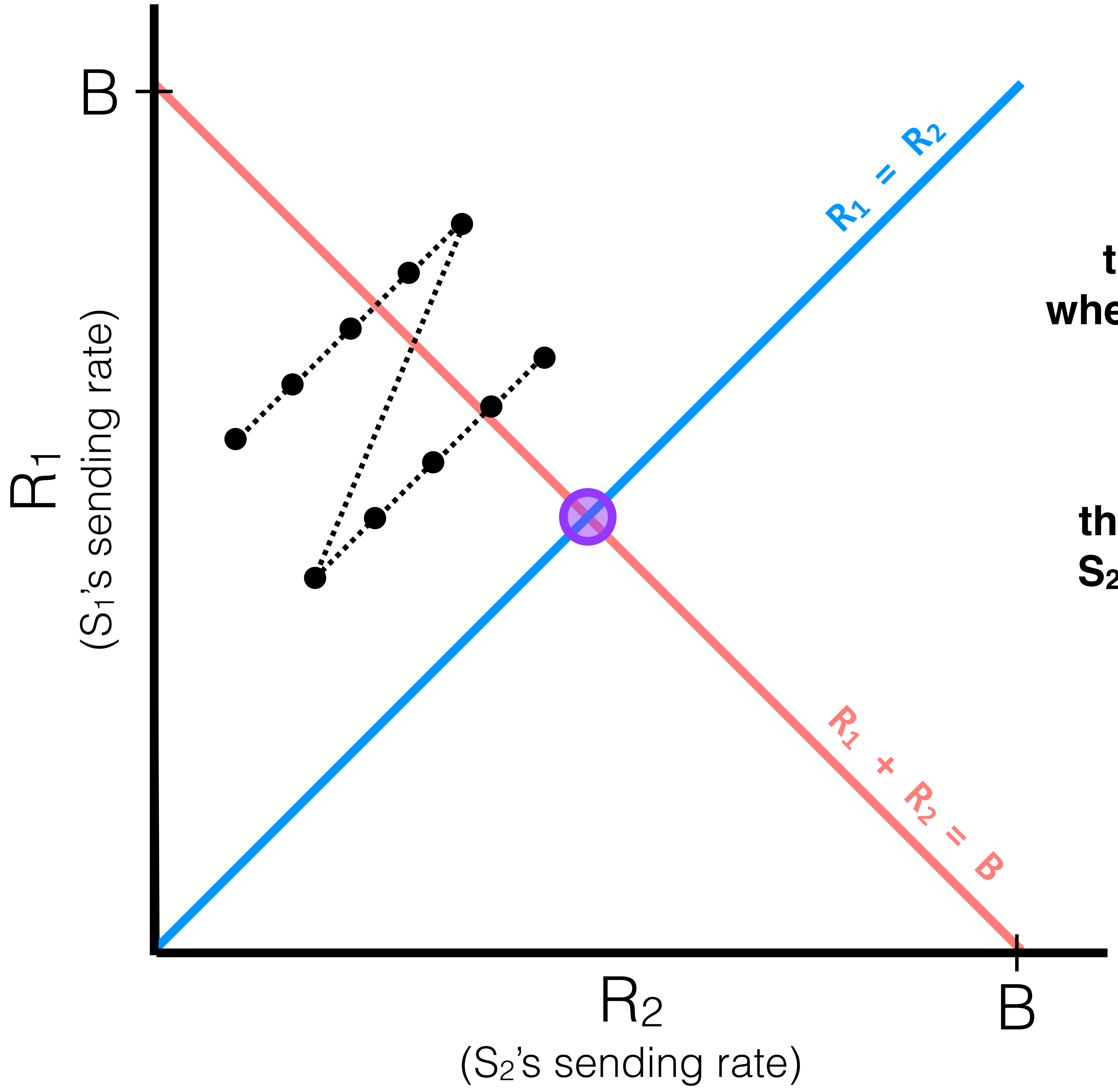
the network is fair when S_1 and S_2 are sending at the same rate

efficiency: minimize drops, minimize delay, maximize bottleneck utilization

fairness: under infinite offered load, split bandwidth evenly among all sources sharing a bottleneck

AIMD: every RTT, if there is no loss, `W = W + 1`; else, `W = W/2`

congestion control: controlling the source rates to achieve **efficiency** and **fairness**



efficiency

(utilization)

the network is fully utilized when the bottleneck link is "full"

fairness

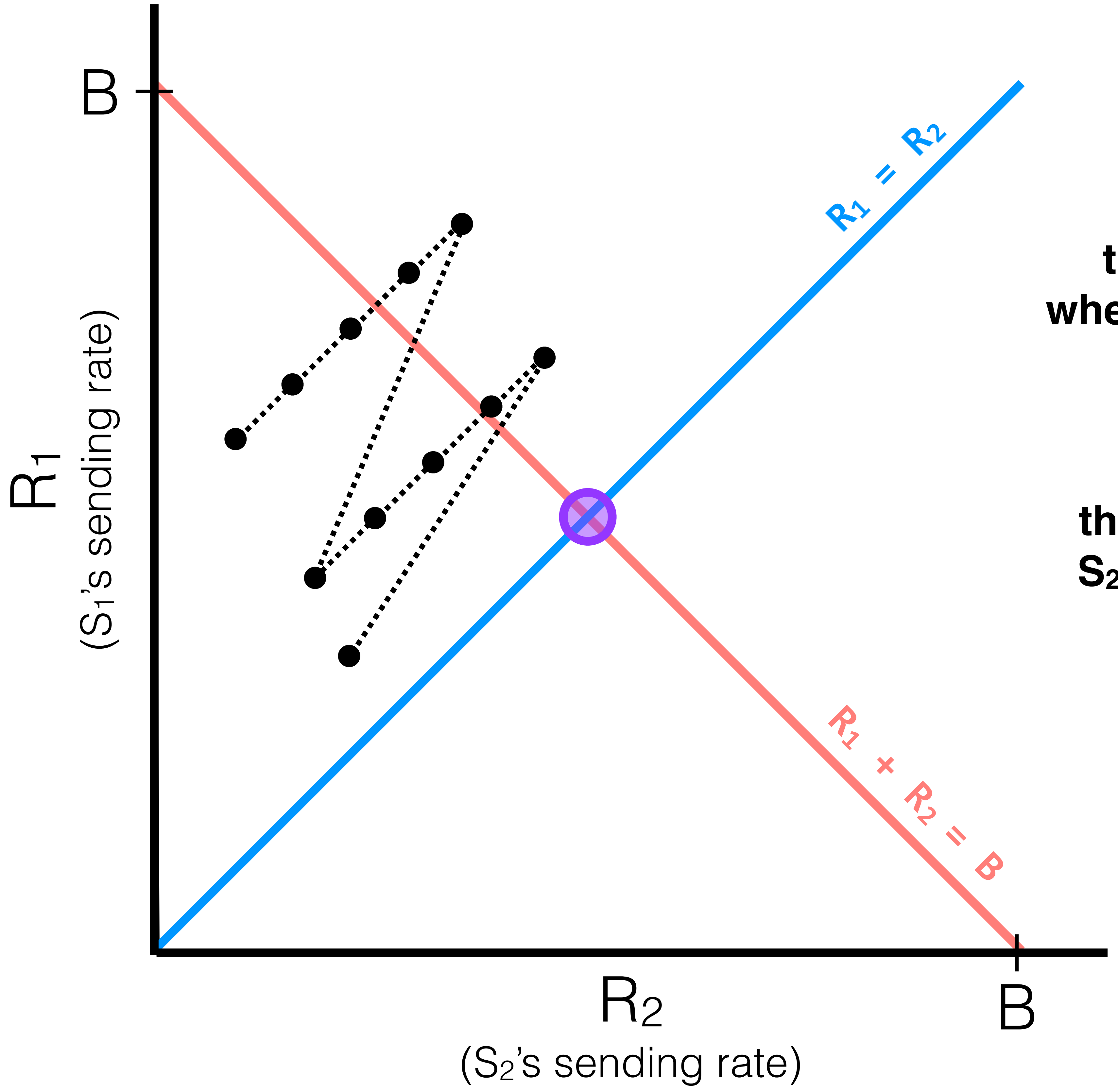
the network is fair when S_1 and S_2 are sending at the same rate

efficiency: minimize drops, minimize delay, maximize bottleneck utilization

fairness: under infinite offered load, split bandwidth evenly among all sources sharing a bottleneck

AIMD: every RTT, if there is no loss, `W = W + 1`; else, `W = W/2`

congestion control: controlling the source rates to achieve **efficiency** and **fairness**



efficiency
(utilization)

the network is fully utilized when the bottleneck link is "full"

fairness

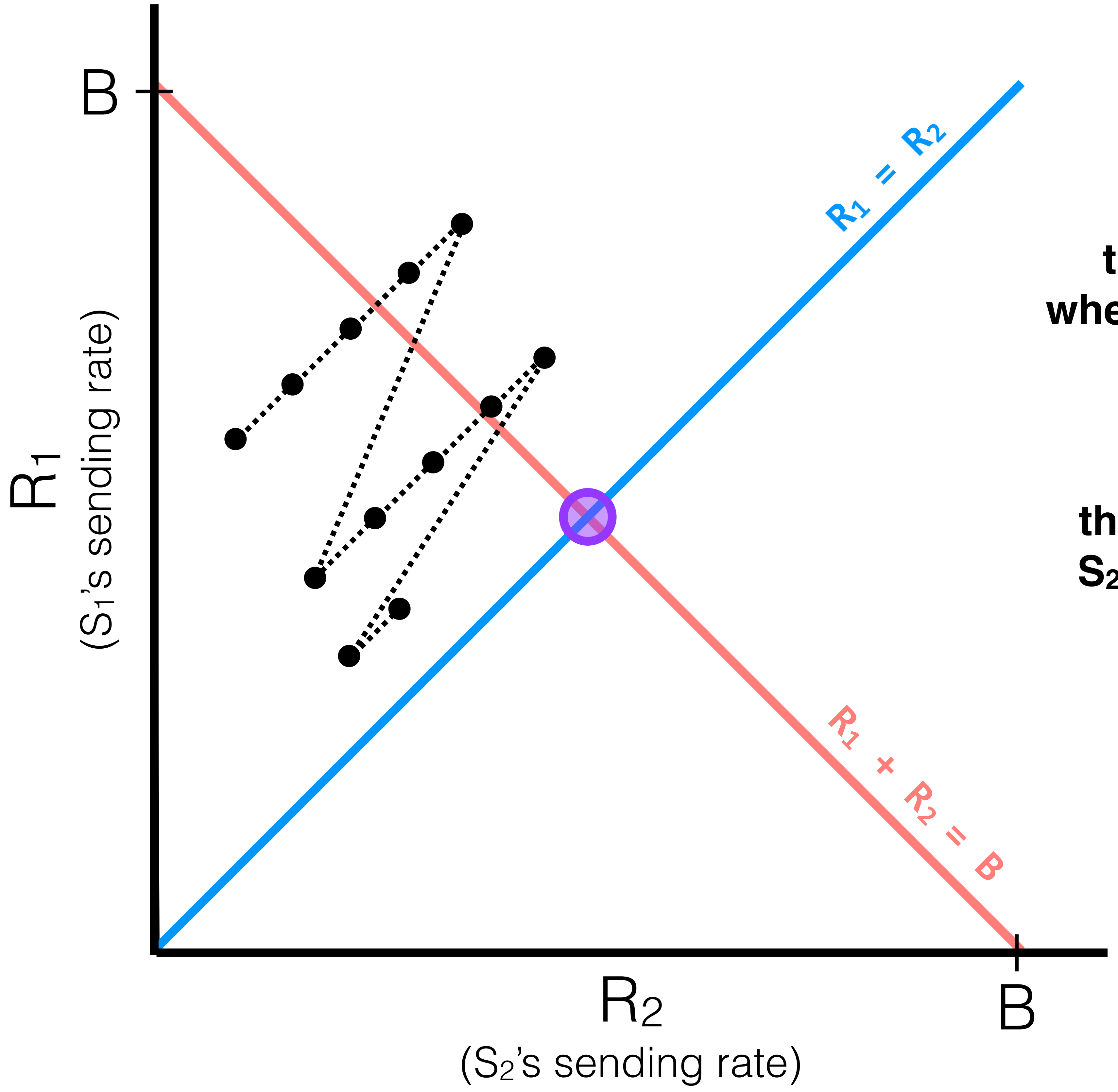
the network is fair when S_1 and S_2 are sending at the same rate

efficiency: minimize drops, minimize delay, maximize bottleneck utilization

fairness: under infinite offered load, split bandwidth evenly among all sources sharing a bottleneck

AIMD: every RTT, if there is no loss, `W = W + 1`; else, `W = W/2`

congestion control: controlling the source rates to achieve **efficiency** and **fairness**



efficiency

(utilization)

the network is fully utilized when the bottleneck link is "full"

fairness

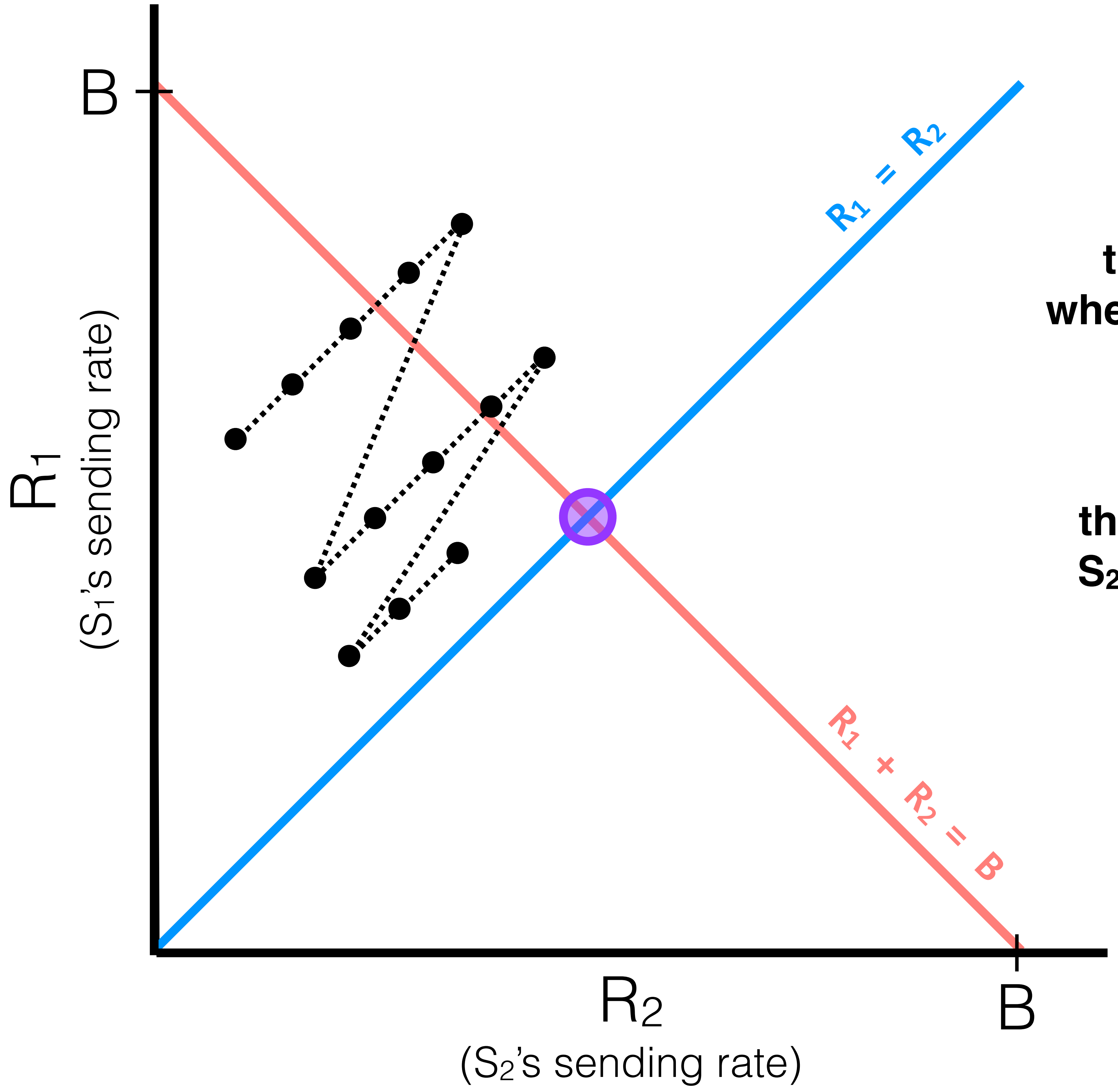
the network is fair when S_1 and S_2 are sending at the same rate

efficiency: minimize drops, minimize delay, maximize bottleneck utilization

fairness: under infinite offered load, split bandwidth evenly among all sources sharing a bottleneck

AIMD: every RTT, if there is no loss, `W = W + 1`; else, `W = W/2`

congestion control: controlling the source rates to achieve **efficiency** and **fairness**



efficiency
(utilization)

the network is fully utilized when the bottleneck link is "full"

fairness

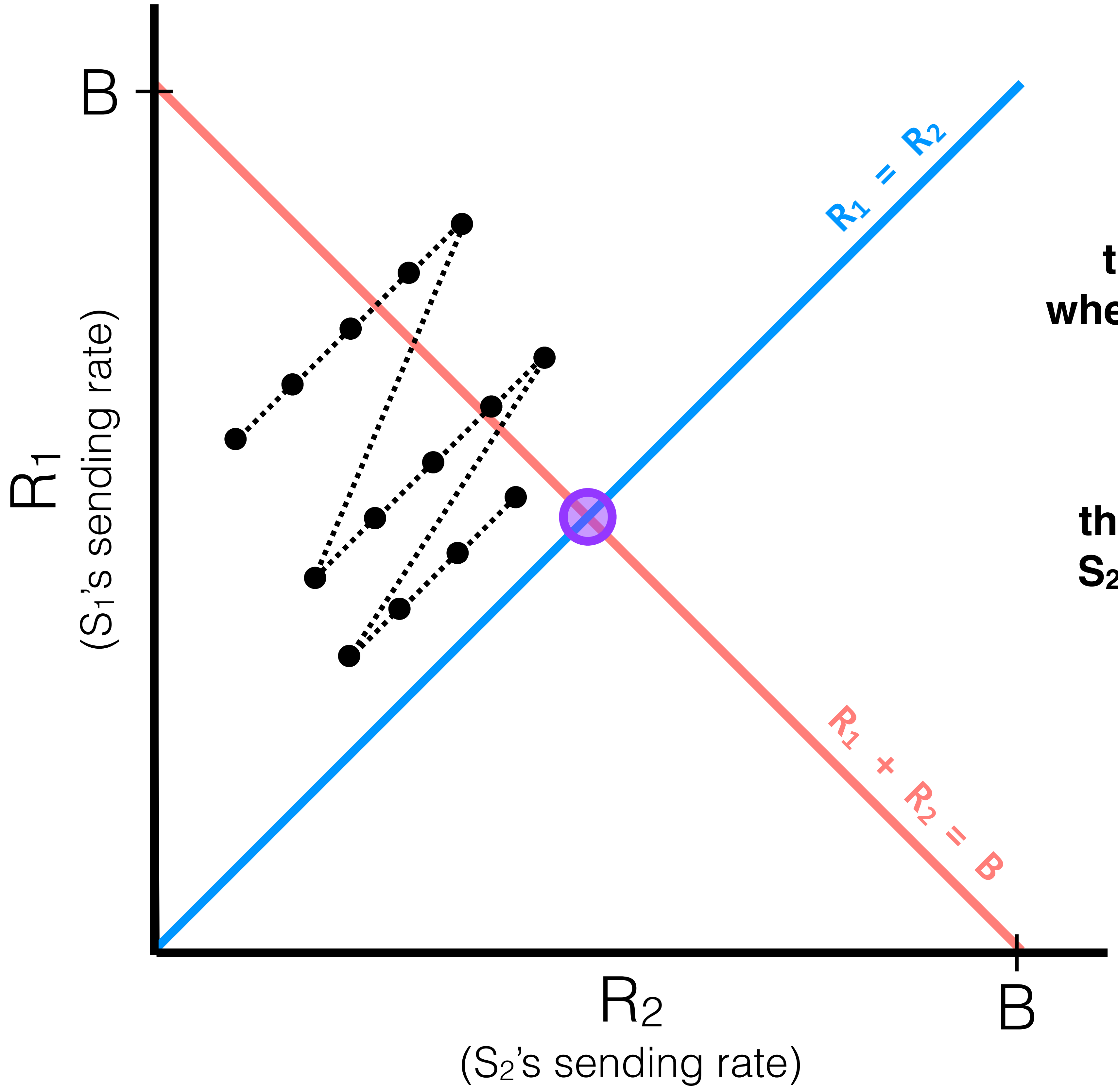
the network is fair when S_1 and S_2 are sending at the same rate

efficiency: minimize drops, minimize delay, maximize bottleneck utilization

fairness: under infinite offered load, split bandwidth evenly among all sources sharing a bottleneck

AIMD: every RTT, if there is no loss, `W = W + 1`; else, `W = W/2`

congestion control: controlling the source rates to achieve **efficiency** and **fairness**



efficiency

(utilization)

the network is fully utilized when the bottleneck link is "full"

fairness

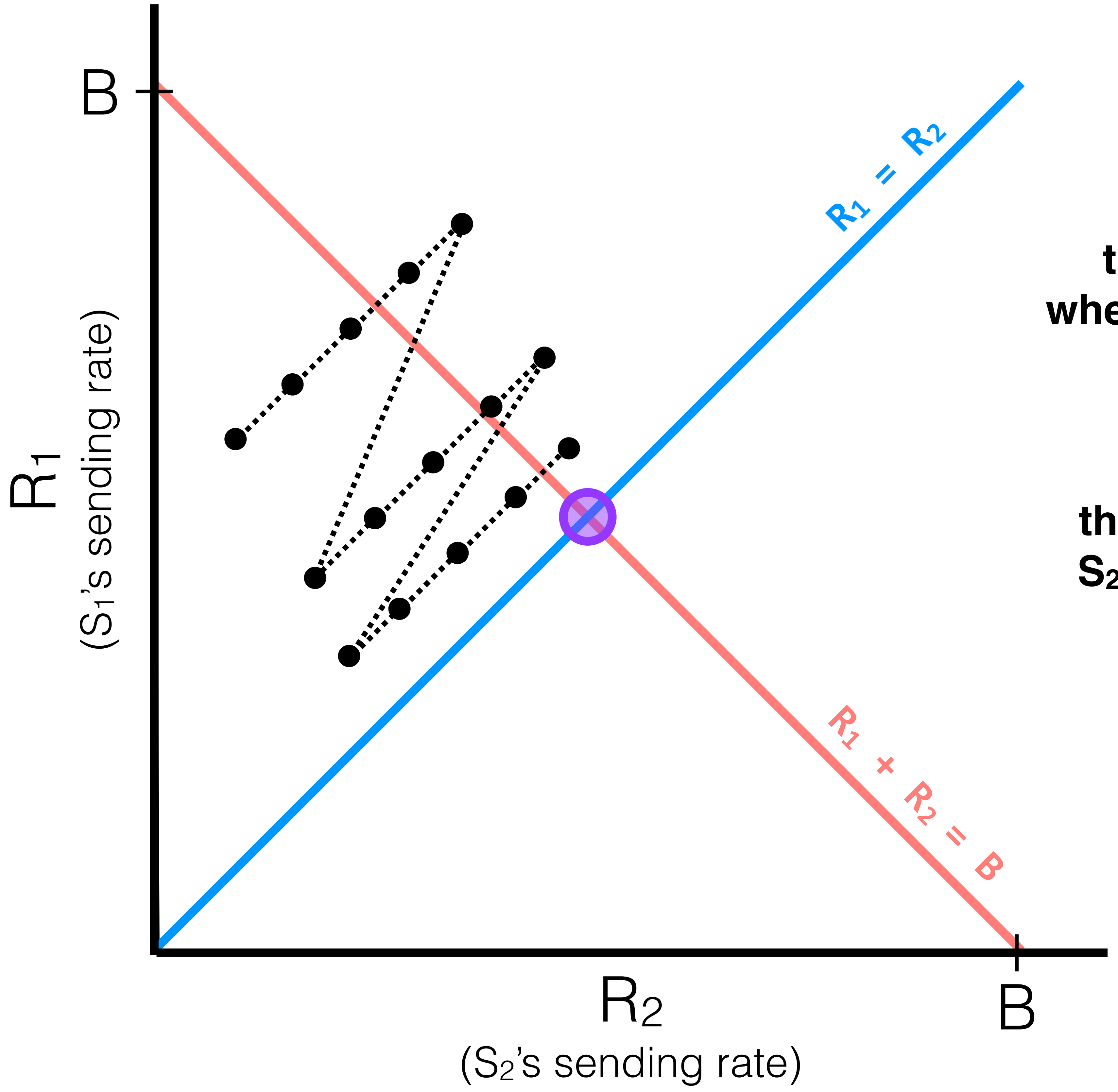
the network is fair when S_1 and S_2 are sending at the same rate

efficiency: minimize drops, minimize delay, maximize bottleneck utilization

fairness: under infinite offered load, split bandwidth evenly among all sources sharing a bottleneck

AIMD: every RTT, if there is no loss, `W = W + 1`; else, `W = W/2`

congestion control: controlling the source rates to achieve **efficiency** and **fairness**



efficiency

(utilization)

the network is fully utilized when the bottleneck link is "full"

fairness

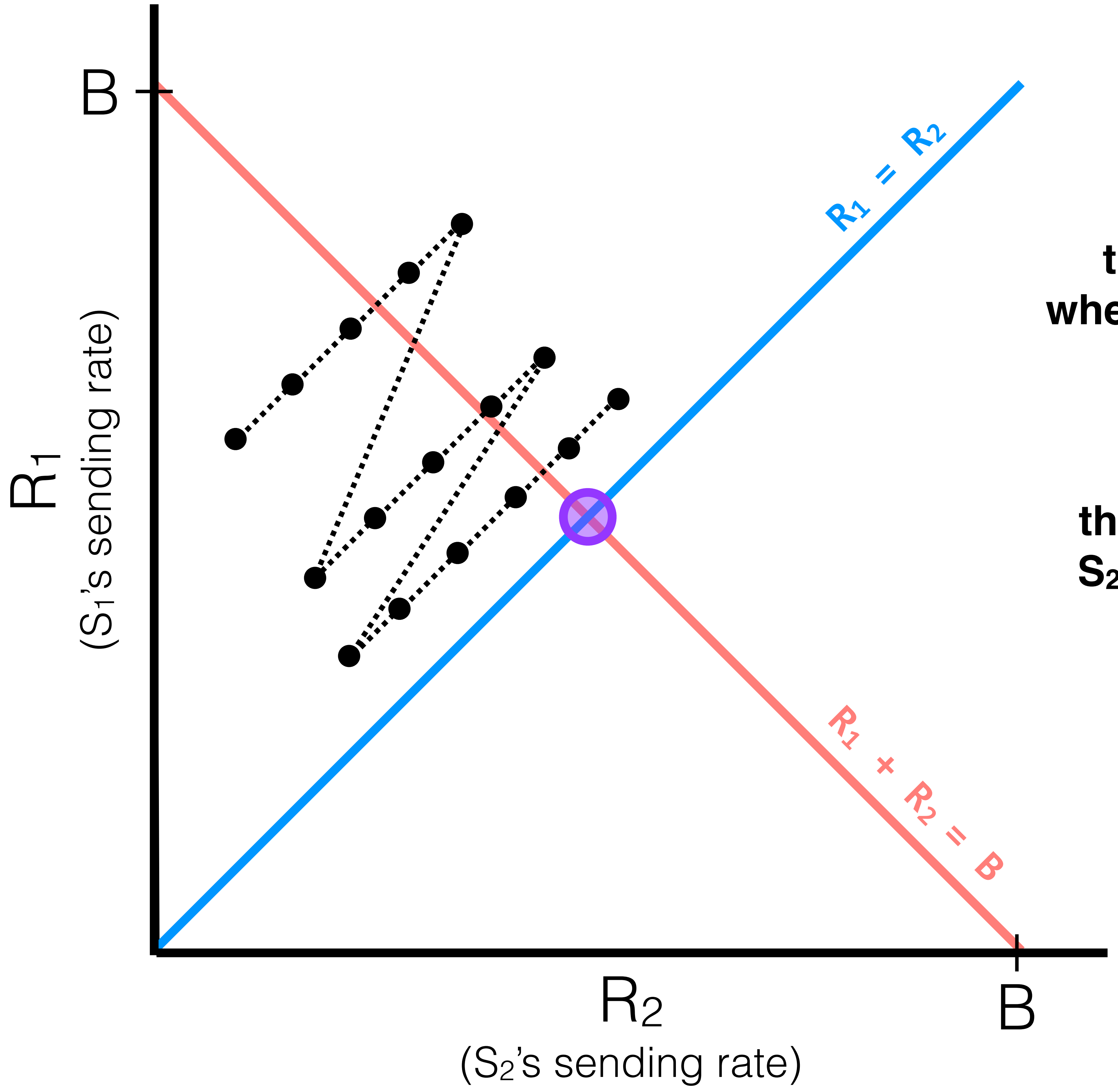
the network is fair when S_1 and S_2 are sending at the same rate

efficiency: minimize drops, minimize delay, maximize bottleneck utilization

fairness: under infinite offered load, split bandwidth evenly among all sources sharing a bottleneck

AIMD: every RTT, if there is no loss, `W = W + 1`; else, `W = W/2`

congestion control: controlling the source rates to achieve **efficiency** and **fairness**



efficiency
(utilization)

the network is fully utilized when the bottleneck link is "full"

fairness

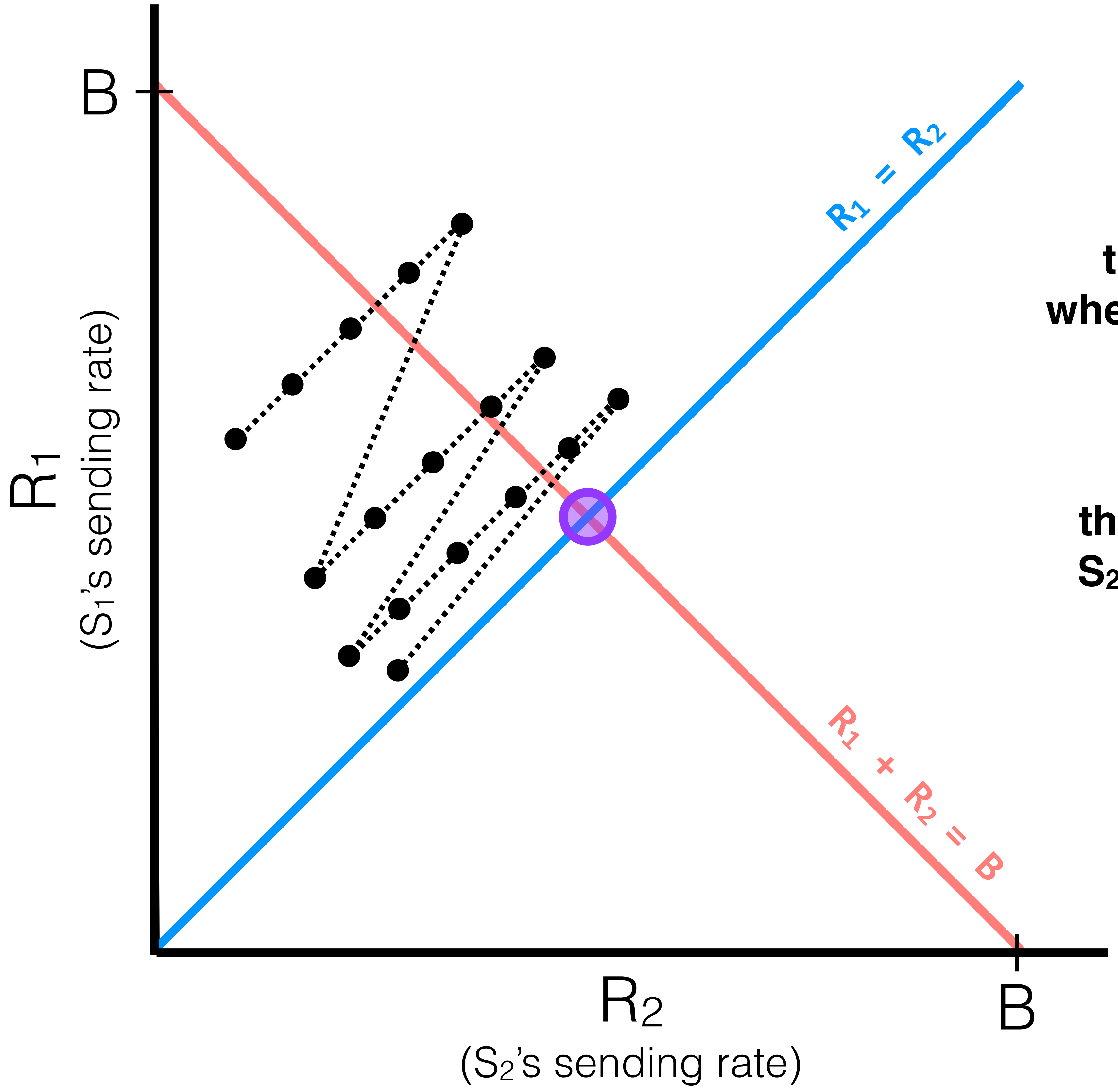
the network is fair when S_1 and S_2 are sending at the same rate

efficiency: minimize drops, minimize delay, maximize bottleneck utilization

fairness: under infinite offered load, split bandwidth evenly among all sources sharing a bottleneck

AIMD: every RTT, if there is no loss, `W = W + 1`; else, `W = W/2`

congestion control: controlling the source rates to achieve **efficiency** and **fairness**



efficiency

(utilization)

the network is fully utilized when the bottleneck link is "full"

fairness

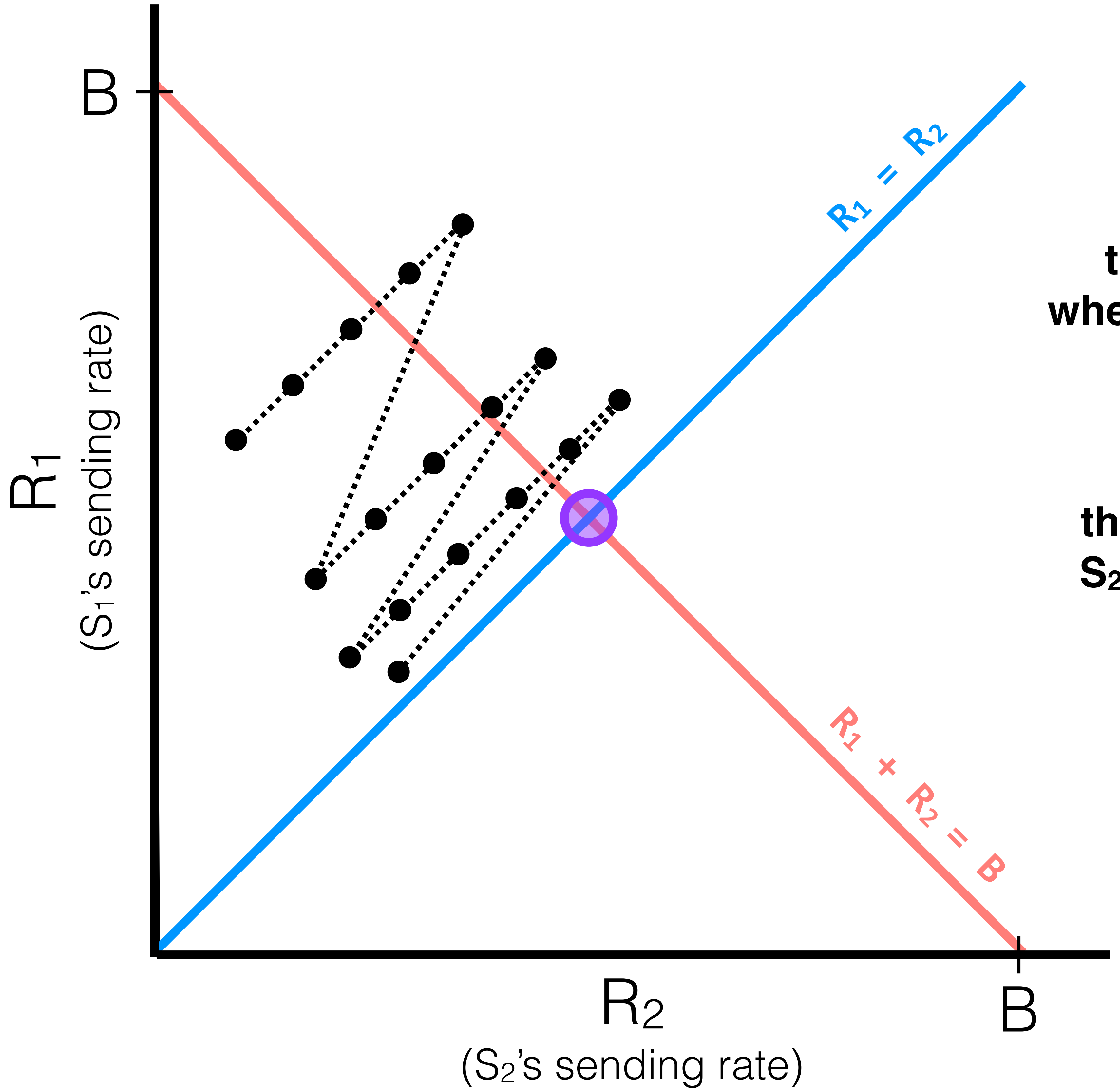
the network is fair when S_1 and S_2 are sending at the same rate

efficiency: minimize drops, minimize delay, maximize bottleneck utilization

fairness: under infinite offered load, split bandwidth evenly among all sources sharing a bottleneck

AIMD: every RTT, if there is no loss, `W = W + 1`; else, `W = W/2`

congestion control: controlling the source rates to achieve **efficiency** and **fairness**



efficiency

(utilization)

the network is fully utilized when the bottleneck link is "full"

fairness

the network is fair when S_1 and S_2 are sending at the same rate

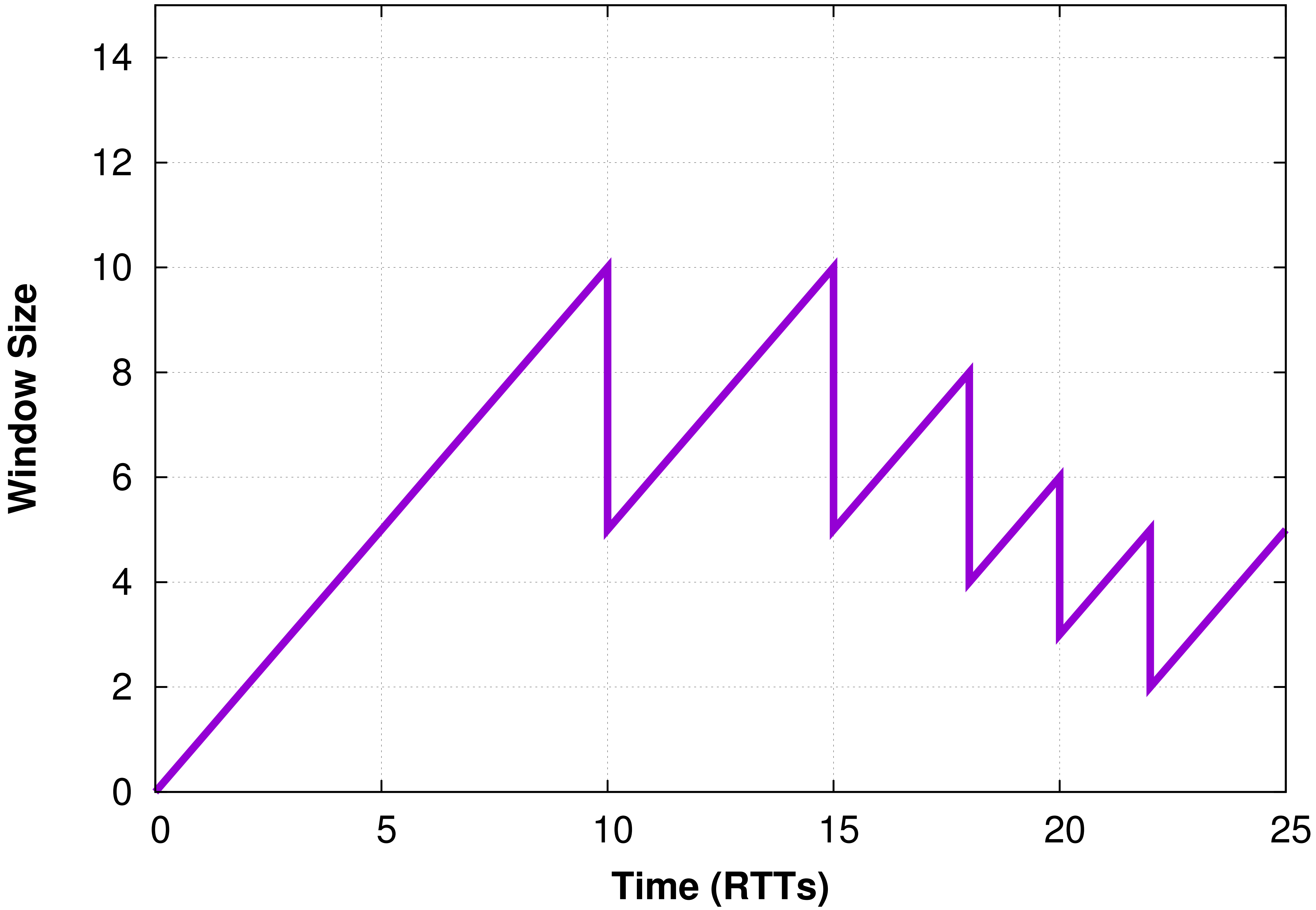
eventually, R1 and R2 will come to oscillate around the fixed point

efficiency: minimize drops, minimize delay, maximize bottleneck utilization

fairness: under infinite offered load, split bandwidth evenly among all sources sharing a bottleneck

AIMD: every RTT, if there is no loss, `W = W + 1`; else, `W = W/2`

congestion control: controlling the source rates to achieve **efficiency** and **fairness**

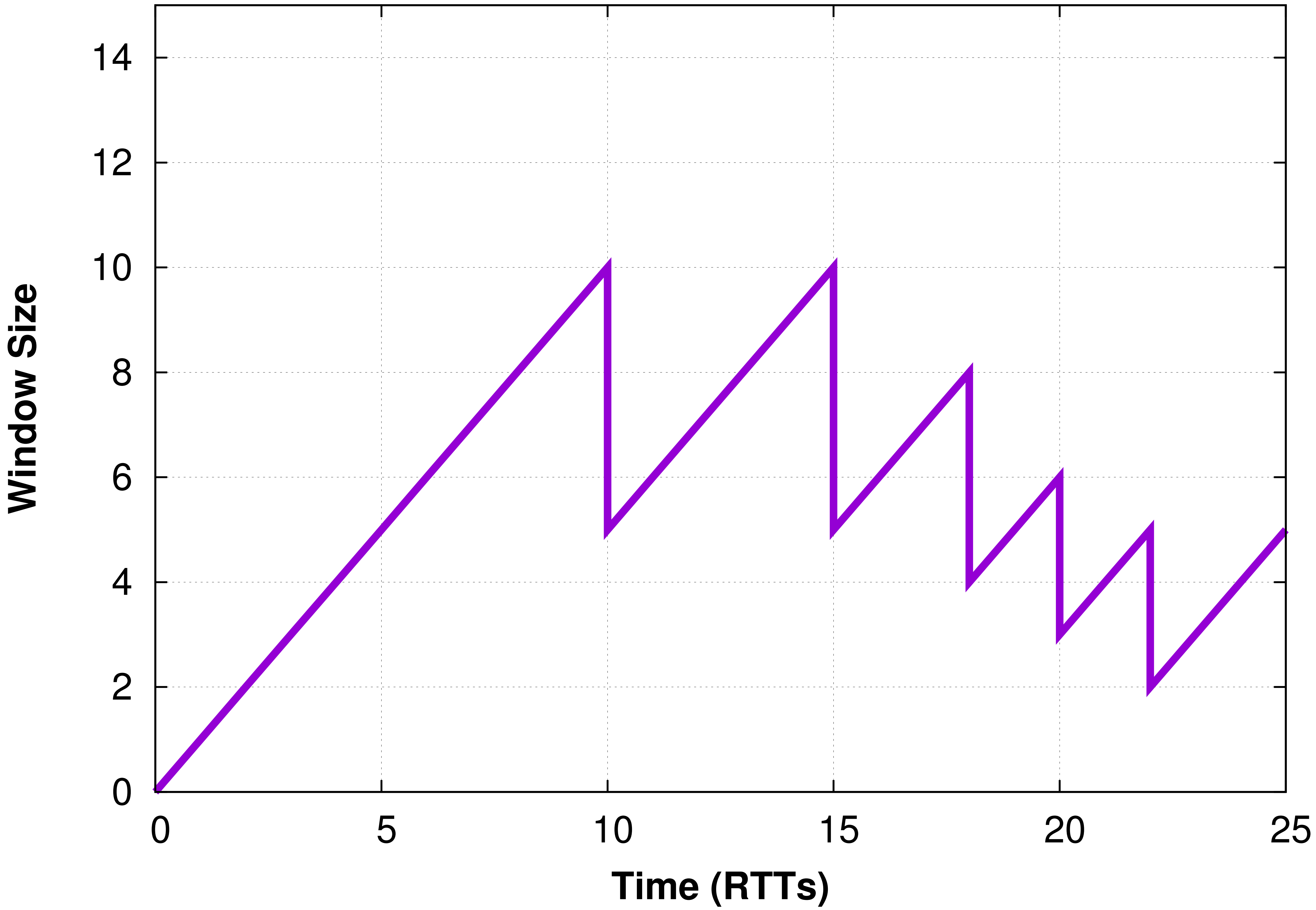


efficiency: minimize drops, minimize delay, maximize bottleneck utilization

fairness: under infinite offered load, split bandwidth evenly among all sources sharing a bottleneck

AIMD: every RTT, if there is no loss, `W = W + 1`; else, `W = W/2`

congestion control: controlling the source rates to achieve **efficiency** and **fairness**



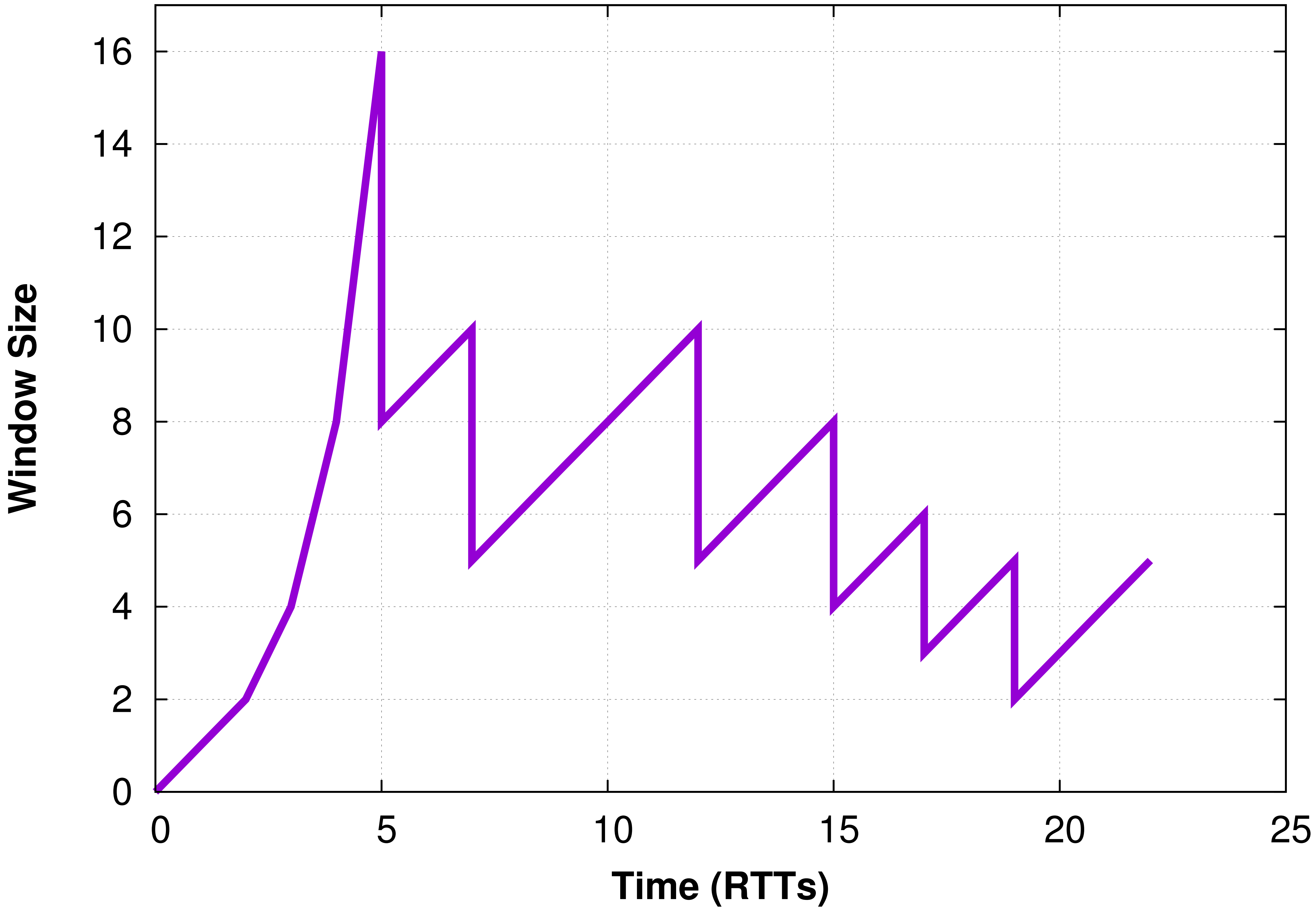
efficiency: minimize drops, minimize delay, maximize bottleneck utilization

fairness: under infinite offered load, split bandwidth evenly among all sources sharing a bottleneck

AIMD: every RTT, if there is no loss, `W = W + 1`; else, `W = W/2`

slow-start: at the start of the connection, double `W` every RTT

congestion control: controlling the source rates to achieve **efficiency** and **fairness**



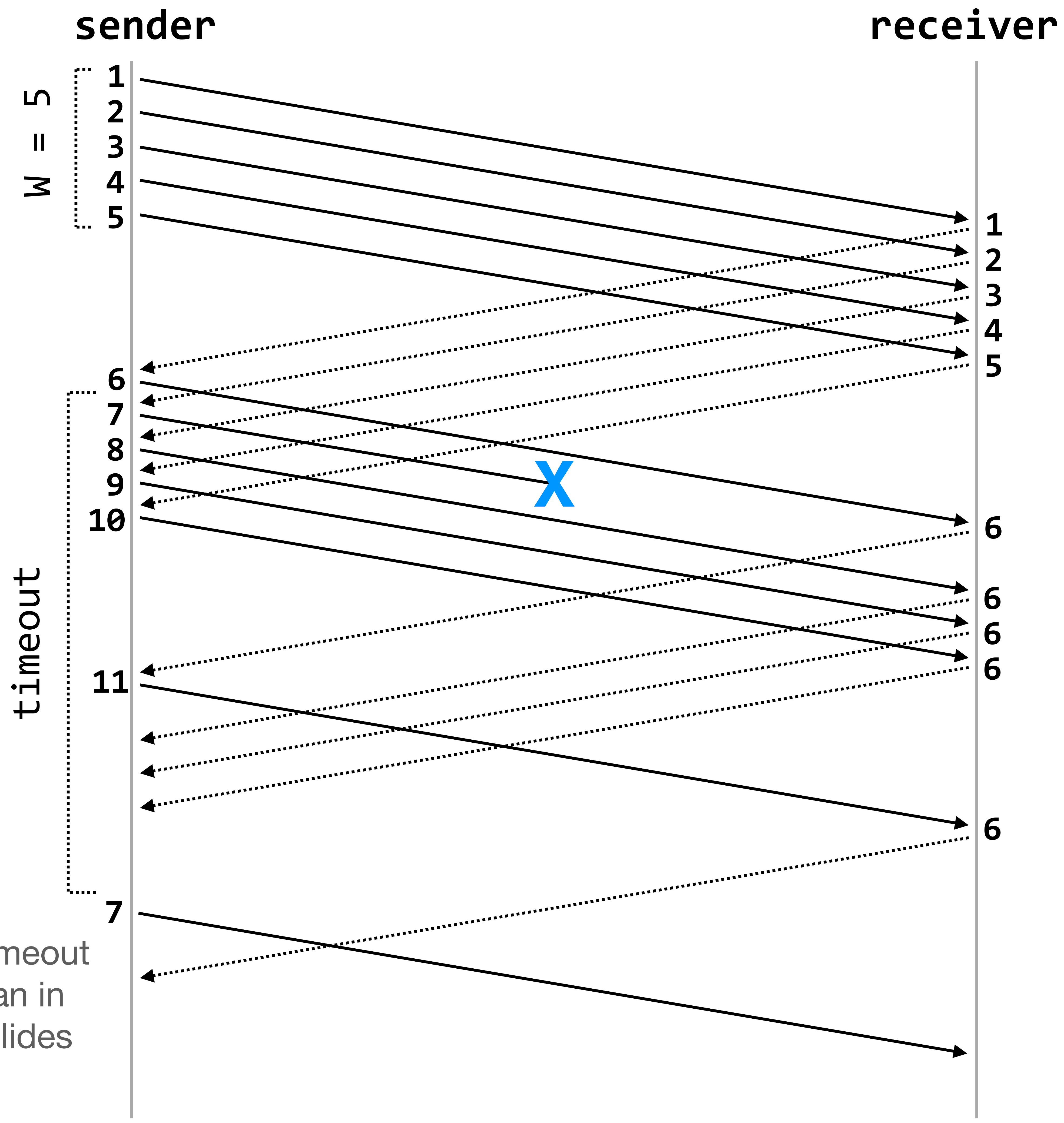
efficiency: minimize drops, minimize delay, maximize bottleneck utilization

fairness: under infinite offered load, split bandwidth evenly among all sources sharing a bottleneck

AIMD: every RTT, if there is no loss, `W = W + 1`; else, `W = W/2`

slow-start: at the start of the connection, double `W` every RTT

congestion control: controlling the source rates to achieve **efficiency** and **fairness**



notice this timeout is larger than in the earlier slides

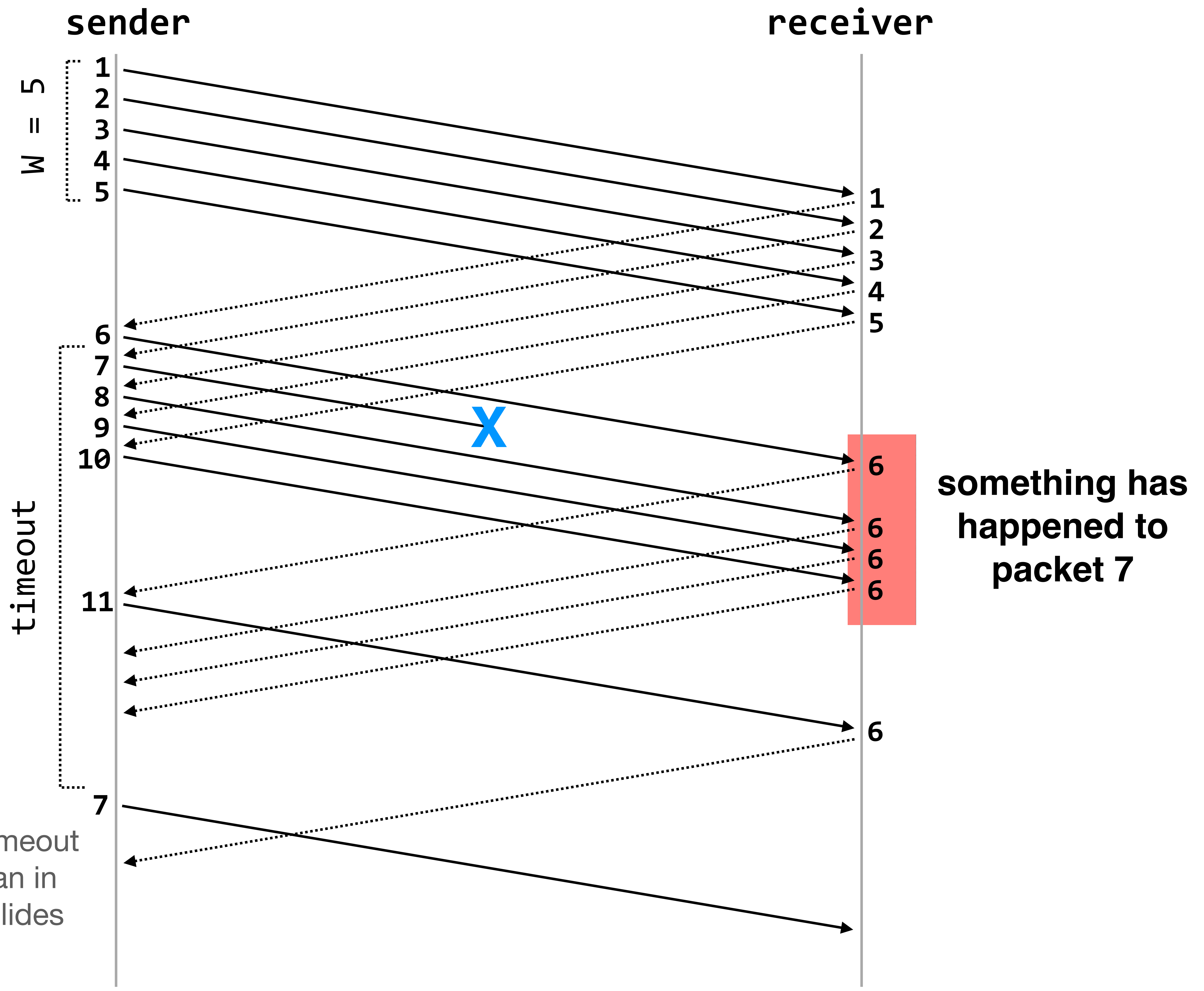
efficiency: minimize drops, minimize delay, maximize bottleneck utilization

fairness: under infinite offered load, split bandwidth evenly among all sources sharing a bottleneck

AIMD: every RTT, if there is no loss, `W = W + 1`; else, `W = W/2`

slow-start: at the start of the connection, double `W` every RTT

congestion control: controlling the source rates to achieve **efficiency** and **fairness**



notice this timeout is larger than in the earlier slides

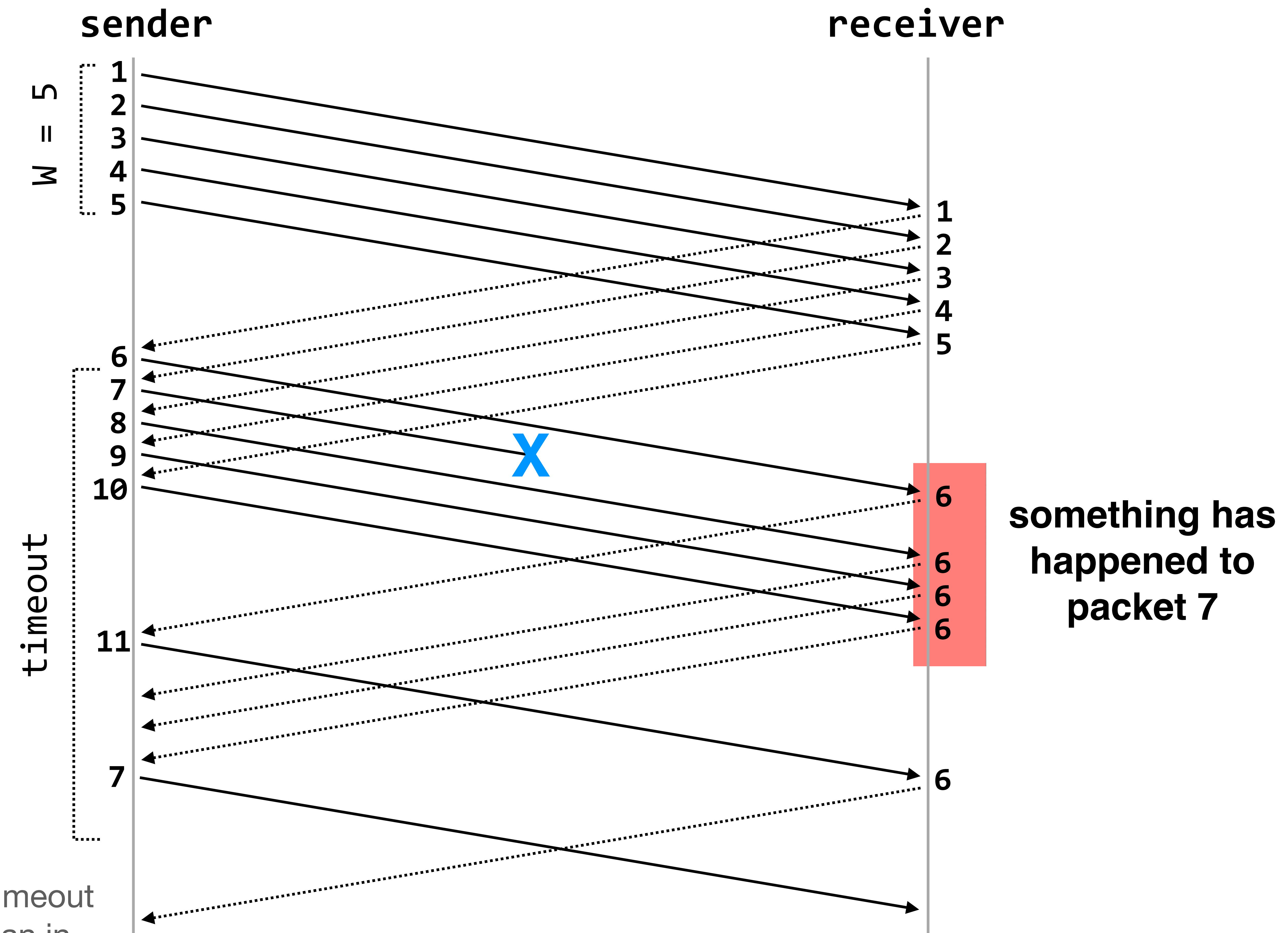
efficiency: minimize drops, minimize delay, maximize bottleneck utilization

fairness: under infinite offered load, split bandwidth evenly among all sources sharing a bottleneck

AIMD: every RTT, if there is no loss, `W = W + 1`; else, `W = W/2`

slow-start: at the start of the connection, double `W` every RTT

congestion control: controlling the source rates to achieve **efficiency** and **fairness**



notice this timeout is larger than in the earlier slides

efficiency: minimize drops, minimize delay, maximize bottleneck utilization

fairness: under infinite offered load, split bandwidth evenly among all sources sharing a bottleneck

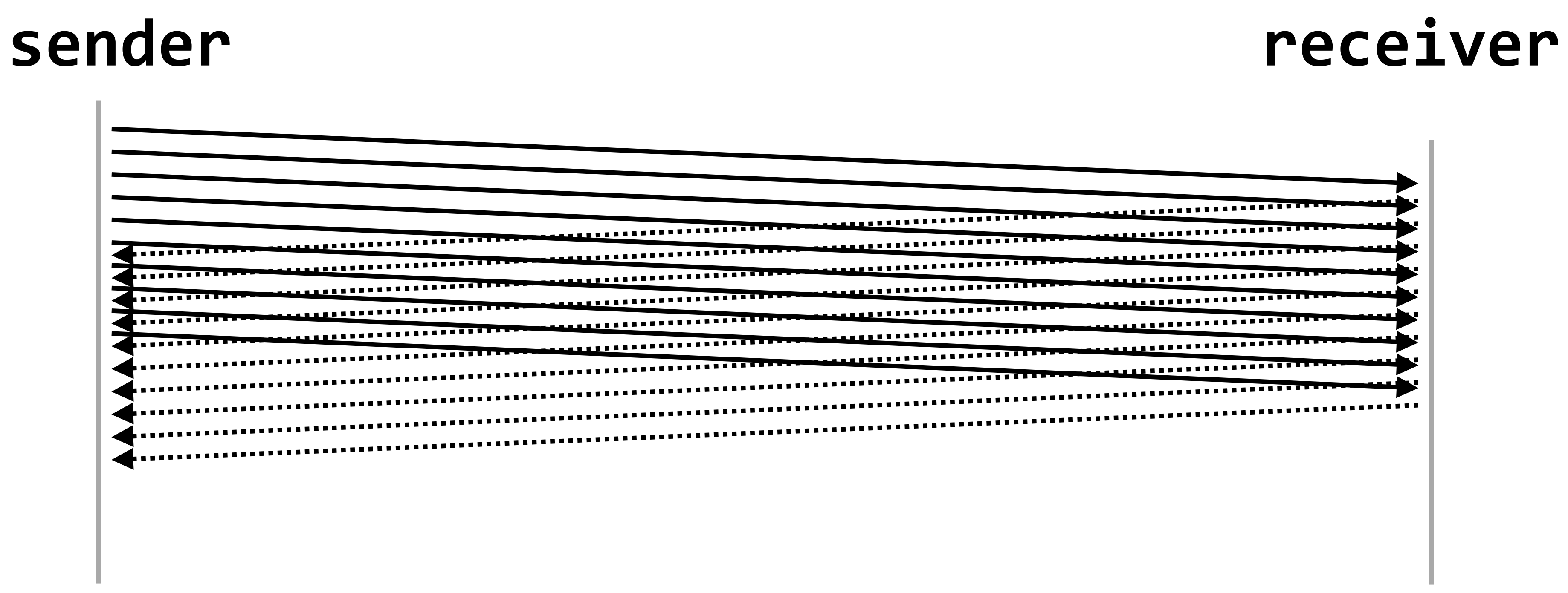
AIMD: every RTT, if there is no loss, `W = W + 1`; else, `W = W/2`

slow-start: at the start of the connection, double `W` every RTT

fast retransmit/fast recovery: retransmit packet `k+1` as soon as four ACKs with sequence number `k` are received

(four = original ACK + 3 "dup" ACKs)

congestion control: controlling the source rates to achieve **efficiency** and **fairness**



in practice, if a single packet is lost, the three "dup" ACKs will be received before the timeout for that packet expires

efficiency: minimize drops, minimize delay, maximize bottleneck utilization

fairness: under infinite offered load, split bandwidth evenly among all sources sharing a bottleneck

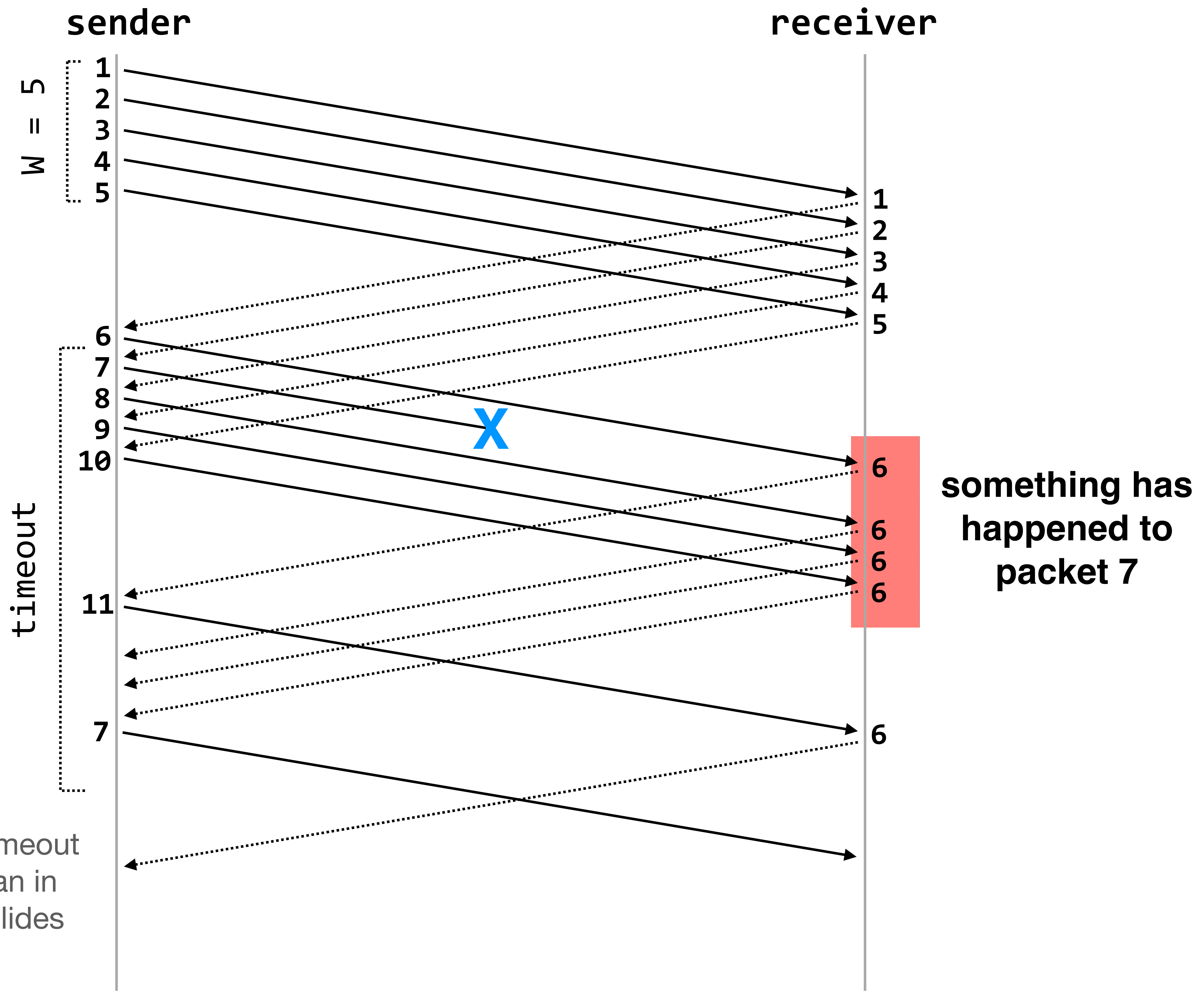
AIMD: every RTT, if there is no loss, `W = W + 1`; else, `W = W/2`

slow-start: at the start of the connection, double `W` every RTT

fast retransmit/fast recovery: retransmit packet **`k+1`** as soon as four ACKs with sequence number **`k`** are received

(four = original ACK + 3 "dup" ACKs)

congestion control: controlling the source rates to achieve **efficiency** and **fairness**



something has happened to packet 7

notice this timeout is larger than in the earlier slides

efficiency: minimize drops, minimize delay, maximize bottleneck utilization

fairness: under infinite offered load, split bandwidth evenly among all sources sharing a bottleneck

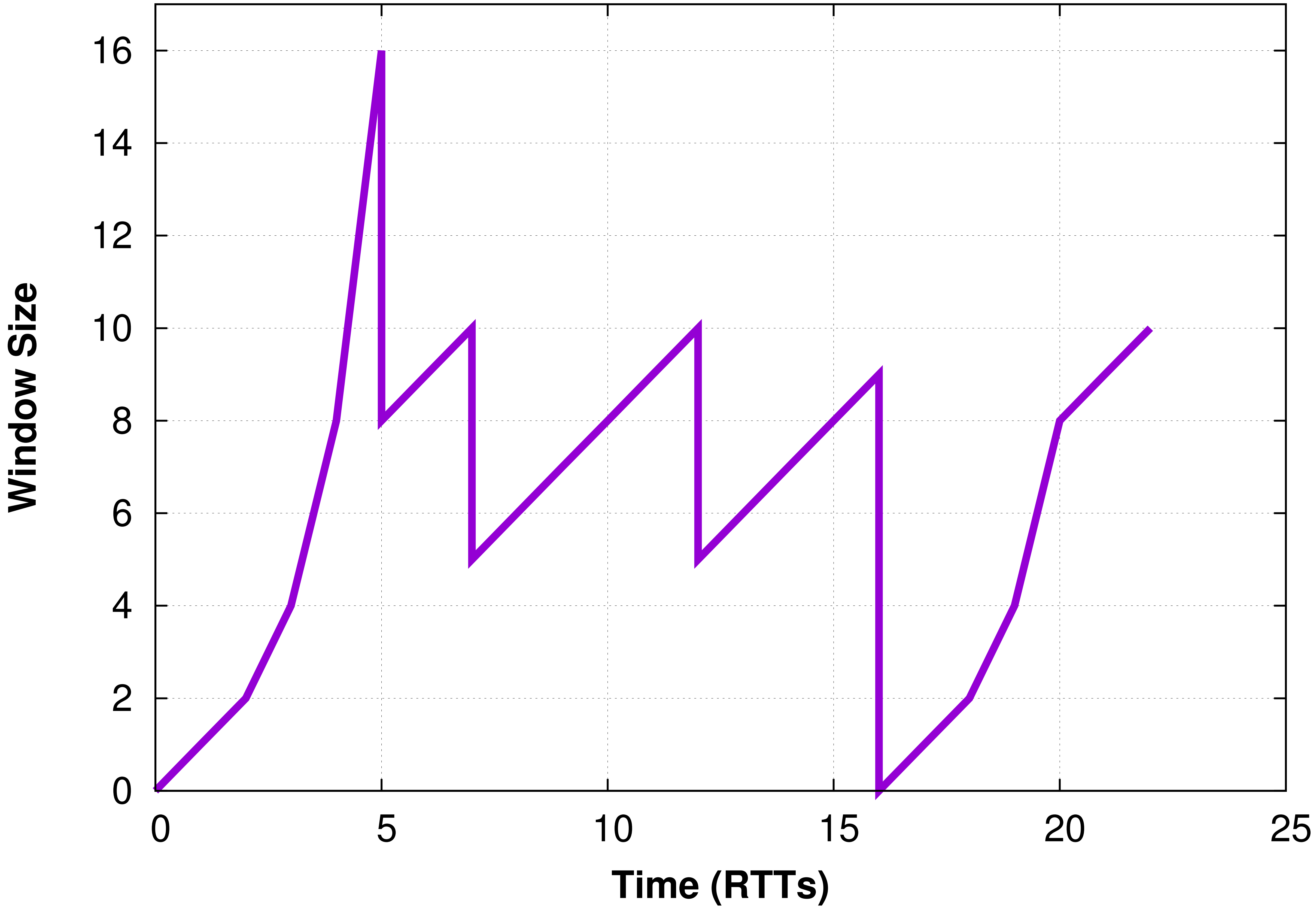
AIMD: every RTT, if there is no loss, `W = W + 1`; else, `W = W/2`

slow-start: at the start of the connection, double `W` every RTT

fast retransmit/fast recovery: retransmit packet `k+1` as soon as four ACKs with sequence number `k` are received

(four = original ACK + 3 "dup" ACKs)

congestion control: controlling the source rates to achieve **efficiency** and **fairness**



in practice, a retransmission due to a timeout happens when there is *significant* loss. senders are even more conservative, dropping their window back down to 1

efficiency: minimize drops, minimize delay, maximize bottleneck utilization

fairness: under infinite offered load, split bandwidth evenly among all sources sharing a bottleneck

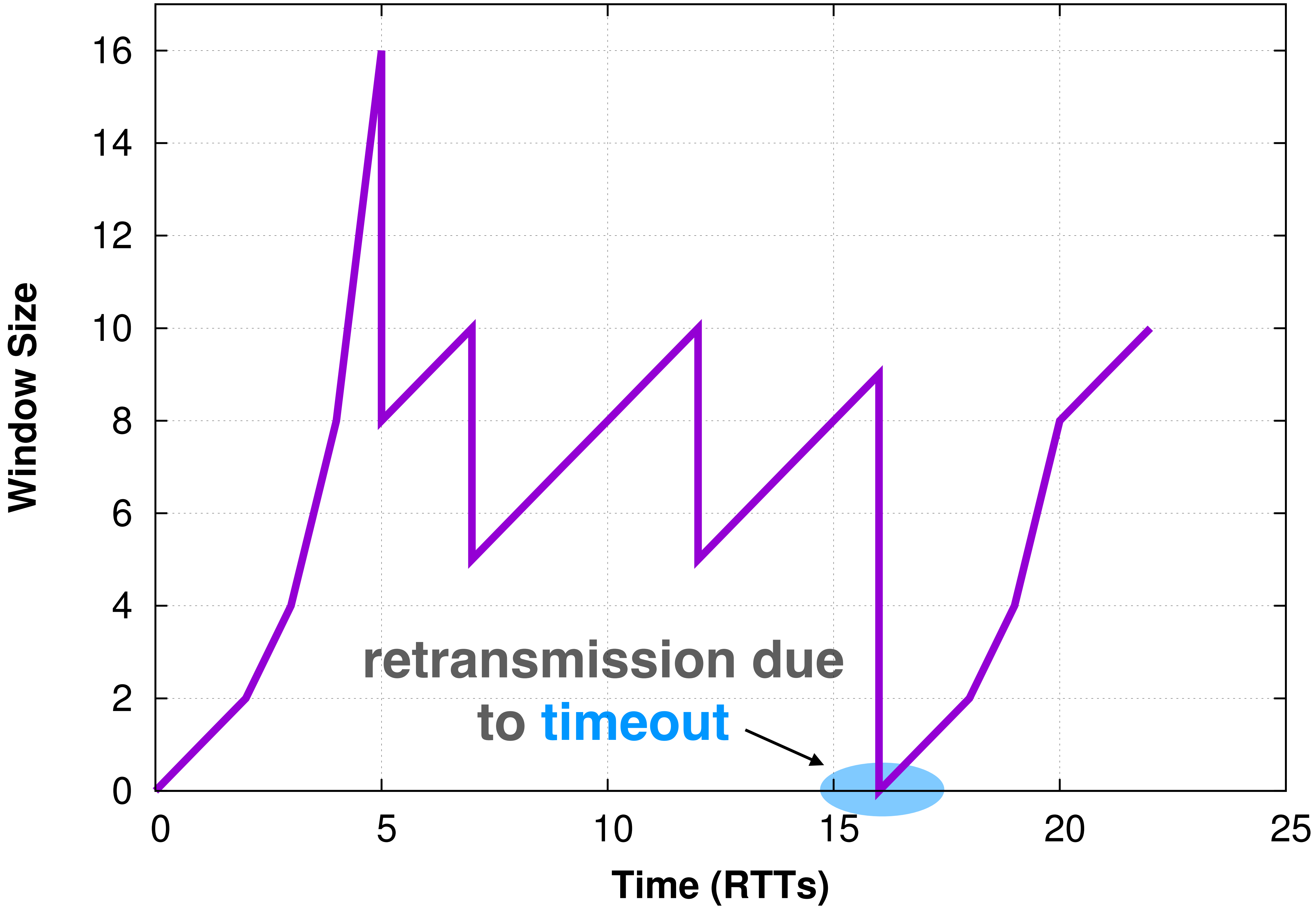
AIMD: every RTT, if there is no loss, `W = W + 1`; else, `W = W/2`

slow-start: at the start of the connection, double `W` every RTT

fast retransmit/fast recovery: retransmit packet `k+1` as soon as four ACKs with sequence number `k` are received

(four = original ACK + 3 “dup” ACKs)

congestion control: controlling the source rates to achieve **efficiency** and **fairness**



retransmission due to timeout

in practice, a retransmission due to a timeout happens when there is *significant* loss. senders are even more conservative, dropping their window back down to 1

efficiency: minimize drops, minimize delay, maximize bottleneck utilization

fairness: under infinite offered load, split bandwidth evenly among all sources sharing a bottleneck

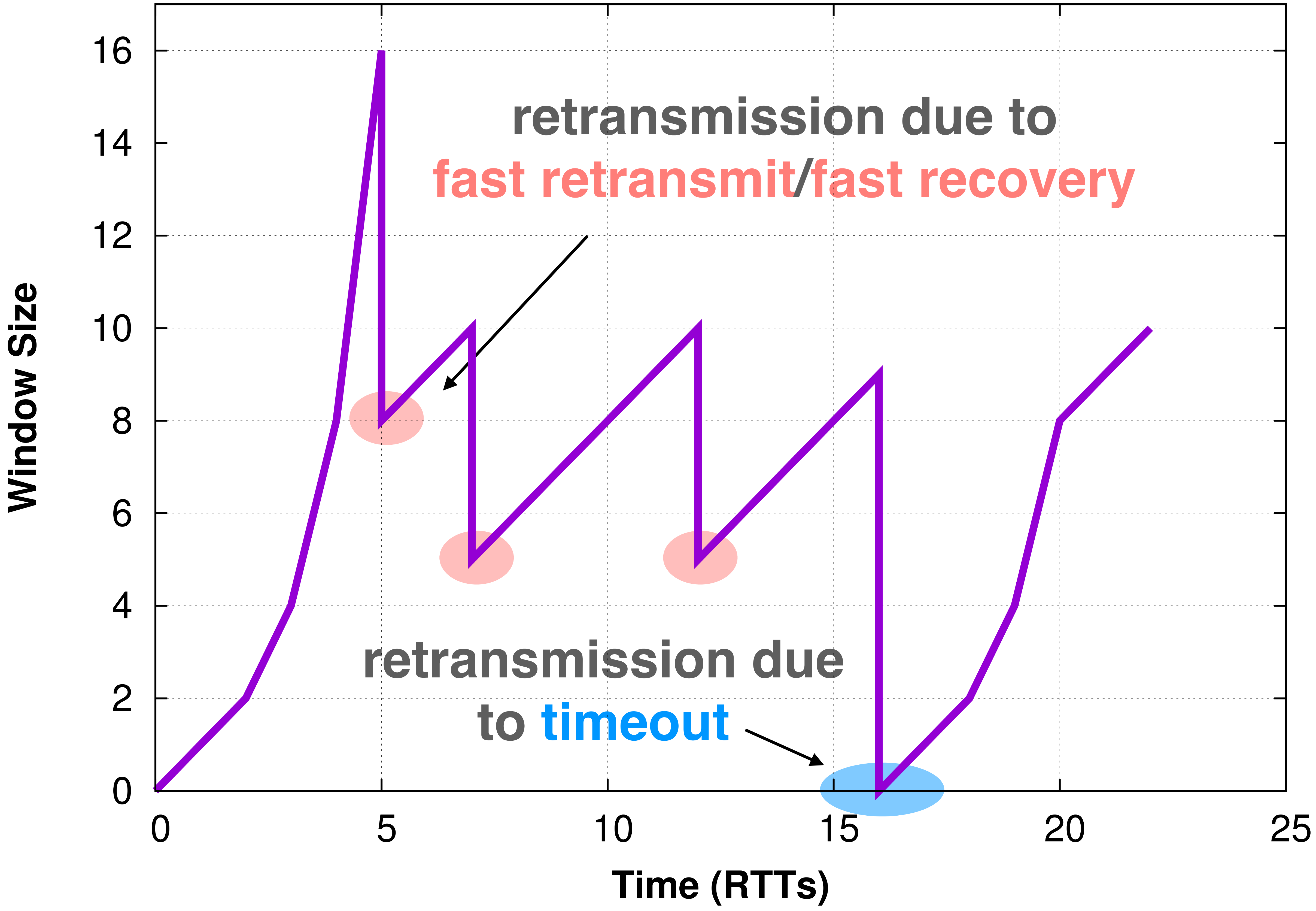
AIMD: every RTT, if there is no loss, `W = W + 1`; else, `W = W/2`

slow-start: at the start of the connection, double `W` every RTT

fast retransmit/fast recovery: retransmit packet `k+1` as soon as four ACKs with sequence number `k` are received

(four = original ACK + 3 “dup” ACKs)

congestion control: controlling the source rates to achieve **efficiency** and **fairness**



in practice, a retransmission due to a timeout happens when there is *significant* loss. senders are even more conservative, dropping their window back down to 1

efficiency: minimize drops, minimize delay, maximize bottleneck utilization

fairness: under infinite offered load, split bandwidth evenly among all sources sharing a bottleneck

AIMD: every RTT, if there is no loss, `W = W + 1`; else, `W = W/2`

slow-start: at the start of the connection, double `W` every RTT

fast retransmit/fast recovery: retransmit packet `k+1` as soon as four ACKs with sequence number `k` are received

(four = original ACK + 3 “dup” ACKs)

congestion control: controlling the source rates to achieve **efficiency** and **fairness**

efficiency: minimize drops, minimize delay, maximize bottleneck utilization

fairness: under infinite offered load, split bandwidth evenly among all sources sharing a bottleneck

AIMD: every RTT, if there is no loss, `W = W + 1`; else, `W = W/2`

slow-start: at the start of the connection, double `W` every RTT

fast retransmit/fast recovery: retransmit packet `k+1` as soon as four ACKs with sequence number `k` are received

(four = original ACK + 3 “dup” ACKs)

congestion control: controlling the source rates to achieve **efficiency** and **fairness**

in certain types of networks, this style of congestion control can make these problems *worse*

efficiency: minimize drops, minimize delay, maximize bottleneck utilization

fairness: under infinite offered load, split bandwidth evenly among all sources sharing a bottleneck

AIMD: every RTT, if there is no loss, `W = W + 1`; else, `W = W/2`

slow-start: at the start of the connection, double `W` every RTT

fast retransmit/fast recovery: retransmit packet `k+1` as soon as four ACKs with sequence number `k` are received

(four = original ACK + 3 “dup” ACKs)

congestion control: controlling the source rates to achieve **efficiency** and **fairness**

in certain types of networks, this style of congestion control can make these problems *worse*

in practice, fairness is tough to define and assess

efficiency: minimize drops, minimize delay, maximize bottleneck utilization

fairness: under infinite offered load, split bandwidth evenly among all sources sharing a bottleneck

AIMD: every RTT, if there is no loss, `W = W + 1`; else, `W = W/2`

slow-start: at the start of the connection, double `W` every RTT

fast retransmit/fast recovery: retransmit packet `k+1` as soon as four ACKs with sequence number `k` are received

(four = original ACK + 3 “dup” ACKs)

congestion control: controlling the source rates to achieve **efficiency** and **fairness**

in certain types of networks, this style of congestion control can make these problems *worse*

in practice, fairness is tough to define and assess

AIMD is not the final word in congestion avoidance; modern versions (e.g. CUBIC TCP) use different rules to set the window size

efficiency: minimize drops, minimize delay, maximize bottleneck utilization

fairness: under infinite offered load, split bandwidth evenly among all sources sharing a bottleneck

AIMD: every RTT, if there is no loss, `W = W + 1`; else, `W = W/2`

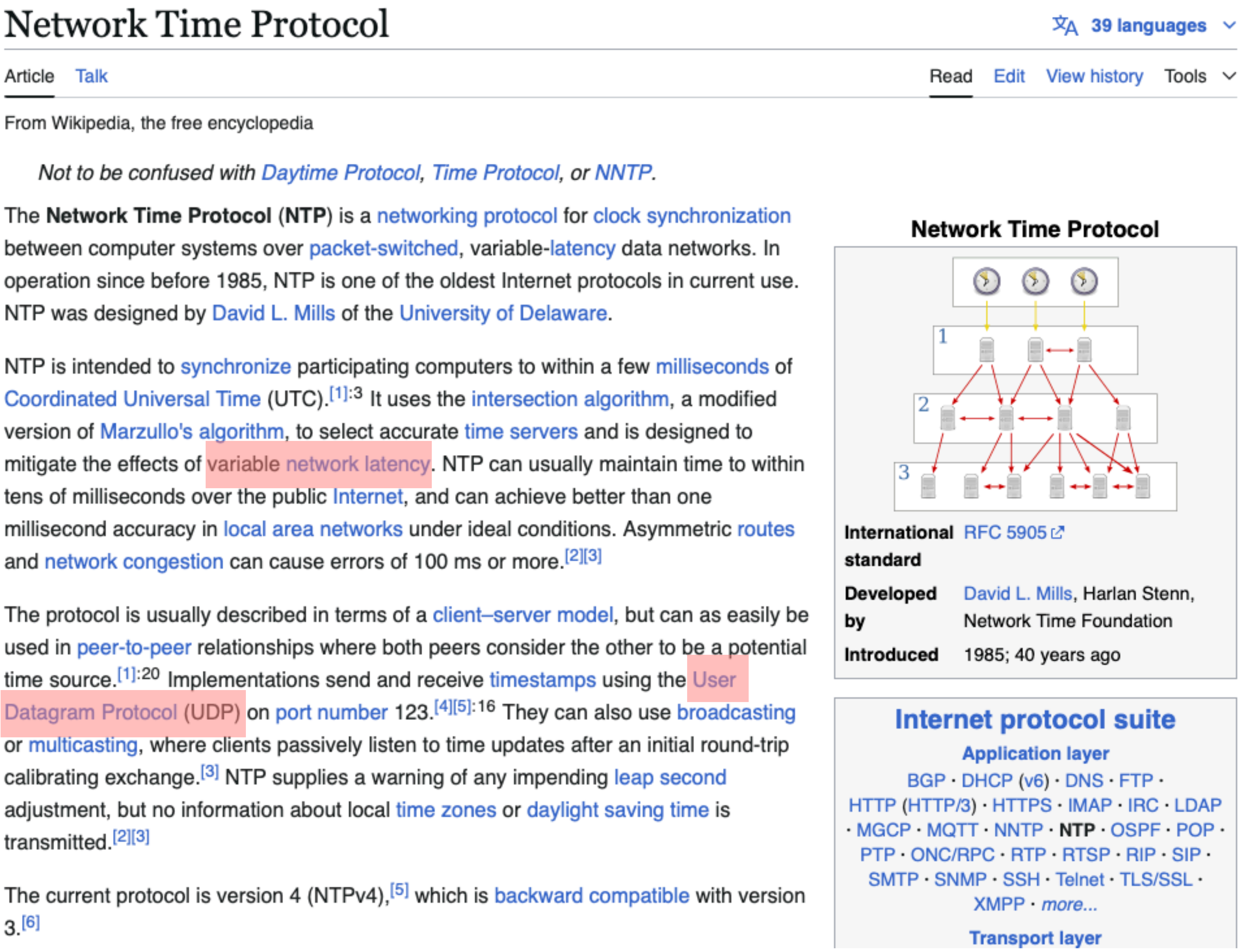
slow-start: at the start of the connection, double `W` every RTT

fast retransmit/fast recovery: retransmit packet `k+1` as soon as four ACKs with sequence number `k` are received

(four = original ACK + 3 “dup” ACKs)

6.1800 in the news

the **network time protocol** synchronizes clocks to UTC



Network Time Protocol

From Wikipedia, the free encyclopedia

Not to be confused with Daytime Protocol, Time Protocol, or NNTP.

The **Network Time Protocol** (**NTP**) is a networking protocol for clock synchronization between computer systems over packet-switched, variable-latency data networks. In operation since before 1985, NTP is one of the oldest Internet protocols in current use. NTP was designed by David L. Mills of the University of Delaware.

NTP is intended to synchronize participating computers to within a few milliseconds of Coordinated Universal Time (UTC).[1]:3 It uses the intersection algorithm, a modified version of Marzullo's algorithm, to select accurate time servers and is designed to mitigate the effects of variable network latency. NTP can usually maintain time to within tens of milliseconds over the public Internet, and can achieve better than one millisecond accuracy in local area networks under ideal conditions. Asymmetric routes and network congestion can cause errors of 100 ms or more.[2][3]

The protocol is usually described in terms of a client–server model, but can as easily be used in peer-to-peer relationships where both peers consider the other to be a potential time source.[1]:20 Implementations send and receive timestamps using the User Datagram Protocol (UDP) on port number 123.[4][5]:16 They can also use broadcasting or multicasting, where clients passively listen to time updates after an initial round-trip calibrating exchange.[3] NTP supplies a warning of any impending leap second adjustment, but no information about local time zones or daylight saving time is transmitted.[2][3]

The current protocol is version 4 (NTPv4),[5] which is backward compatible with version 3.[6]

Network Time Protocol

International standard	RFC 5905
Developed by	David L. Mills, Harlan Stenn, Network Time Foundation
Introduced	1985; 40 years ago

Internet protocol suite

Application layer

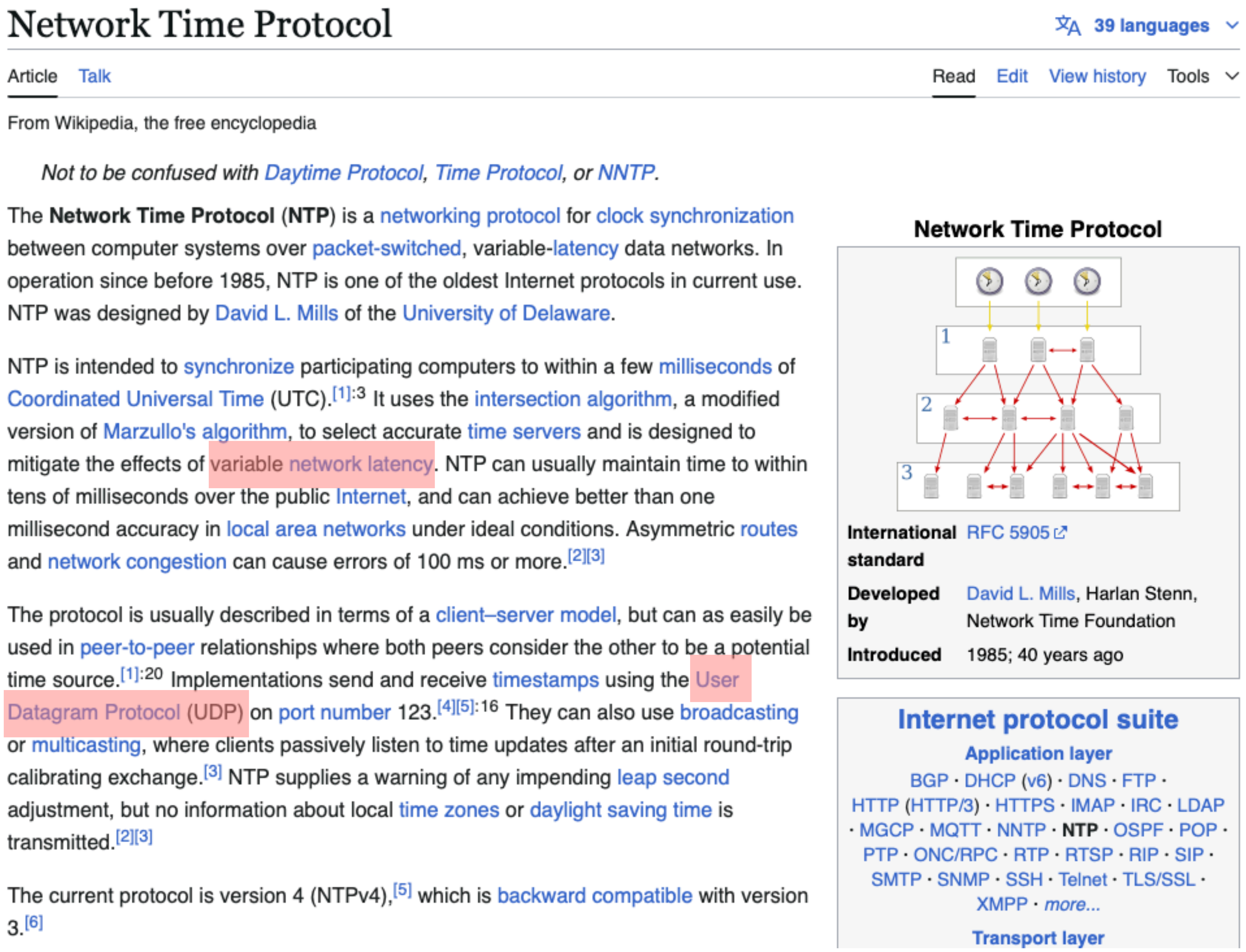
BGP · DHCP (v6) · DNS · FTP · HTTP (HTTP/3) · HTTPS · IMAP · IRC · LDAP · MGCP · MQTT · NNTP · **NTP** · OSPF · POP · PTP · ONC/RPC · RTP · RTSP · RIP · SIP · SMTP · SNMP · SSH · Telnet · TLS/SSL · XMPP · *more...*

Transport layer

6.1800 in the news

the **network time protocol** synchronizes clocks to UTC

queues growing (and shrinking) in a network causes latency to be variable



Network Time Protocol

From Wikipedia, the free encyclopedia

Not to be confused with Daytime Protocol, Time Protocol, or NNTP.

The **Network Time Protocol** (**NTP**) is a networking protocol for clock synchronization between computer systems over packet-switched, variable-latency data networks. In operation since before 1985, NTP is one of the oldest Internet protocols in current use. NTP was designed by David L. Mills of the University of Delaware.

NTP is intended to synchronize participating computers to within a few milliseconds of Coordinated Universal Time (UTC).[1]:3 It uses the intersection algorithm, a modified version of Marzullo's algorithm, to select accurate time servers and is designed to mitigate the effects of variable network latency. NTP can usually maintain time to within tens of milliseconds over the public Internet, and can achieve better than one millisecond accuracy in local area networks under ideal conditions. Asymmetric routes and network congestion can cause errors of 100 ms or more.[2][3]

The protocol is usually described in terms of a client–server model, but can as easily be used in peer-to-peer relationships where both peers consider the other to be a potential time source.[1]:20 Implementations send and receive timestamps using the User Datagram Protocol (UDP) on port number 123.[4][5]:16 They can also use broadcasting or multicasting, where clients passively listen to time updates after an initial round-trip calibrating exchange.[3] NTP supplies a warning of any impending leap second adjustment, but no information about local time zones or daylight saving time is transmitted.[2][3]

The current protocol is version 4 (NTPv4),[5] which is backward compatible with version 3.[6]

Network Time Protocol

International standard	RFC 5905
Developed by	David L. Mills, Harlan Stenn, Network Time Foundation
Introduced	1985; 40 years ago

Internet protocol suite

Application layer

BGP · DHCP (v6) · DNS · FTP · HTTP (HTTP/3) · HTTPS · IMAP · IRC · LDAP · MGCP · MQTT · NNTP · **NTP** · OSPF · POP · PTP · ONC/RPC · RTP · RTSP · RIP · SIP · SMTP · SNMP · SSH · Telnet · TLS/SSL · XMPP · *more...*

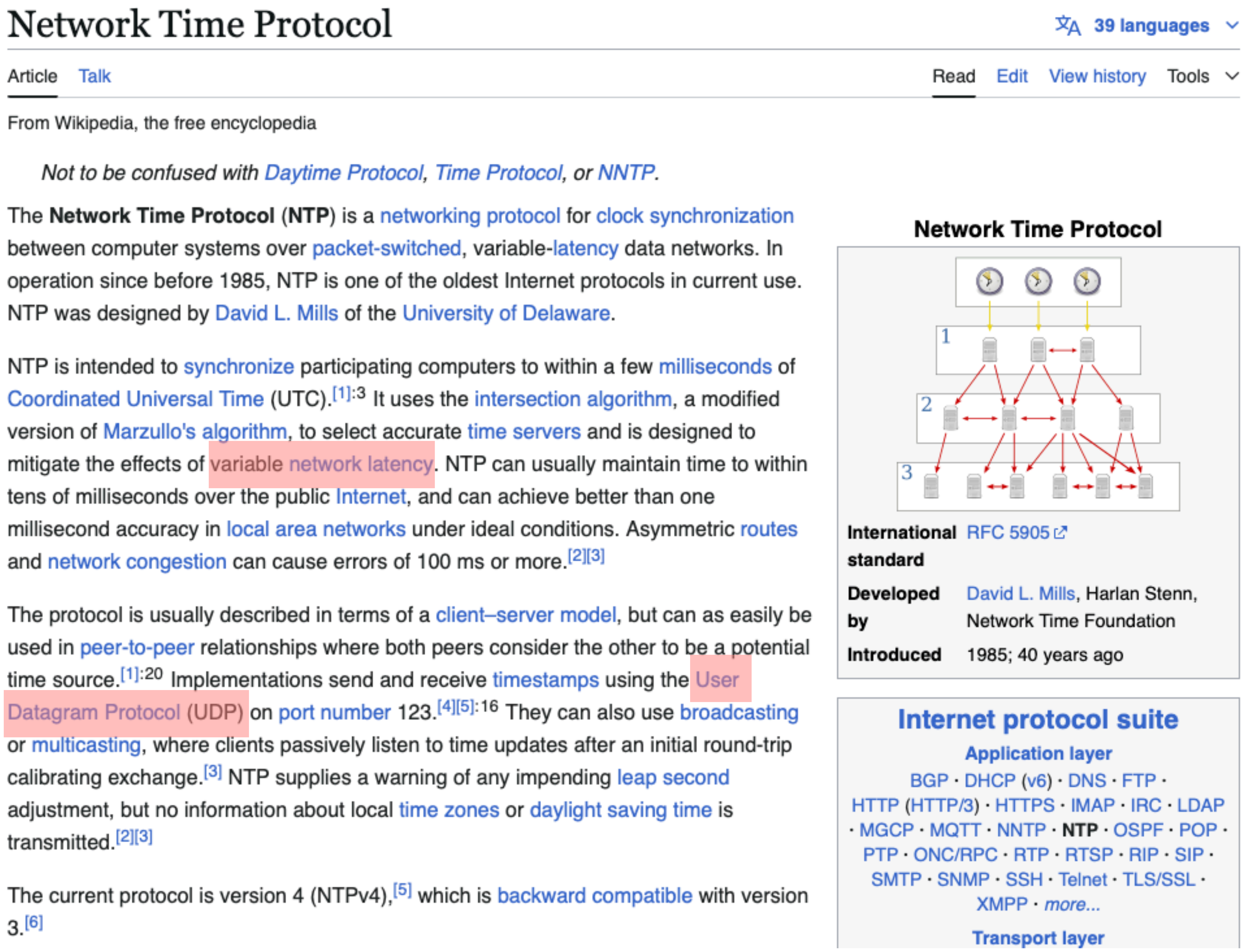
Transport layer

6.1800 in the news

the **network time protocol** synchronizes clocks to UTC

queues growing (and shrinking) in a network causes latency to be variable

UDP has much lower overhead than TCP (smaller packet headers, no congestion control, no error-checking, no connection set-up phase)



Network Time Protocol

From Wikipedia, the free encyclopedia

Not to be confused with Daytime Protocol, Time Protocol, or NNTP.

The **Network Time Protocol** (**NTP**) is a networking protocol for clock synchronization between computer systems over packet-switched, variable-latency data networks. In operation since before 1985, NTP is one of the oldest Internet protocols in current use. NTP was designed by David L. Mills of the University of Delaware.

NTP is intended to synchronize participating computers to within a few milliseconds of Coordinated Universal Time (UTC).[1]:3 It uses the intersection algorithm, a modified version of Marzullo's algorithm, to select accurate time servers and is designed to mitigate the effects of variable network latency. NTP can usually maintain time to within tens of milliseconds over the public Internet, and can achieve better than one millisecond accuracy in local area networks under ideal conditions. Asymmetric routes and network congestion can cause errors of 100 ms or more.[2][3]

The protocol is usually described in terms of a client–server model, but can as easily be used in peer-to-peer relationships where both peers consider the other to be a potential time source.[1]:20 Implementations send and receive timestamps using the User Datagram Protocol (UDP) on port number 123.[4][5]:16 They can also use broadcasting or multicasting, where clients passively listen to time updates after an initial round-trip calibrating exchange.[3] NTP supplies a warning of any impending leap second adjustment, but no information about local time zones or daylight saving time is transmitted.[2][3]

The current protocol is version 4 (NTPv4),[5] which is backward compatible with version 3.[6]

Network Time Protocol

International standard	RFC 5905
Developed by	David L. Mills, Harlan Stenn, Network Time Foundation
Introduced	1985; 40 years ago

Internet protocol suite

Application layer

BGP · DHCP (v6) · DNS · FTP · HTTP (HTTP/3) · HTTPS · IMAP · IRC · LDAP · MGCP · MQTT · NNTP · **NTP** · OSPF · POP · PTP · ONC/RPC · RTP · RTSP · RIP · SIP · SMTP · SNMP · SSH · Telnet · TLS/SSL · XMPP · *more...*

Transport layer

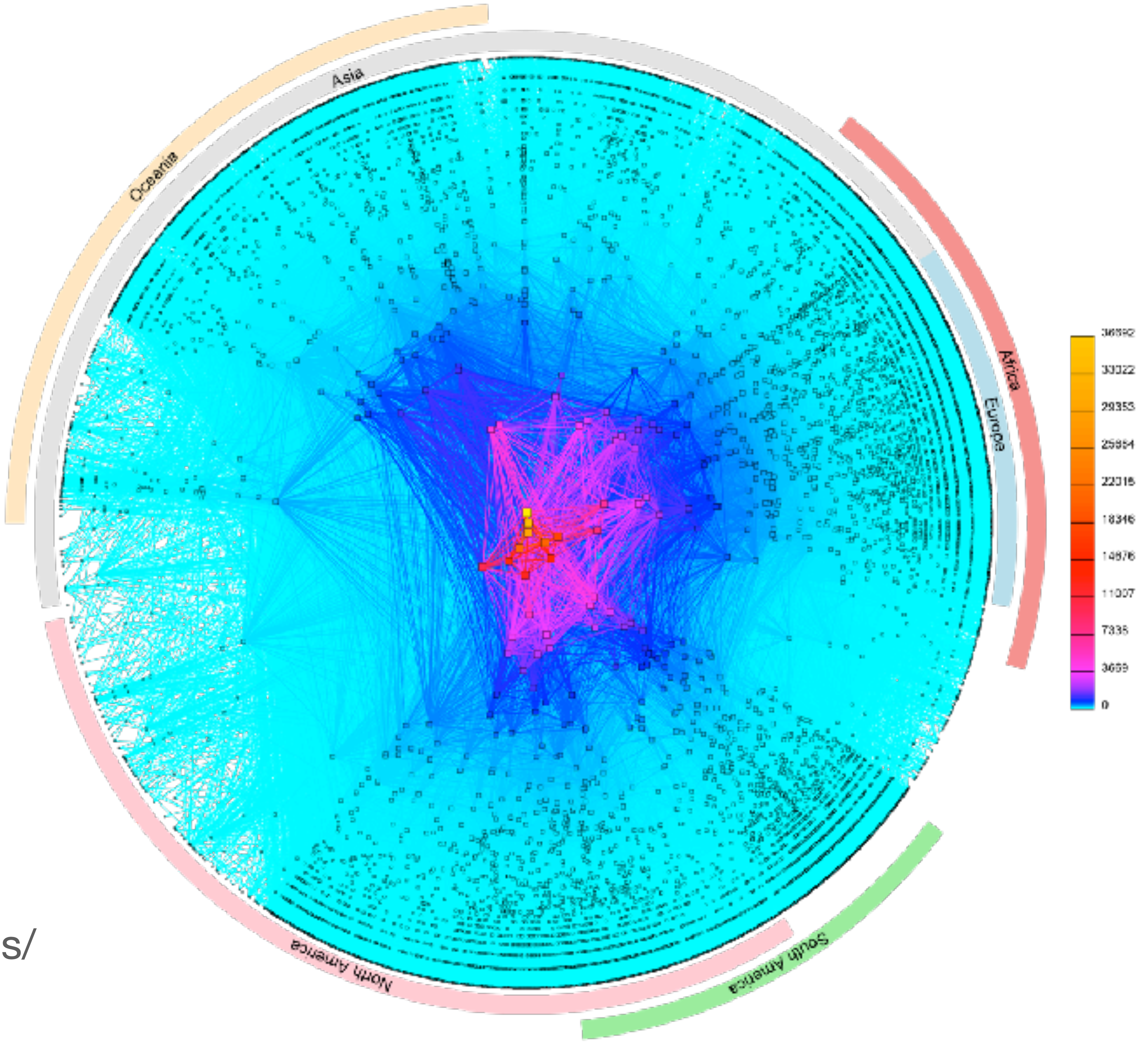
1970s: ARPAnet — hosts.txt

1978: flexibility and layering — distance-vector

early 80s: growth → change — **TCP**, UDP — OSPF, EGP, DNS

late 80s: growth → problems — congestion collapse (which led to **congestion control**) — policy routing — CIDR

1993: commercialization



CAIDA's IPv4 AS Core, January 2020 (https://www.caida.org/projects/cartography/as-core/2020/)

`application`	`the things that actually generate traffic`
`transport`	`sharing the network, reliability (or not)` *examples: TCP, UDP*
`network`	`naming, addressing, routing` *examples: IP*
`link`	`communication between two directly-connected nodes` *examples: ethernet, bluetooth, 802.11 (wifi)*

next time: TCP congestion control doesn't react to congestion until after it's a problem; could we get senders to react before queues are full?