Communication Networks

Spring 2022

Laurent Vanbever

nsg.ee.ethz.ch

ETH Zürich (D-ITET)

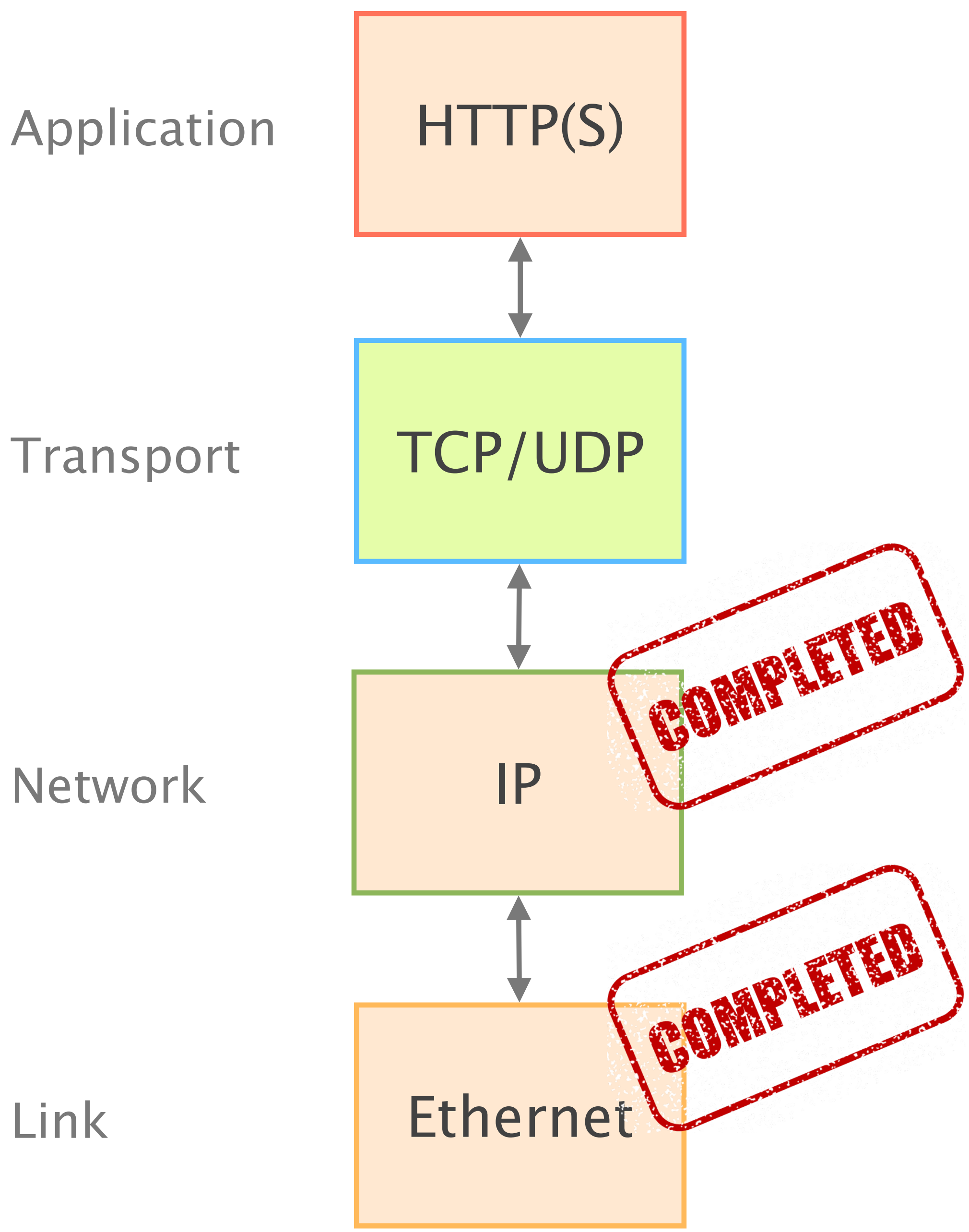
May 9 2022

Materials inspired from Scott Shenker & Jennifer Rexford

Last week on

Communication Networks

We continued our journey up the layers,
and started to look at the transport layer



What Problems Should Be Solved Here?

Data delivering, to the *correct* application

- IP just points towards next protocol
- *Transport needs to demultiplex incoming data (ports)*

Files or bytestreams abstractions for the applications

- Network deals with packets
- *Transport layer needs to translate between them*

Reliable transfer (if needed)

Not overloading the receiver

Not overloading the network

UDP: Datagram messaging service

UDP provides a connectionless, unreliable transport service

- No-frills extension of “best-effort” IP
- UDP provides only two services to the App layer
  - Multiplexing/Demultiplexing among processes
  - Discarding corrupted packets (optional)

TCP: Reliable, in-order delivery

TCP provides a connection-oriented, reliable, bytestream transport service

What UDP provides, plus:

- Retransmission of lost and corrupted packets
- Flow control (to not overflow receiver)
- Congestion control (to not overload network)
- “Connection” set-up & tear-down

Sockets

A socket is a software abstraction by which an application process exchanges network messages with the (transport layer in the) operating system

- socketID = socket(…, socket.TYPE)
- socketID.sendto(message, …)
- socketID.recvfrom(…)

Two important types of sockets

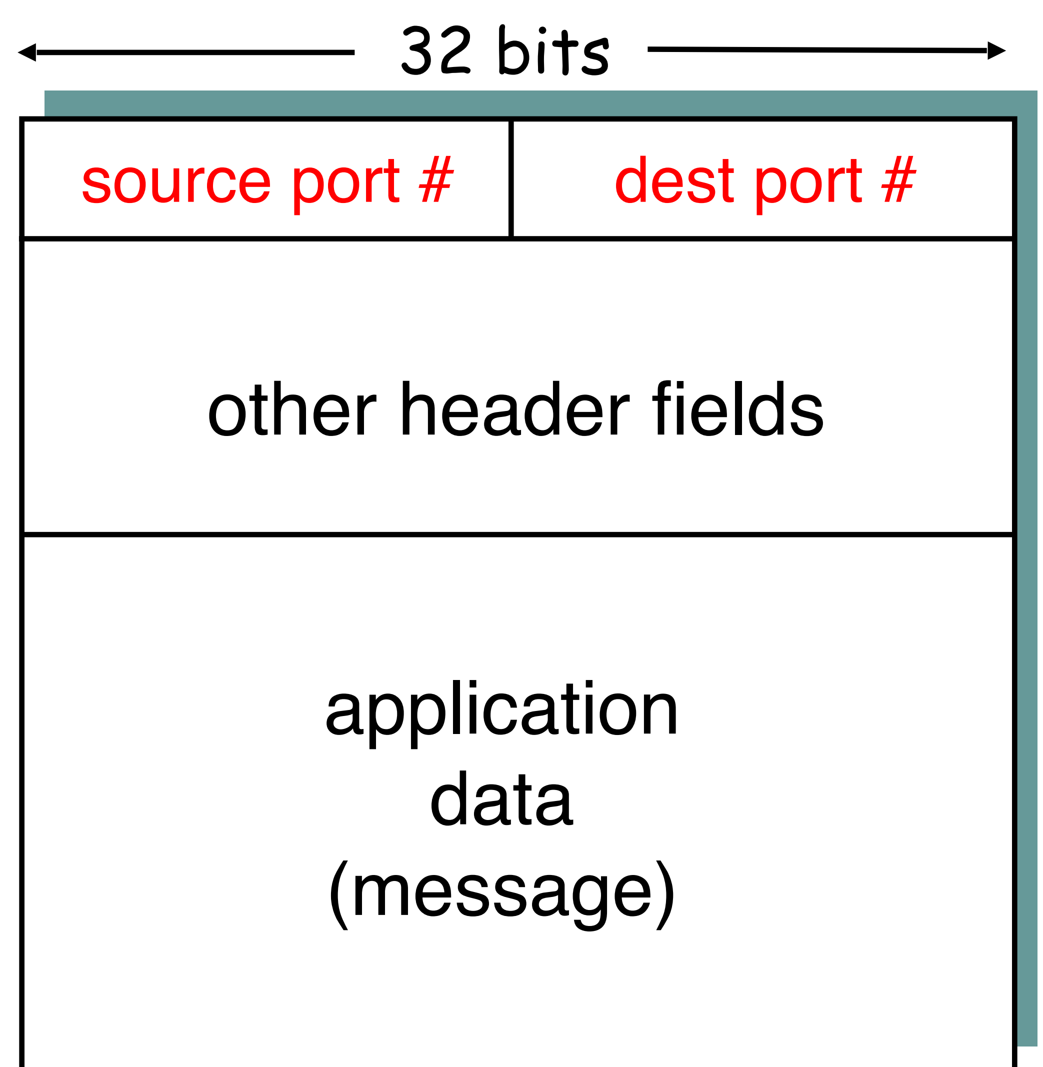
- UDP socket: TYPE is SOCK_DGRAM
- TCP socket: TYPE is SOCK_STREAM

Multiplexing and Demultiplexing

Host receives IP datagrams

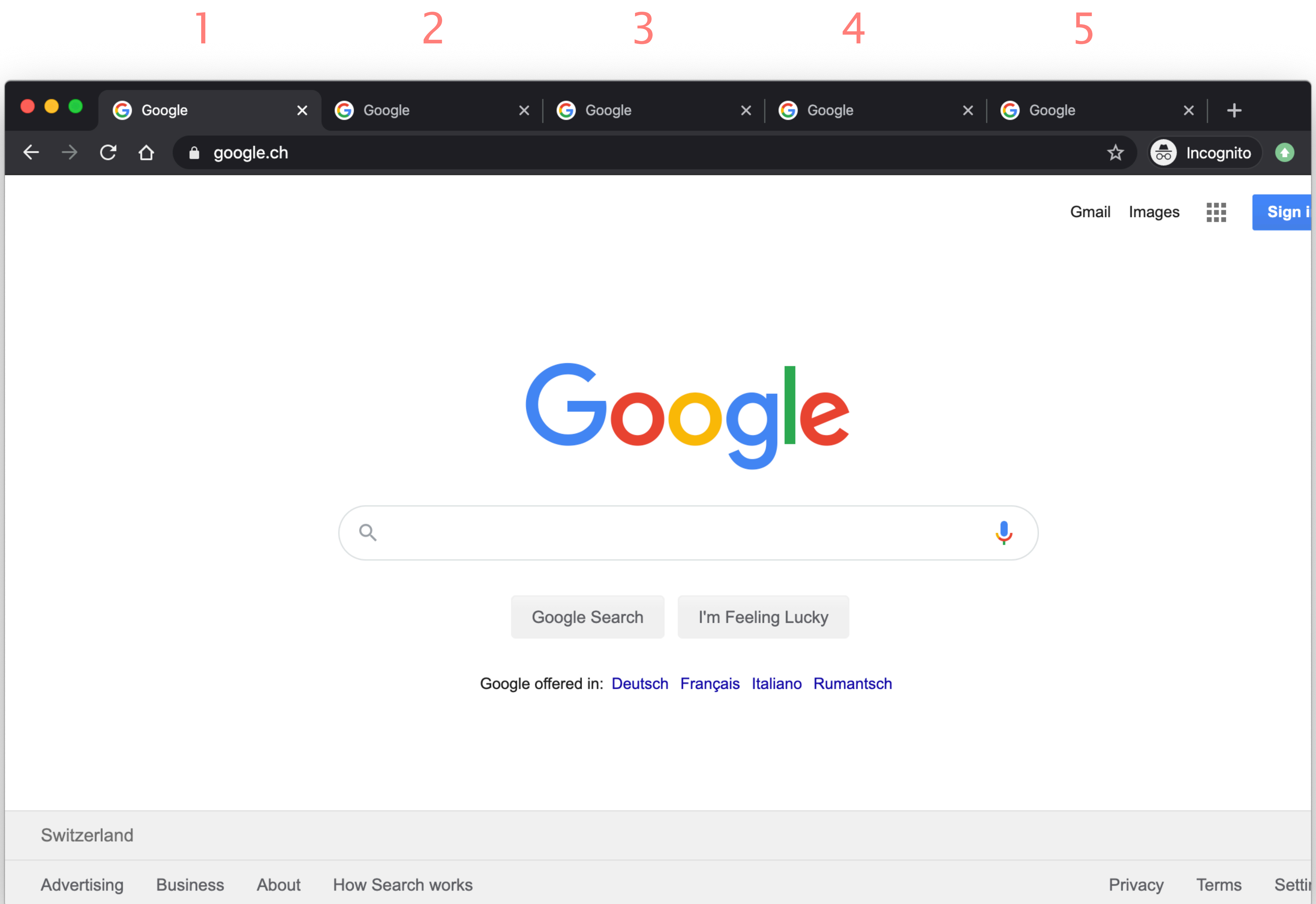
- Each datagram has source and destination IP address,
- Each segment has source and destination port number

Host uses IP addresses *and* port numbers to direct the segment to appropriate socket



A TCP/UDP socket is identified by a 4-tuple:

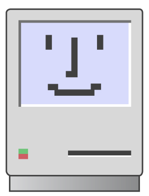
(src IP, src port, dst IP, dest port)



Let's say you open 5 tabs to google.ch

Your IP: 129.132.19.1 Google's IP: 172.217.168.3

Client OS		src IP	src port	dest IP	dest port
socket	1	129.132.19.1	54001	172.217.168.3	443
	2	129.132.19.1	55240	172.217.168.3	443
	3	129.132.19.1	48472	172.217.168.3	443
	4	129.132.19.1	35456	172.217.168.3	443
	5	129.132.19.1	42001	172.217.168.3	443



Server OS		src IP	src port	dest IP	dest port
socket	1	172.217.168.3	443	129.132.19.1	54001
	2	172.217.168.3	443	129.132.19.1	55240
	3	172.217.168.3	443	129.132.19.1	48472
	4	172.217.168.3	443	129.132.19.1	35456
	5	172.217.168.3	443	129.132.19.1	42001

This week on

Communication Networks

UDP / TCP

Congestion Control

finishing off
last week's slides

UDP / TCP

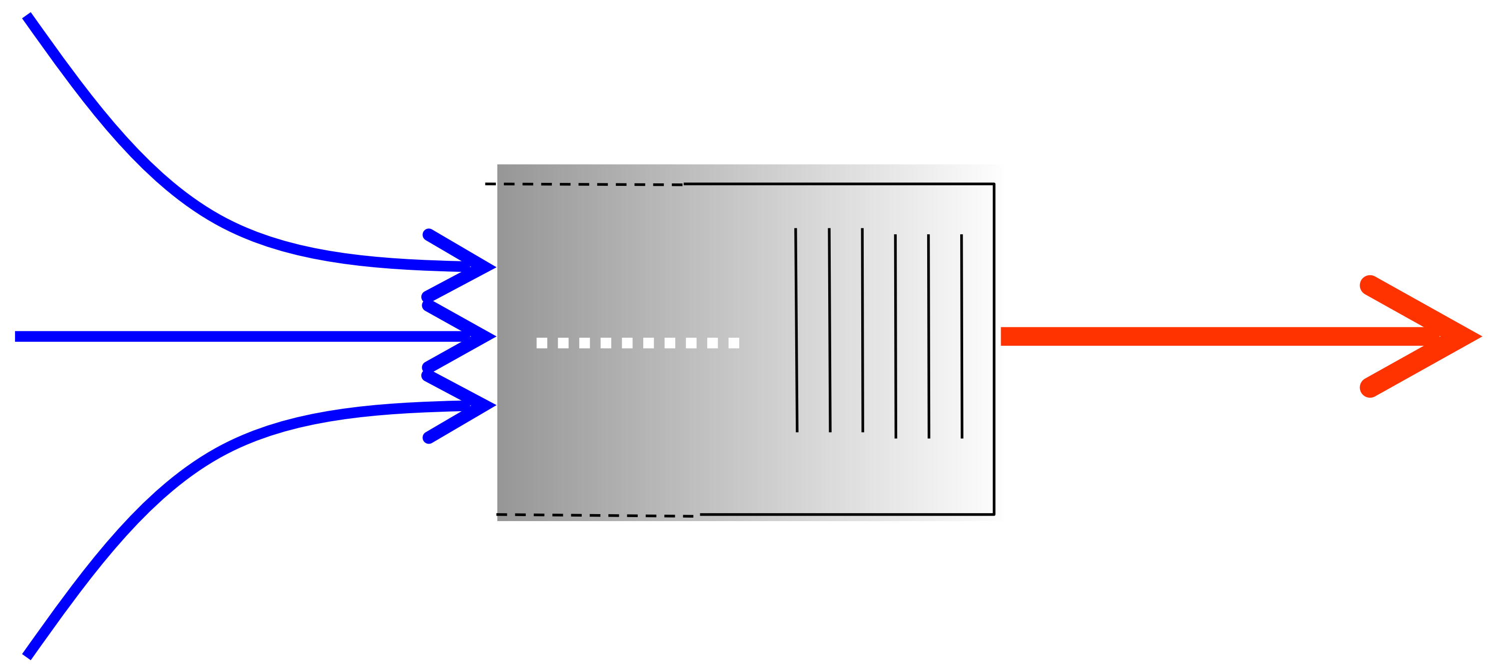
Congestion Control

finishing off
last week's slides

UDP / TCP

Congestion Control

Because of traffic burstiness and lack of BW reservation,
congestion is inevitable



If many packets arrive within
a short period of time
the node cannot keep up anymore

Congestion is harmful

average packet arrival rate	a	[packet/sec]
transmission rate of outgoing link	R	[bit/sec]
fixed packets length	L	[bit/packet]
average bits arrival rate	La	[bit/sec]
traffic intensity	La/R	

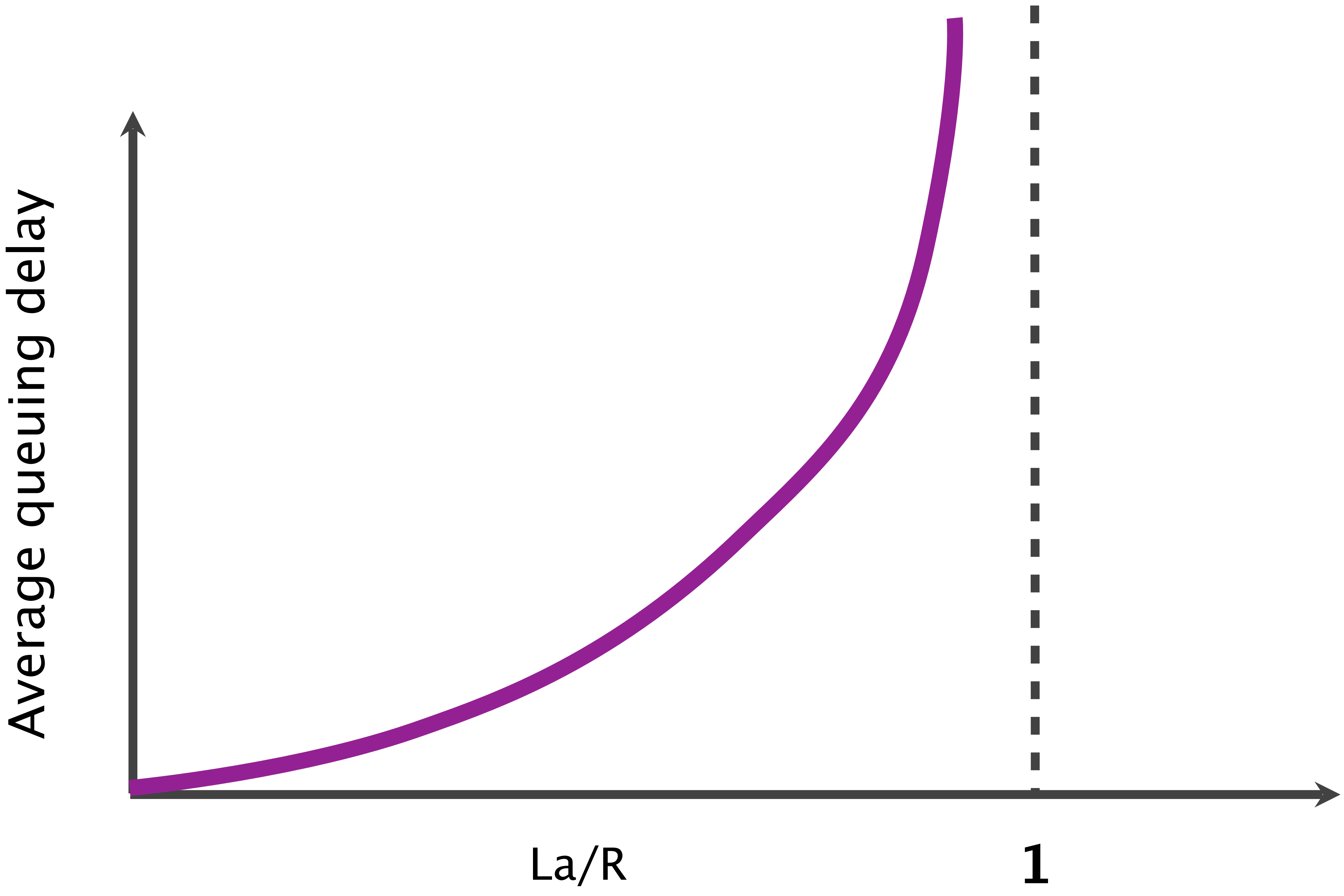
When the traffic intensity is >1, the queue will increase without bound, and so does the queuing delay

Golden rule

Design your queuing system,

so that it operates far from that point

When the traffic intensity is <=1,
queueing delay depends on the burst size



Congestion is not a new problem

The Internet almost died of congestion in 1986

throughput collapsed from 32 Kbps to… 40 bps

Van Jacobson saved us with Congestion Control

his solution went right into BSD

Recent resurgence of research interest after brief lag

new methods (ML), context (Data centers), requirements

The Internet almost died of congestion in 1986

throughput collapsed from 32 Kbps to… 40 bps

original behavior	On connection, nodes send full window of packets
	Upon timer expiration, retransmit packet immediately
meaning	sending rate only limited by flow control
net effect	window-sized burst of packets

Increase in network load results in a decrease of useful work done

Sudden load increased the round-trip time (RTT)
faster than the hosts' measurements of it

As RTT exceeds the maximum retransmission interval,
hosts begin to retransmit packets

Hosts are sending each packet several times,
eventually some copies arrive at the destination.

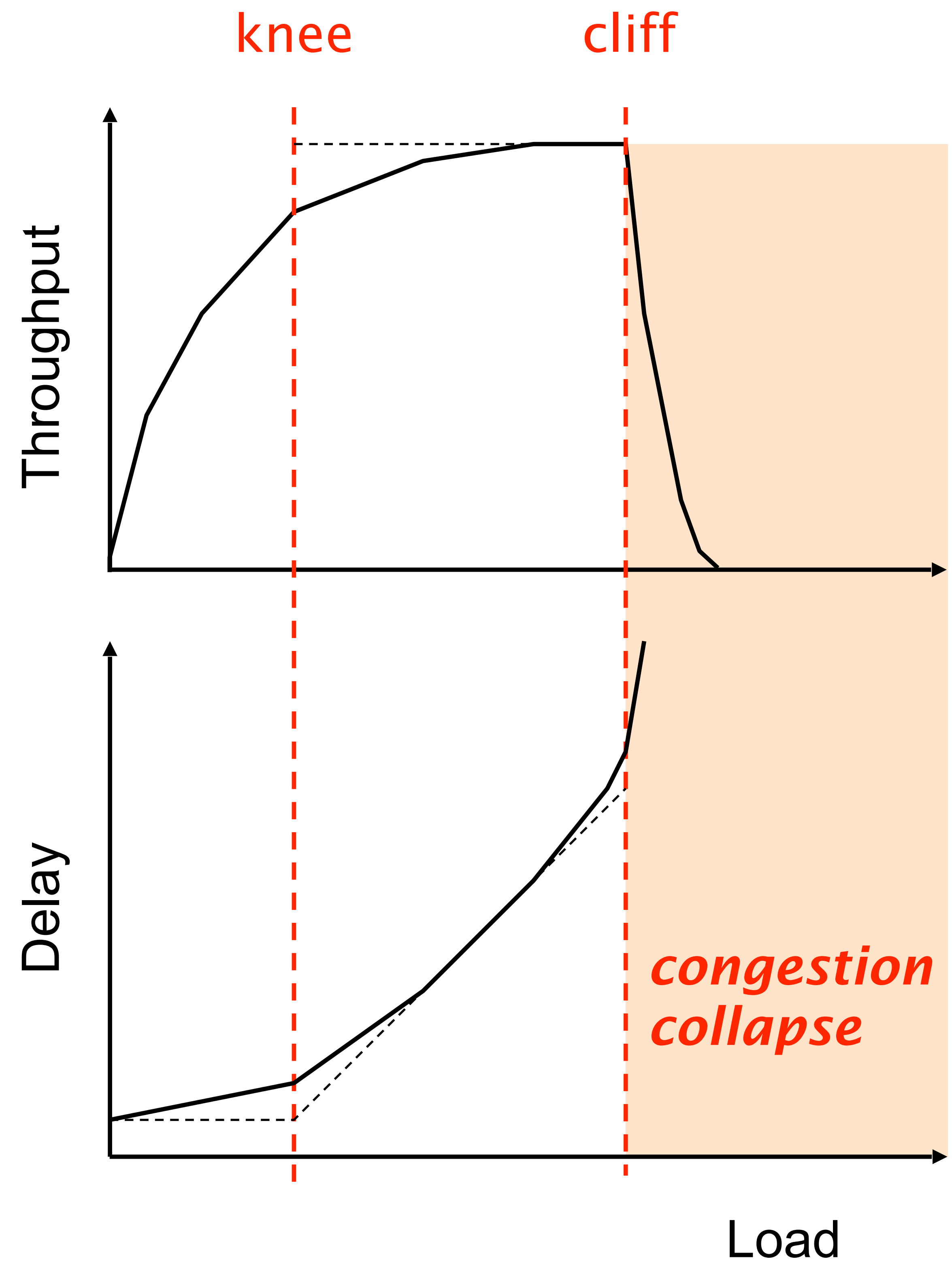
This phenomenon is known as congestion collapse

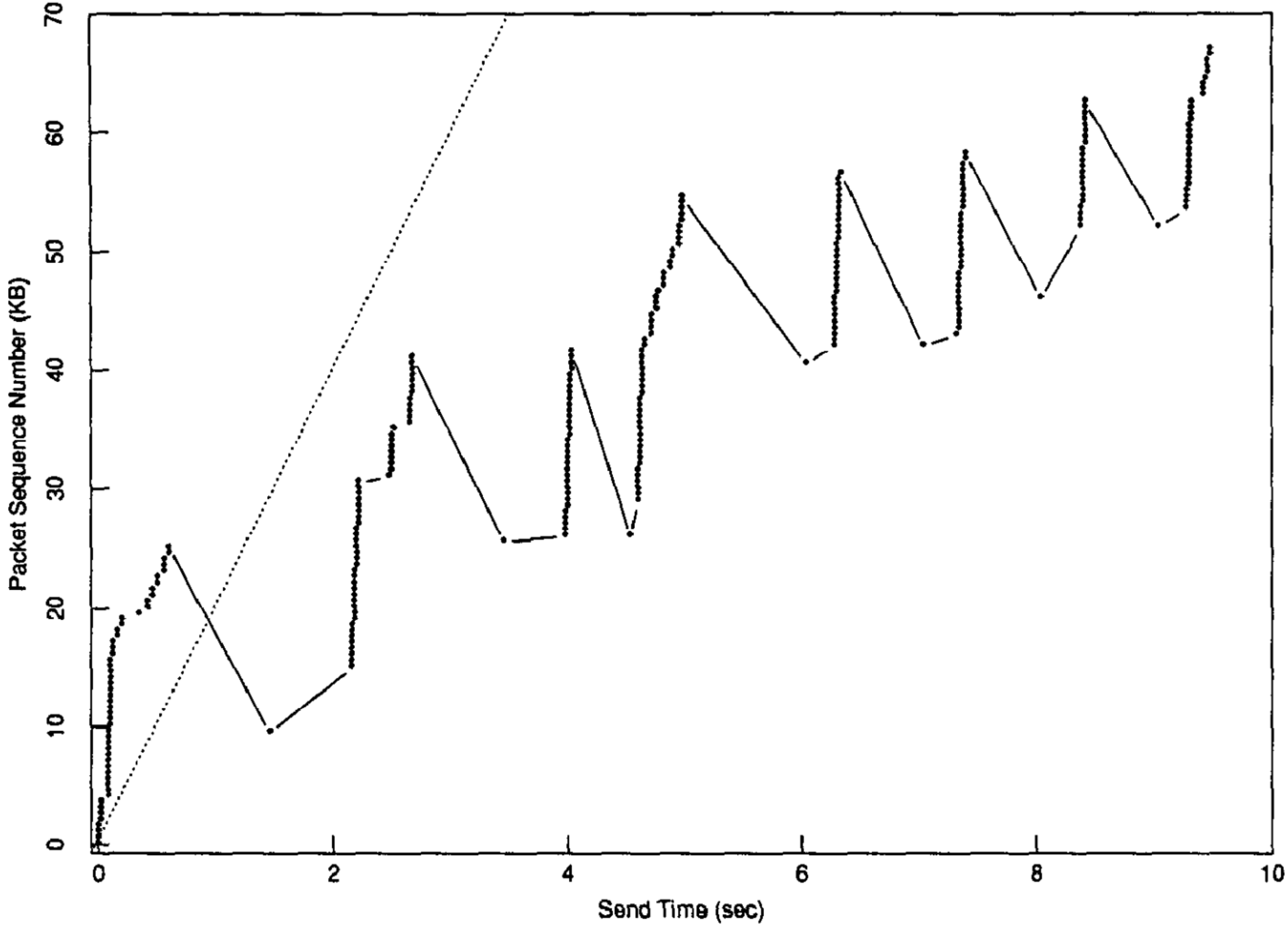
Knee point after which

throughput	increases	slowly
delay	increases	quickly

Cliff point after which

throughput	decreases	quickly
delay	tends to	infinity





Packet Sequence Number (KB)
0
10
20
30
40
50
60
70
0
2
4
6
8
10
Send Time (sec)

Van Jacobson saved us with Congestion Control

his solution went right into BSD

Congestion control aims at solving three problems

#1	bandwidth **estimation**	How to adjust the bandwidth of a single flow to the bottleneck bandwidth? could be 1 Mbps or 1 Gbps…
#2	bandwidth **adaptation**	How to adjust the bandwidth of a single flow to variation of the bottleneck bandwidth?
#3	**fairness**	How to share bandwidth “fairly" among flows, without overloading the network

Congestion control differs from flow control

both are provided by TCP though

Flow control	prevents one fast sender from overloading a slow receiver
Congestion control	prevents a set of senders from overloading the network

TCP solves both using two distinct windows

Flow control	prevents one fast sender from overloading a slow receiver solved using a **receiving window**
Congestion control	prevents a set of senders from overloading the network solved using a **"congestion" window**

The sender adapts its sending rate based on these two windows

Receiving Window RWND	How many bytes can be sent without overflowing the receiver buffer? based on the receiver input
Congestion Window CWND	How many bytes can be sent without overflowing the routers? based on network conditions
Sender Window	minimum(CWND, RWND)

The 2 key mechanisms of Congestion Control

detecting
congestion

reacting to
congestion

The 2 key mechanisms of Congestion Control

detecting

congestion

reacting to

congestion

There are essentially three ways to detect congestion

Approach #1	Network could tell the source but signal itself could be lost
Approach #2	Measure packet delay but signal is noisy delay often varies considerably
Approach #3	Measure packet loss fail-safe signal that TCP already has to detect

Packet dropping is the best solution

delay- and signaling-based methods are hard & risky

Approach #3

Measure packet loss

fail-safe signal that TCP already has to detect

Detecting losses can be done using ACKs or timeouts, the two signal differ in their degree of severity

duplicated ACKs	mild congestion signal packets are still making it
timeout	severe congestion signal multiple consequent losses

The 2 key mechanisms of Congestion Control

detecting

congestion

reacting to

congestion

question

What increase/decrease function should we use?

it depends on the problem we are solving...

Remember that Congestion Control aims at solving three problems

#1	bandwidth estimation	How to adjust the bandwidth of a single flow to the bottleneck bandwidth? could be 1 Mbps or 1 Gbps…
#2	bandwidth adaptation	How to adjust the bandwidth of a single flow to variation of the bottleneck bandwidth?
#3	fairness	How to share bandwidth “fairly" among flows, without overloading the network

#1 bandwidth estimation

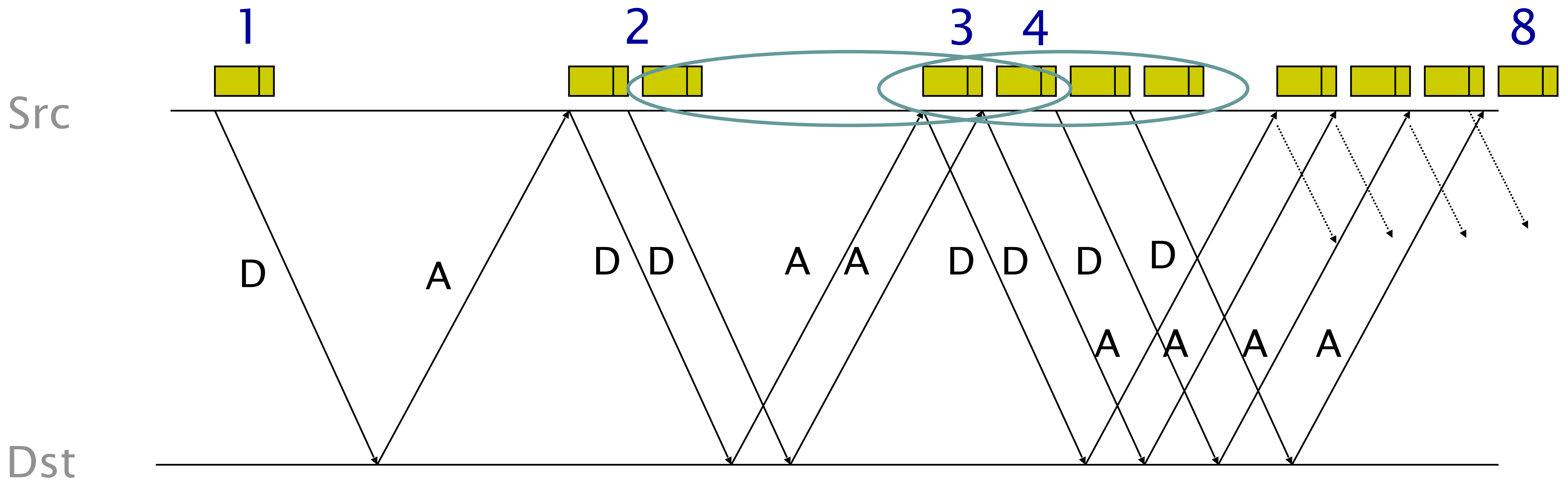
How to adjust the bandwidth of a single flow to the bottleneck bandwidth?

could be 1 Mbps or 1 Gbps…

The goal here is to quickly get a first-order estimate of the available bandwidth

Intuition	Start slow but rapidly increase until a packet drop occurs	
Increase policy	cwnd = 1	initially
	cwnd += 1	upon receipt of an ACK

This increase phase, known as slow start, corresponds to an… exponential increase of CWND!



slow start is called like this only because of starting point

The problem with slow start is that it can result in a full window of packet losses

Example

Assume that CWND is just enough to “fill the pipe”

After one RTT, CWND has doubled

All the excess packets are now dropped

Solution

We need a more gentle adjustment algorithm once we have a rough estimate of the bandwidth

#2	bandwidth adaptation	How to adjust the bandwidth of a single flow to variation of the bottleneck bandwidth?

The goal here is to track the available bandwidth, and oscillate around its current value

Two possible variations

- Multiplicative Increase or Decrease

  cwnd = a * cwnd

- Additive Increase or Decrease

  cwnd = b + cwnd

... leading to four alternative design

	increase behavior	decrease behavior
AIAD	gentle	gentle
AIMD	gentle	aggressive
MIAD	aggressive	gentle
MIMD	aggressive	aggressive

To select one scheme, we need to consider the 3rd problem: fairness

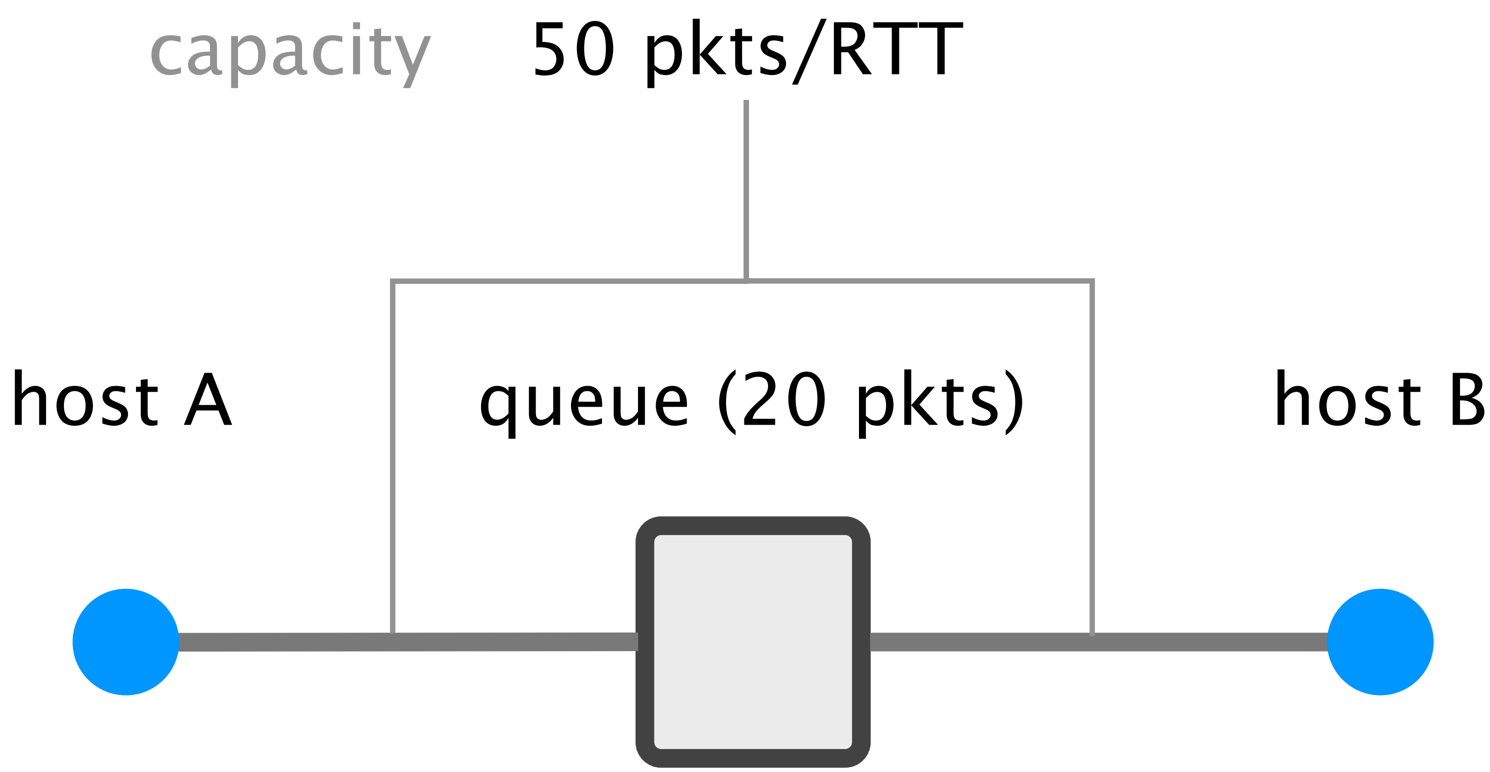
	increase behavior	decrease behavior
AIAD	gentle	gentle
AIMD	gentle	aggressive
MIAD	aggressive	gentle
MIMD	aggressive	aggressive

#3 fairness

How to share bandwidth “fairly" among flows, without overloading the network

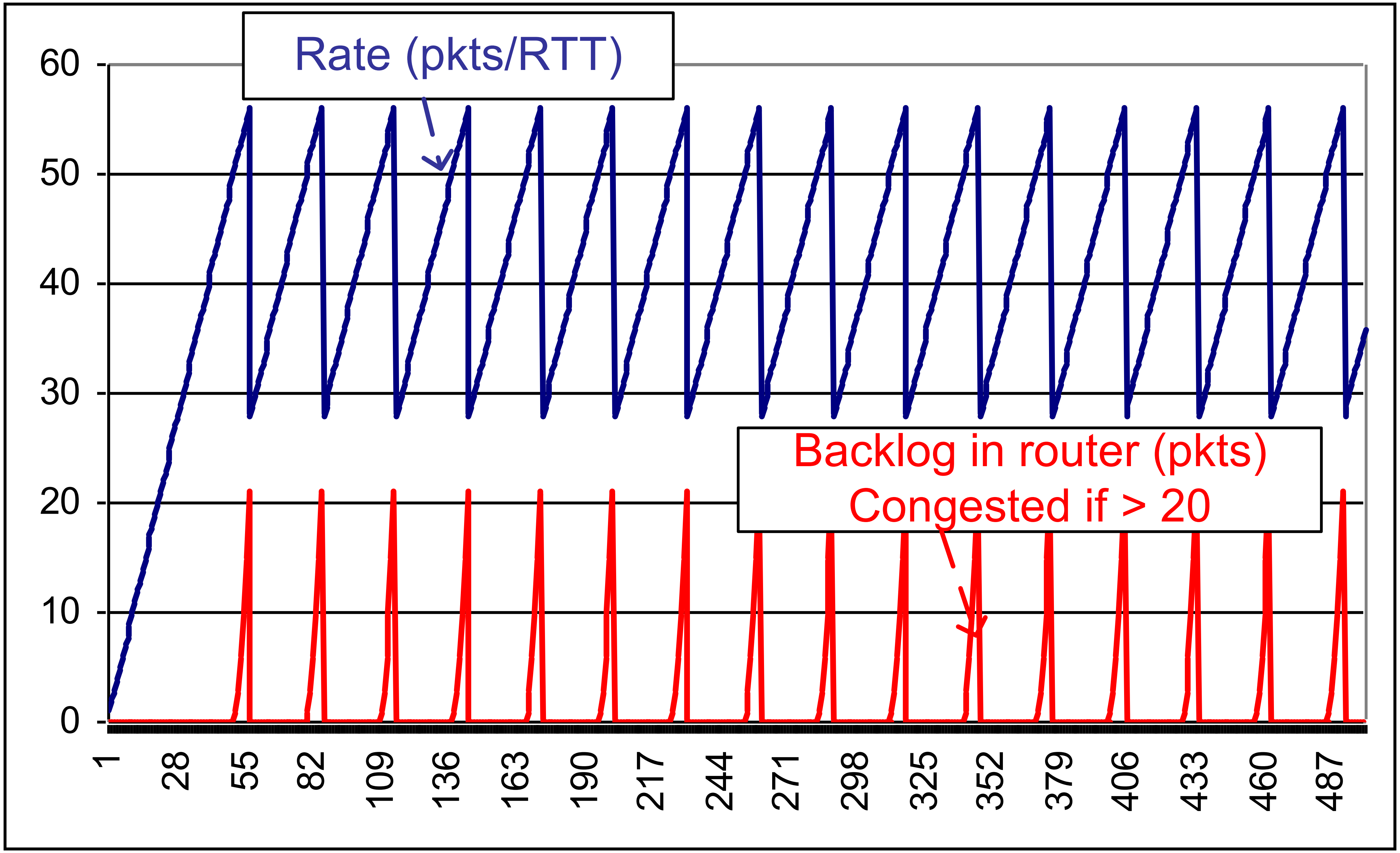
TCP notion of fairness: 2 identical flows should end up with the same bandwidth

Consider first a single flow between A and B and AIMD



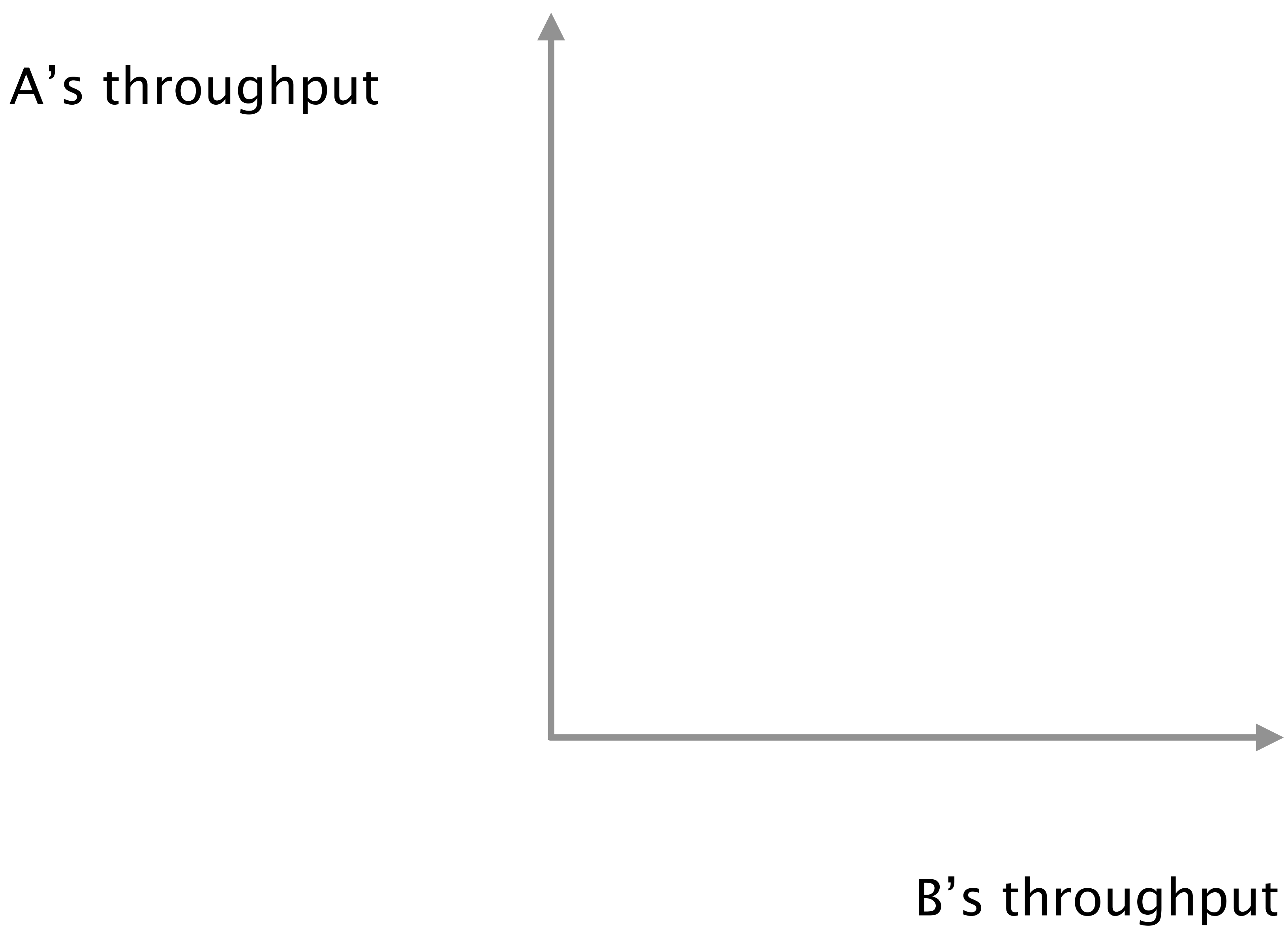
without congestion — CWND increases by one packet every ACK

upon congestion — CWND decreases by a factor 2

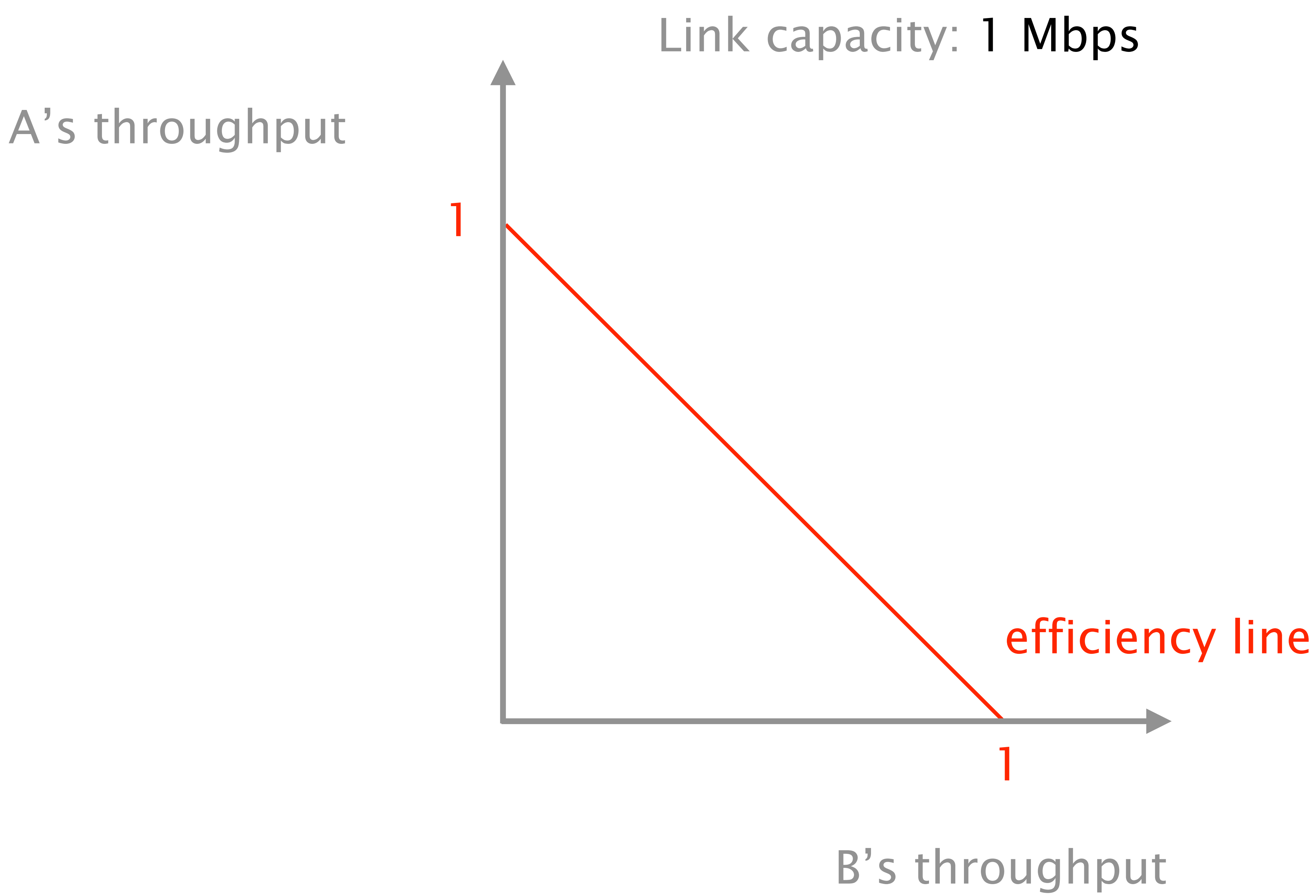


Rate (pkts/RTT)
Backlog in router (pkts)
Congested if > 20
60
50
40
30
20
10
0
1
28
55
82
109
136
163
190
217
244
271
298
325
352
379
406
433
460
487

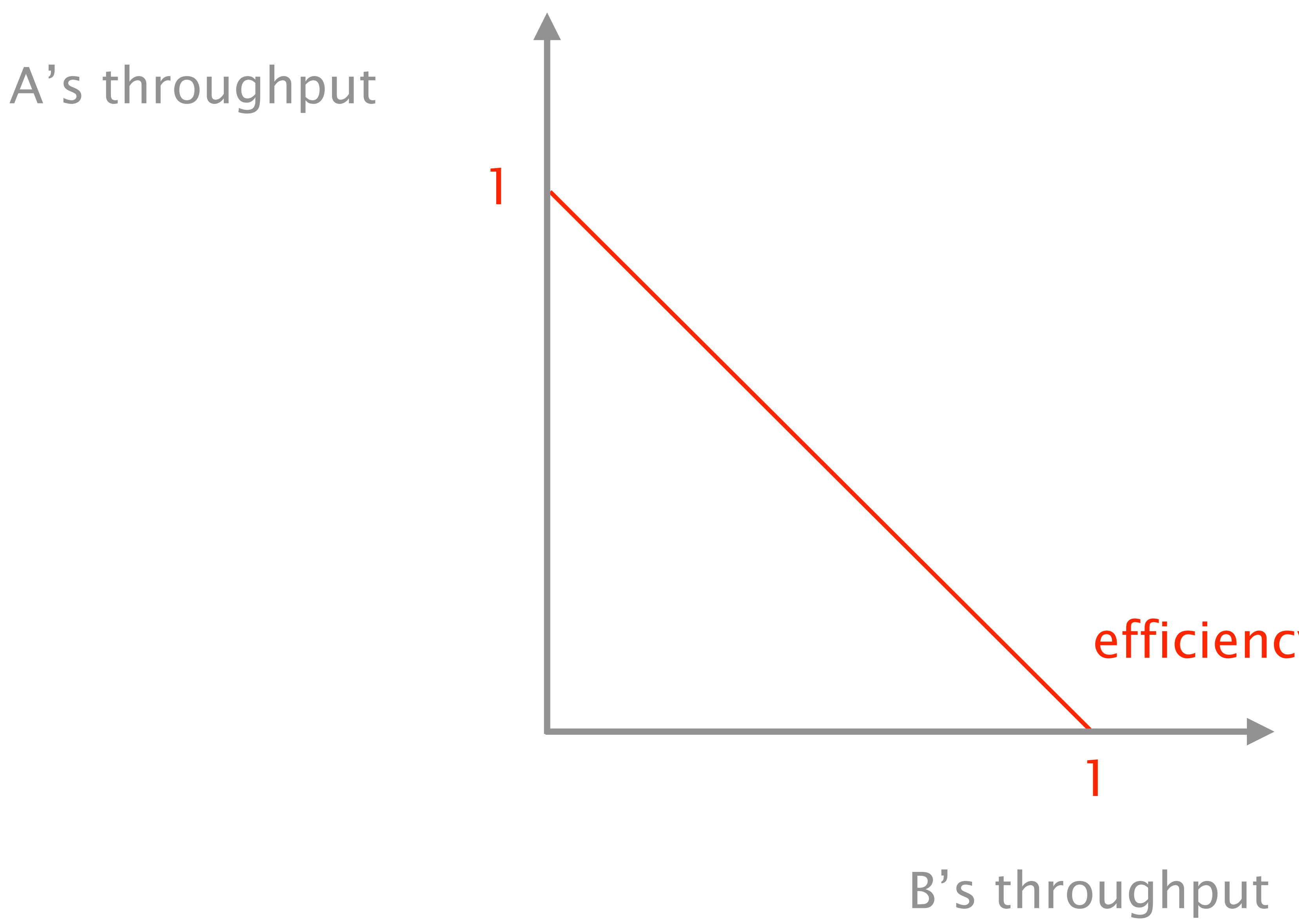
We can analyze the system behavior using a system trajectory plot

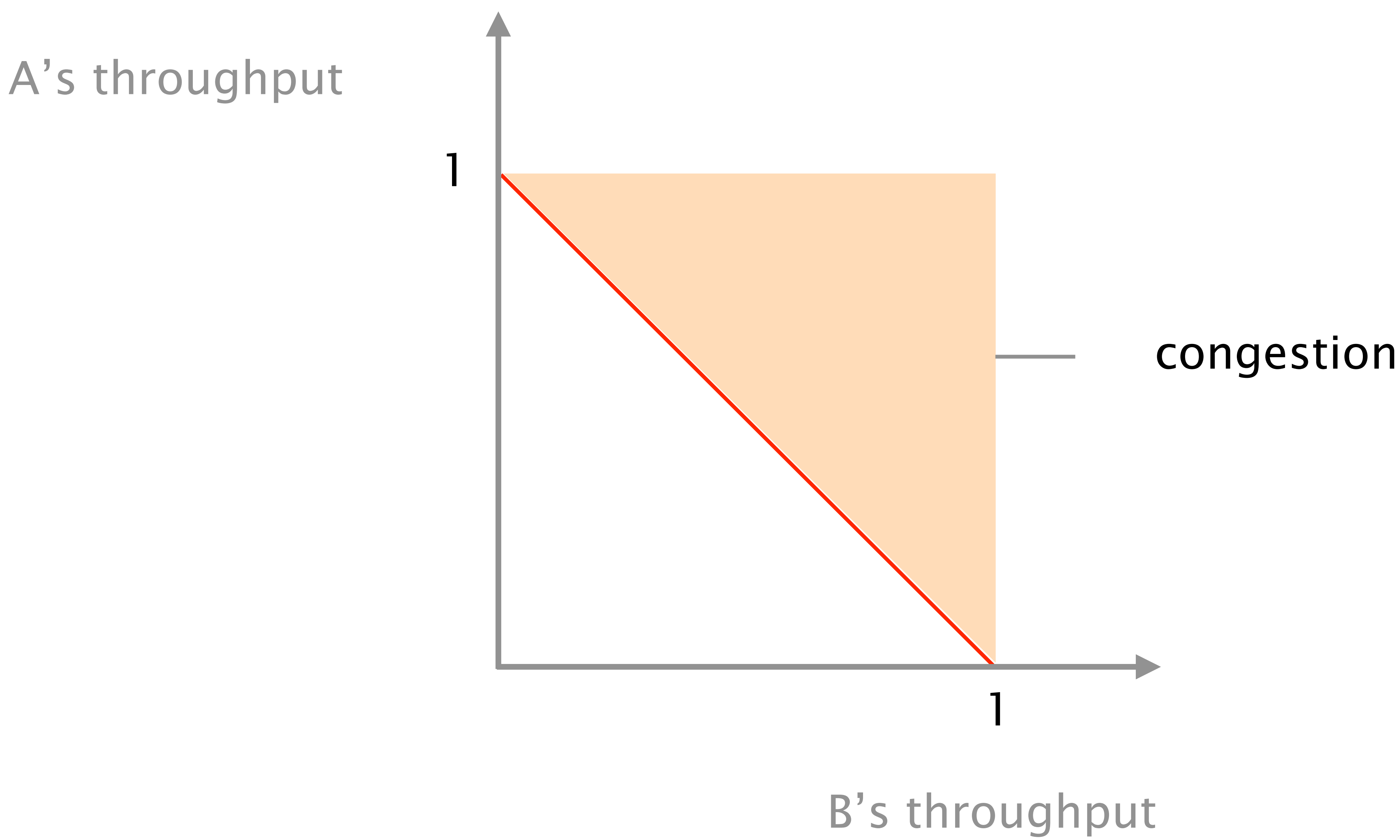


The system is efficient if the capacity is fully used, defining an efficiency line where $a + b = 1$

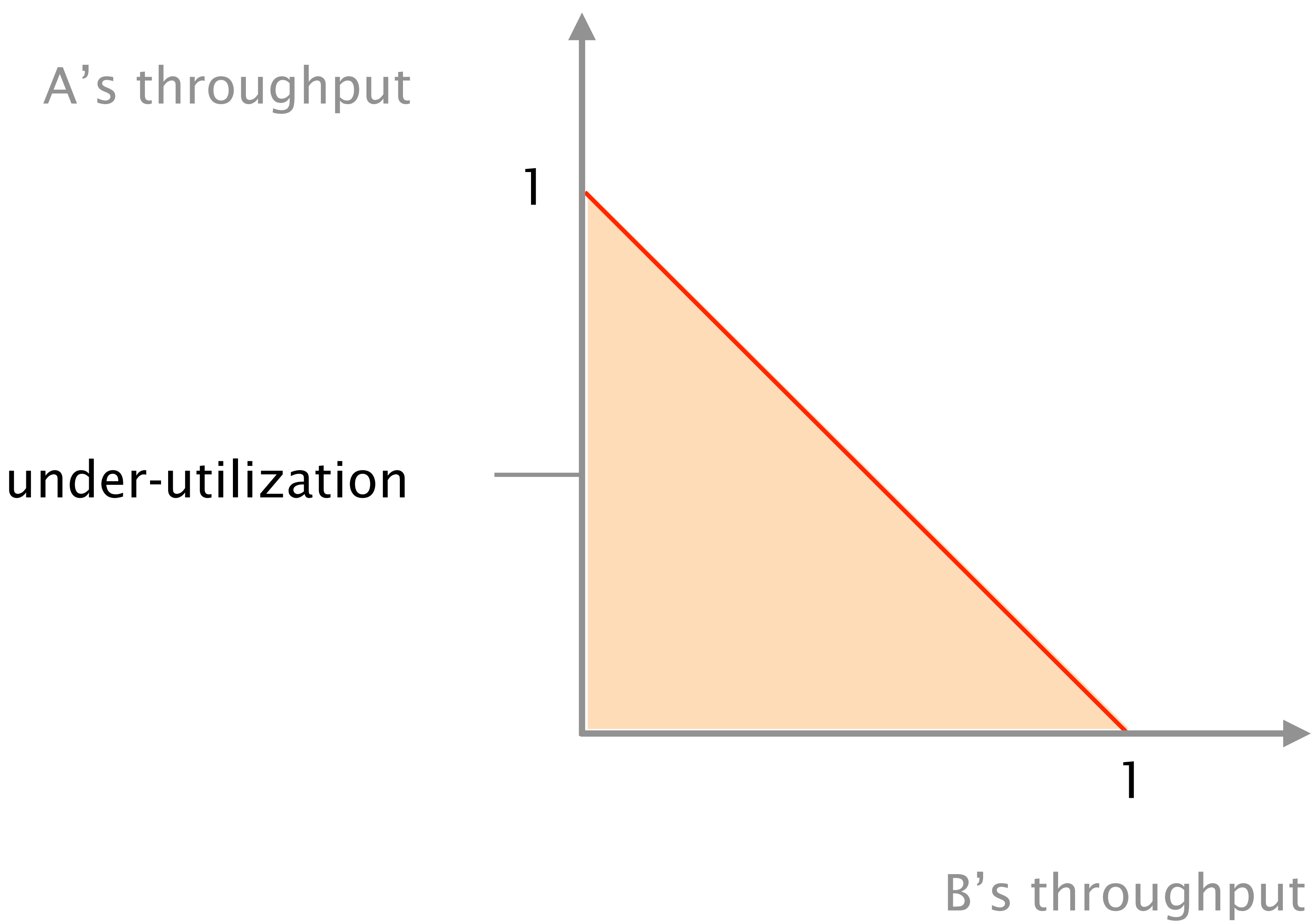


The goal of congestion control is to bring the system as close as possible to this line, and stay there

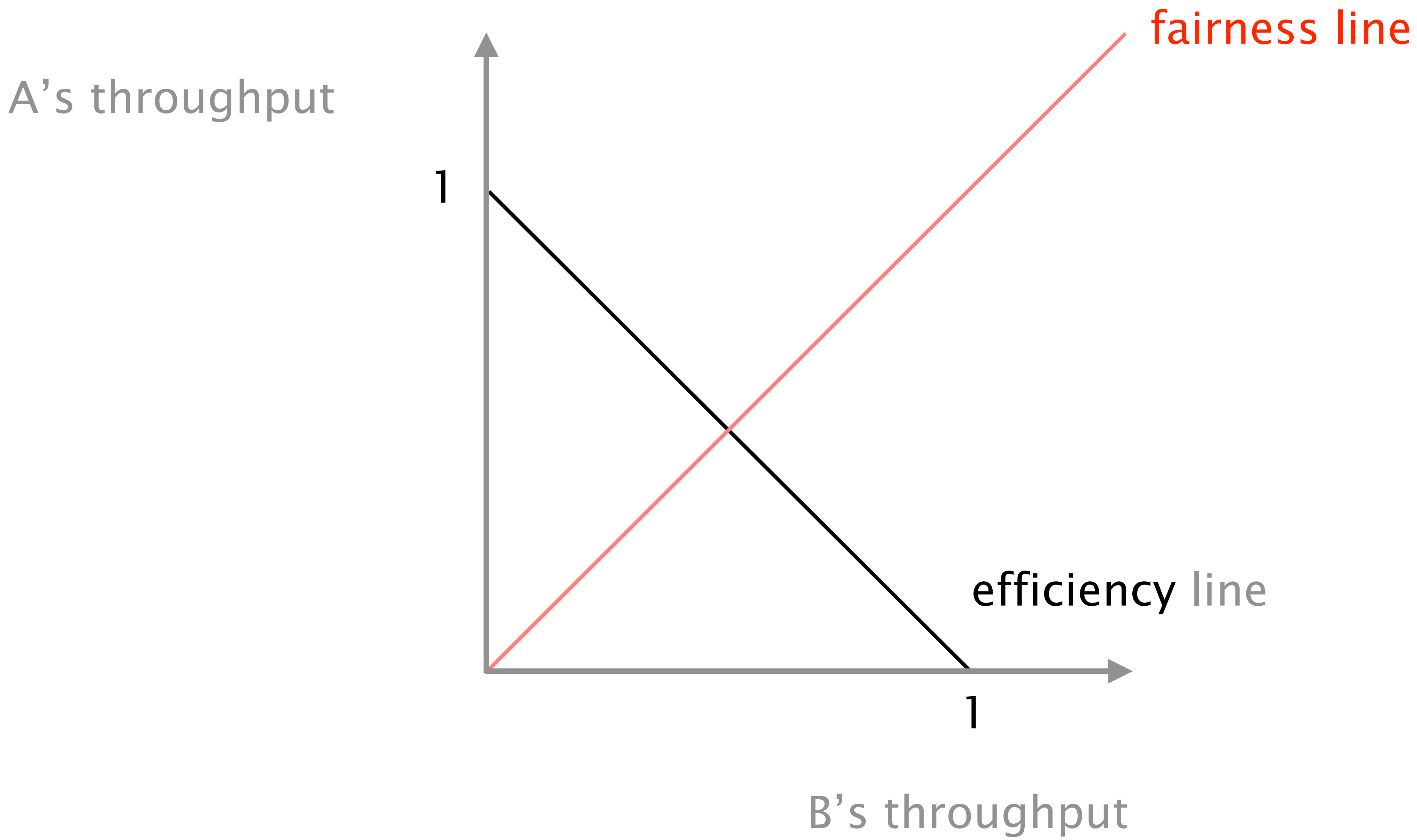


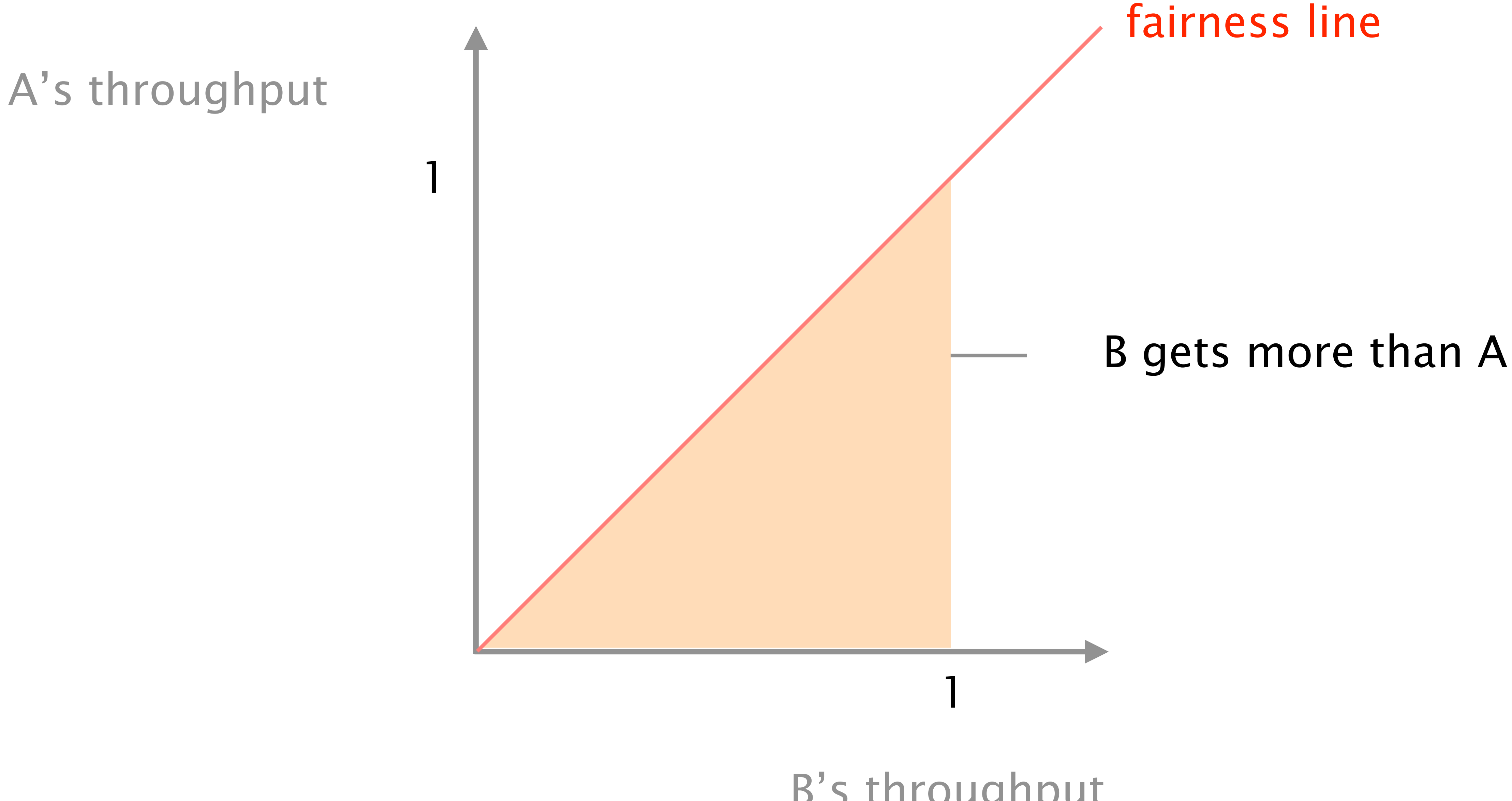


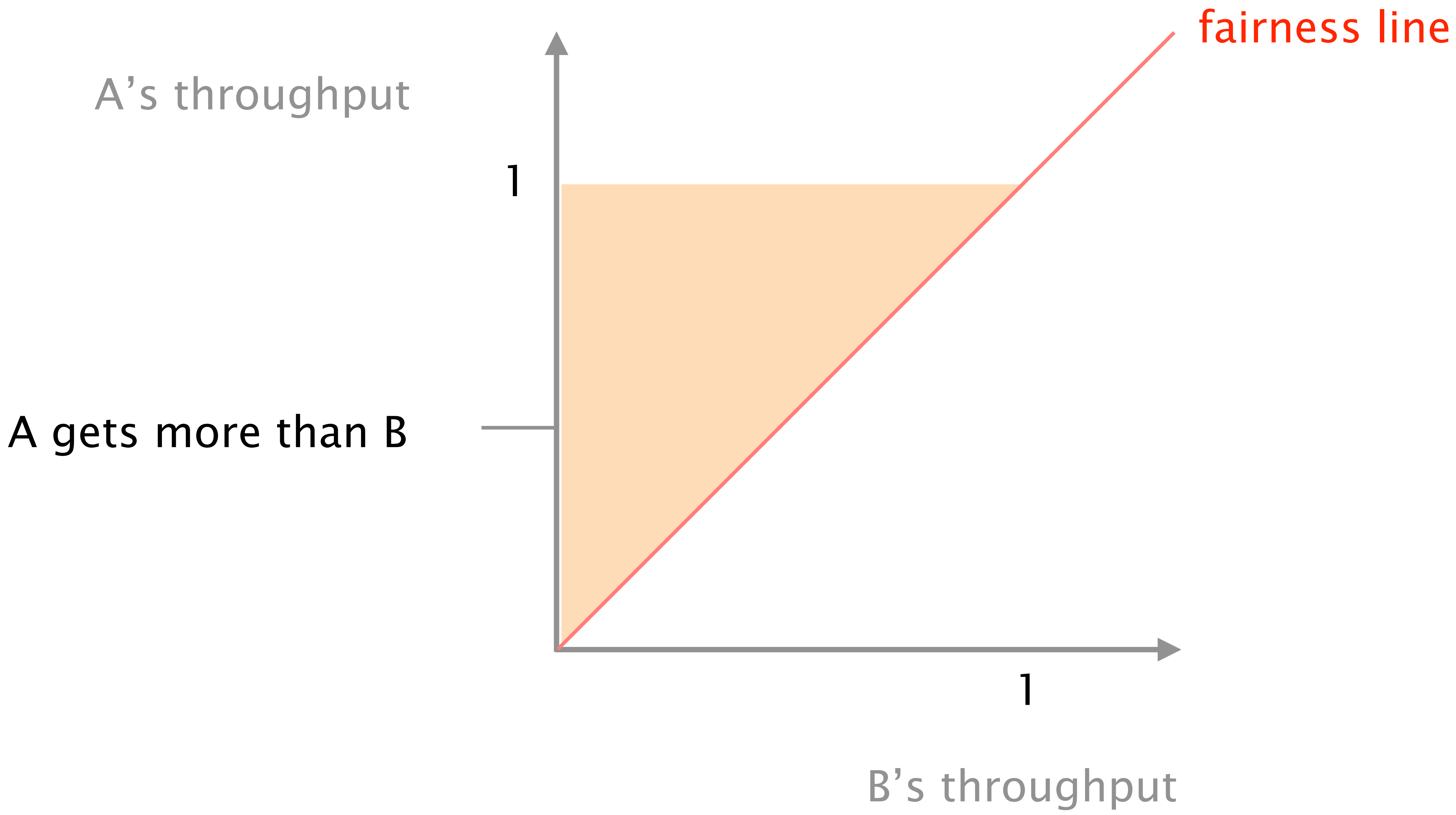
A’s throughput
1
congestion
1
B’s throughput


A’s throughput
1
under-utilization
1
B’s throughput

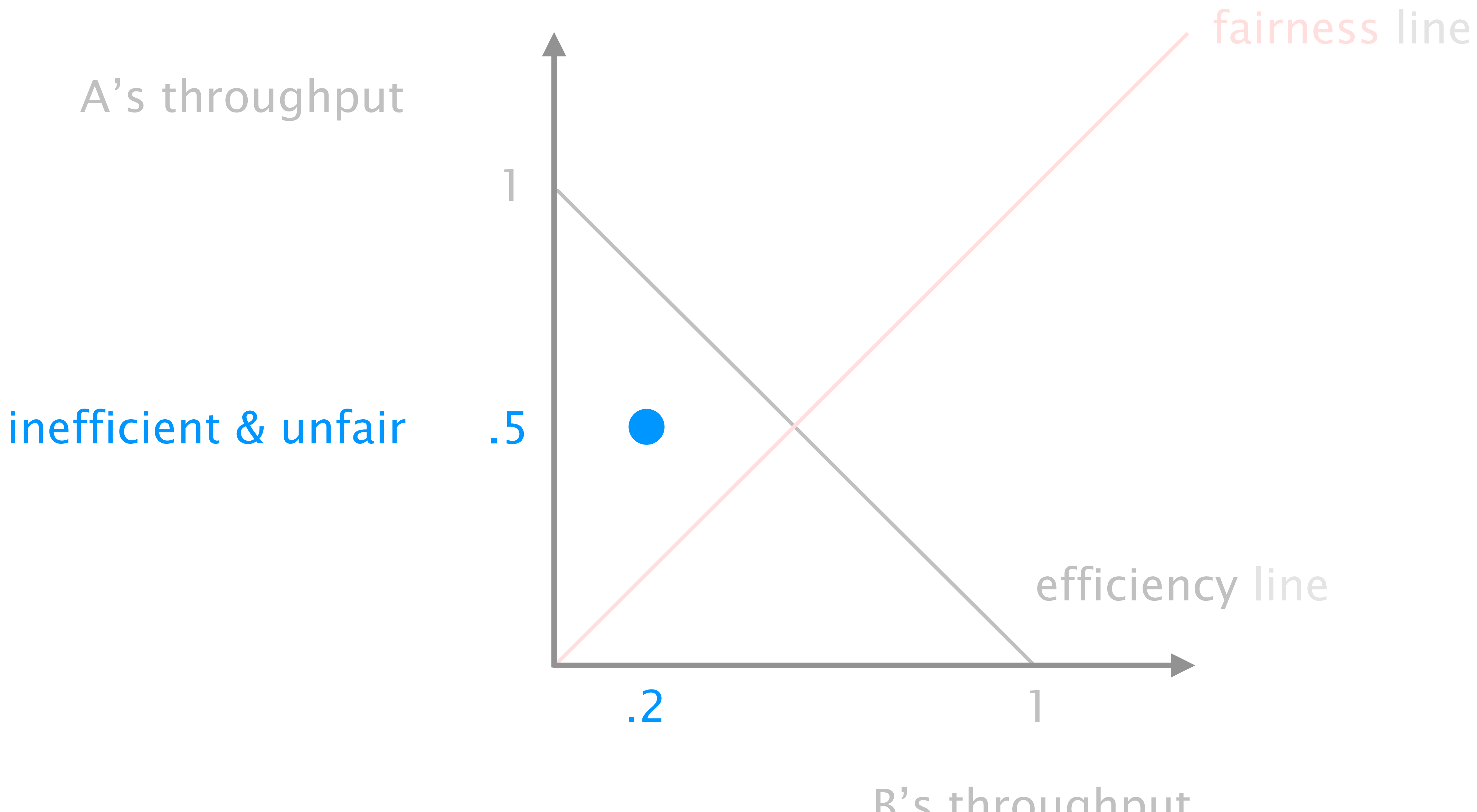
The system is fair whenever A and B have equal throughput, defining a fairness line where $a = b$

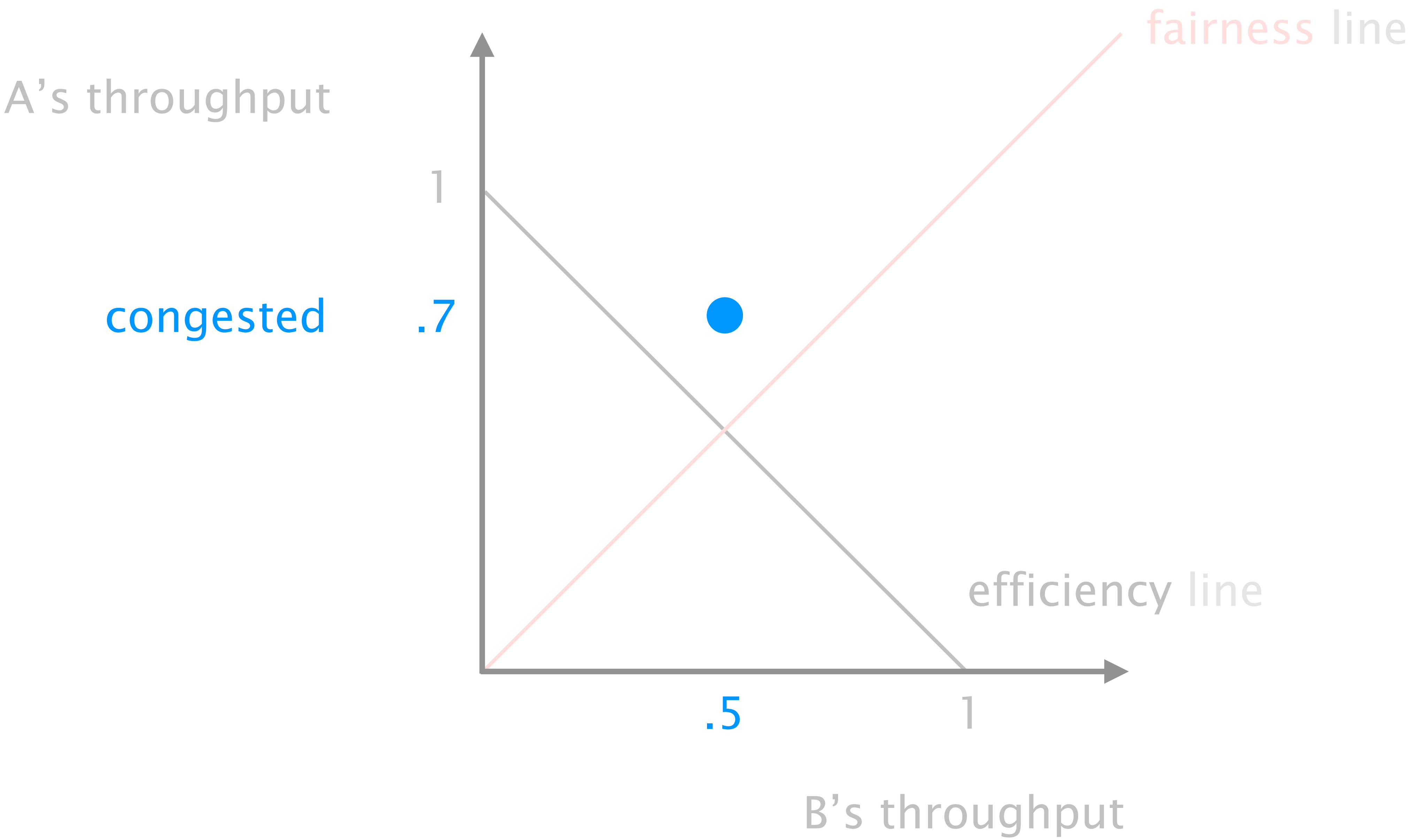


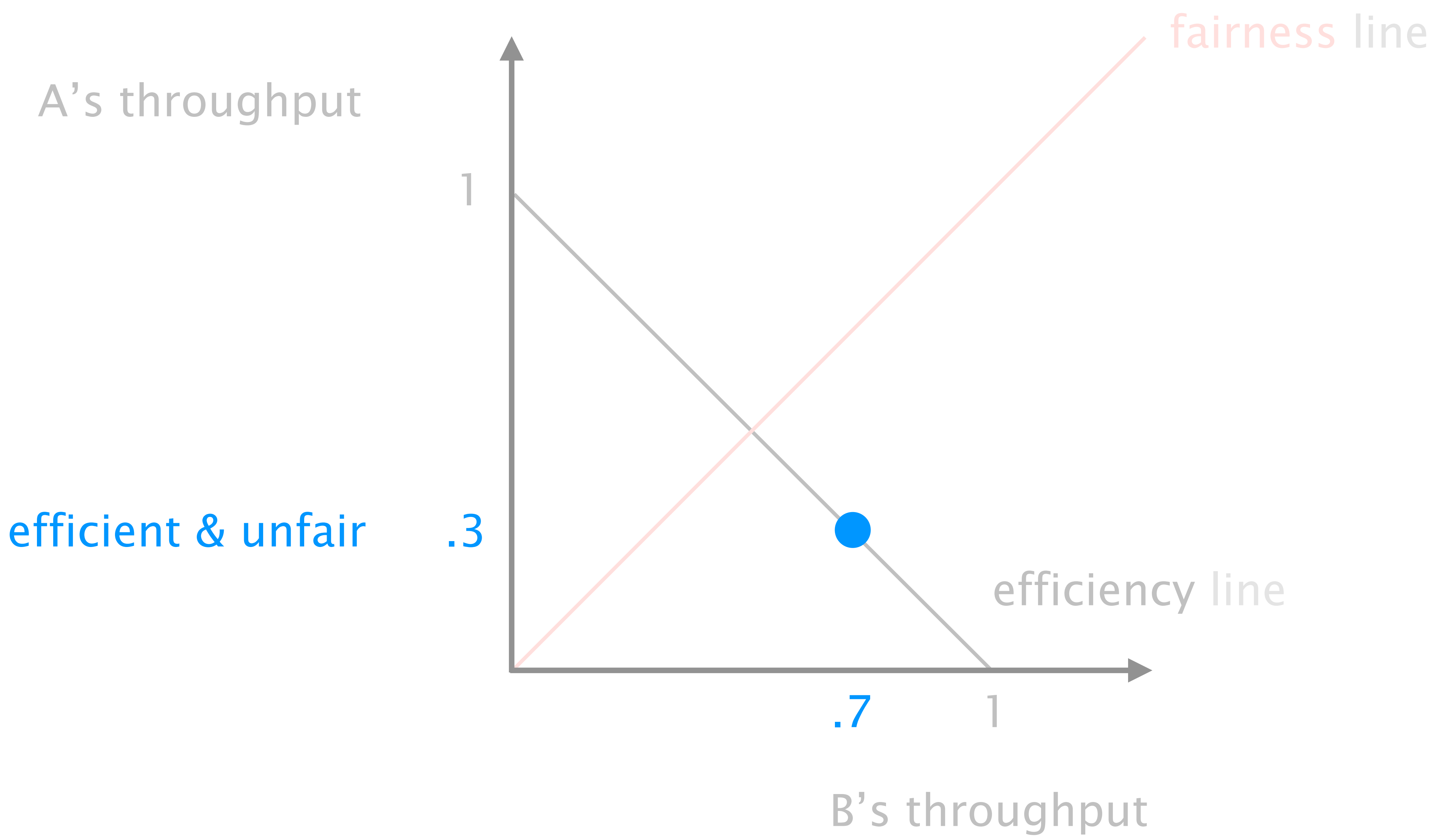

fairness line
A's throughput
1
B gets more than A
1
B's throughput



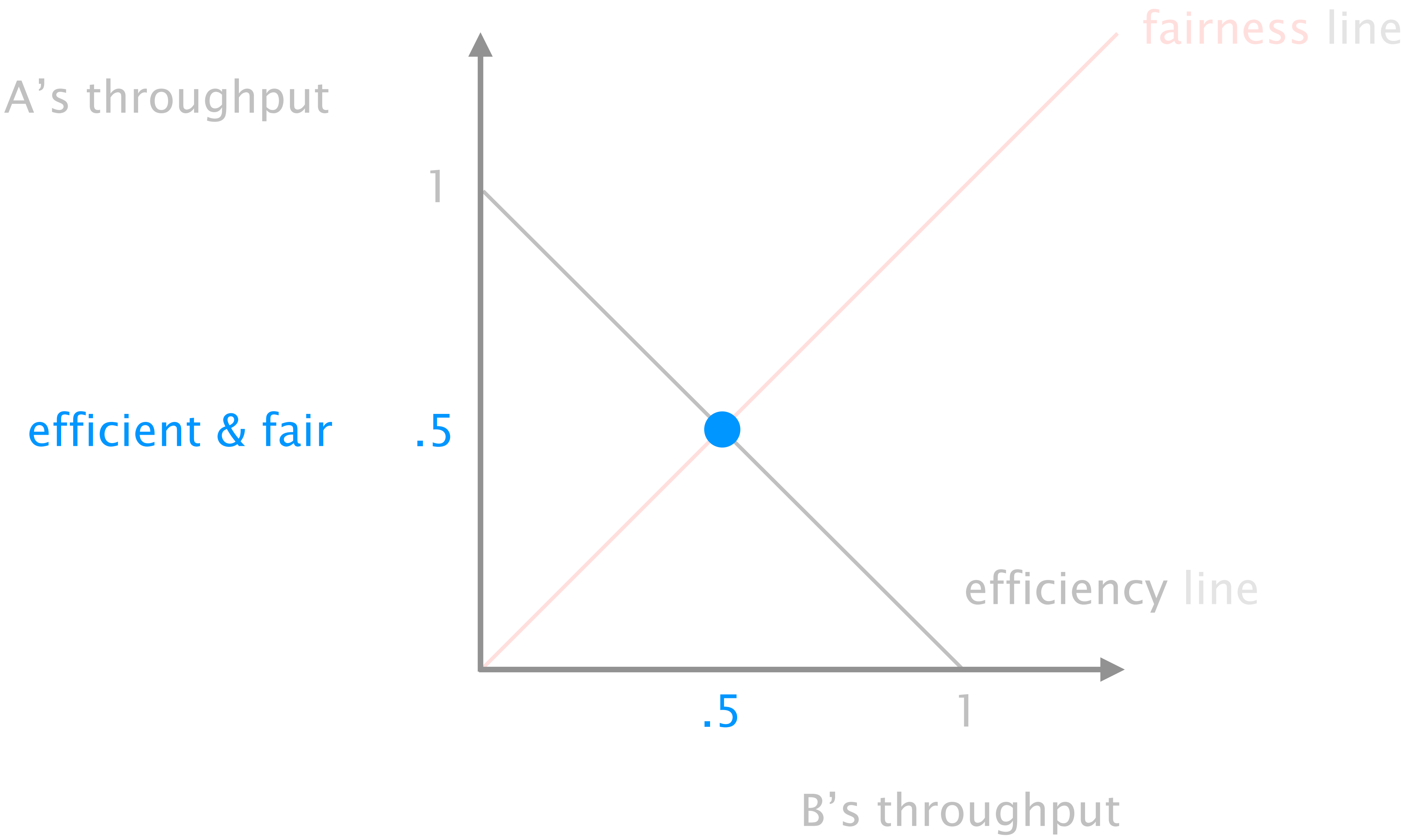
fairness line
A's throughput
1
A gets more than B
1
B's throughput


fairness line
A's throughput
1
inefficient & unfair
.5
efficiency line
.2
1
B's throughput


fairness line
A's throughput
1
congested
.7
efficiency line
.5
1
B's throughput



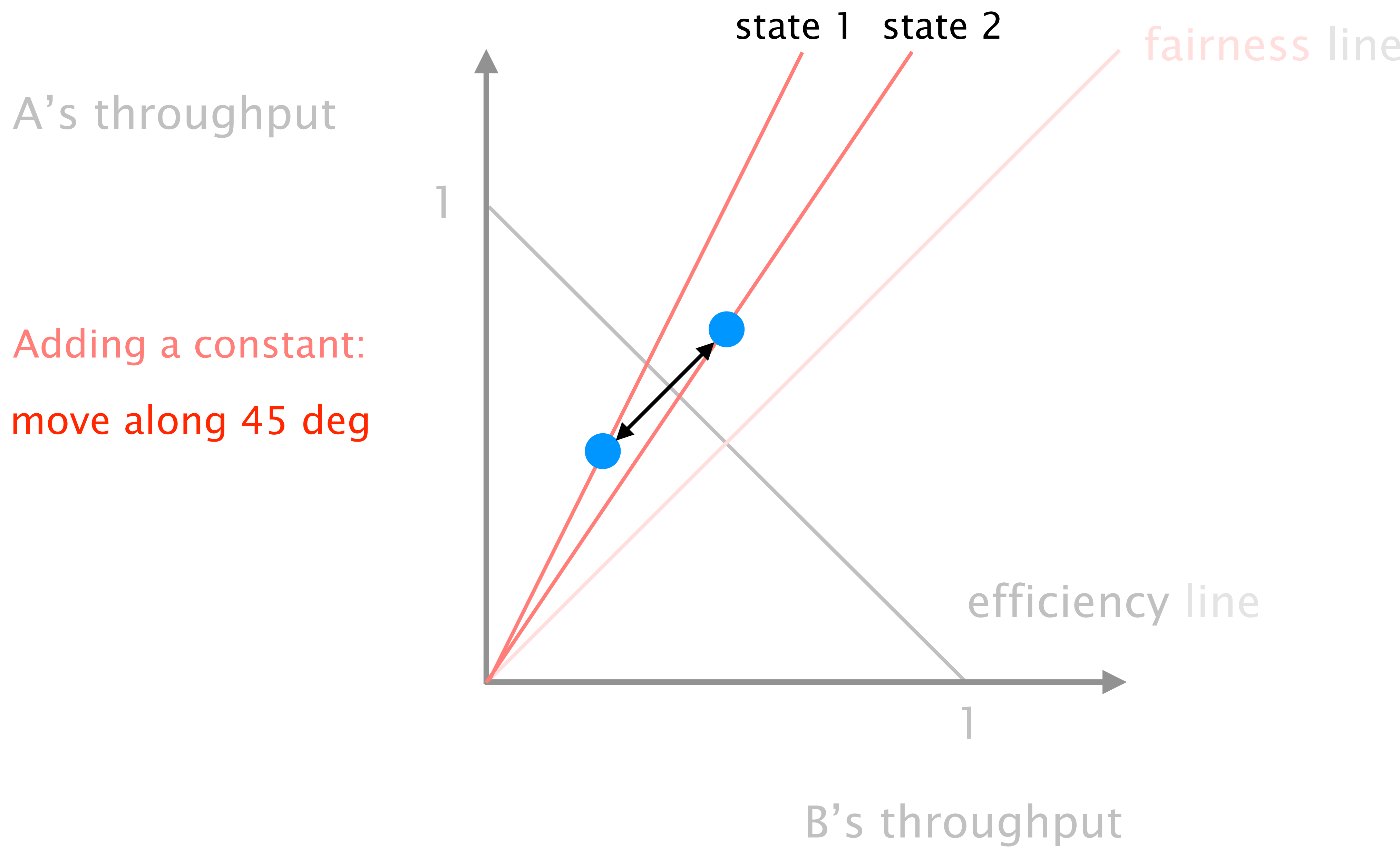
fairness line
A's throughput
1
efficient & unfair
.3
efficiency line
.7
1
B's throughput


fairness line
A's throughput
1
efficient & fair
.5
efficiency line
.5
1
B's throughput

	increase behavior	decrease behavior
AIAD	gentle	gentle
AIMD	gentle	aggressive
MIAD	aggressive	gentle
MIMD	aggressive	aggressive

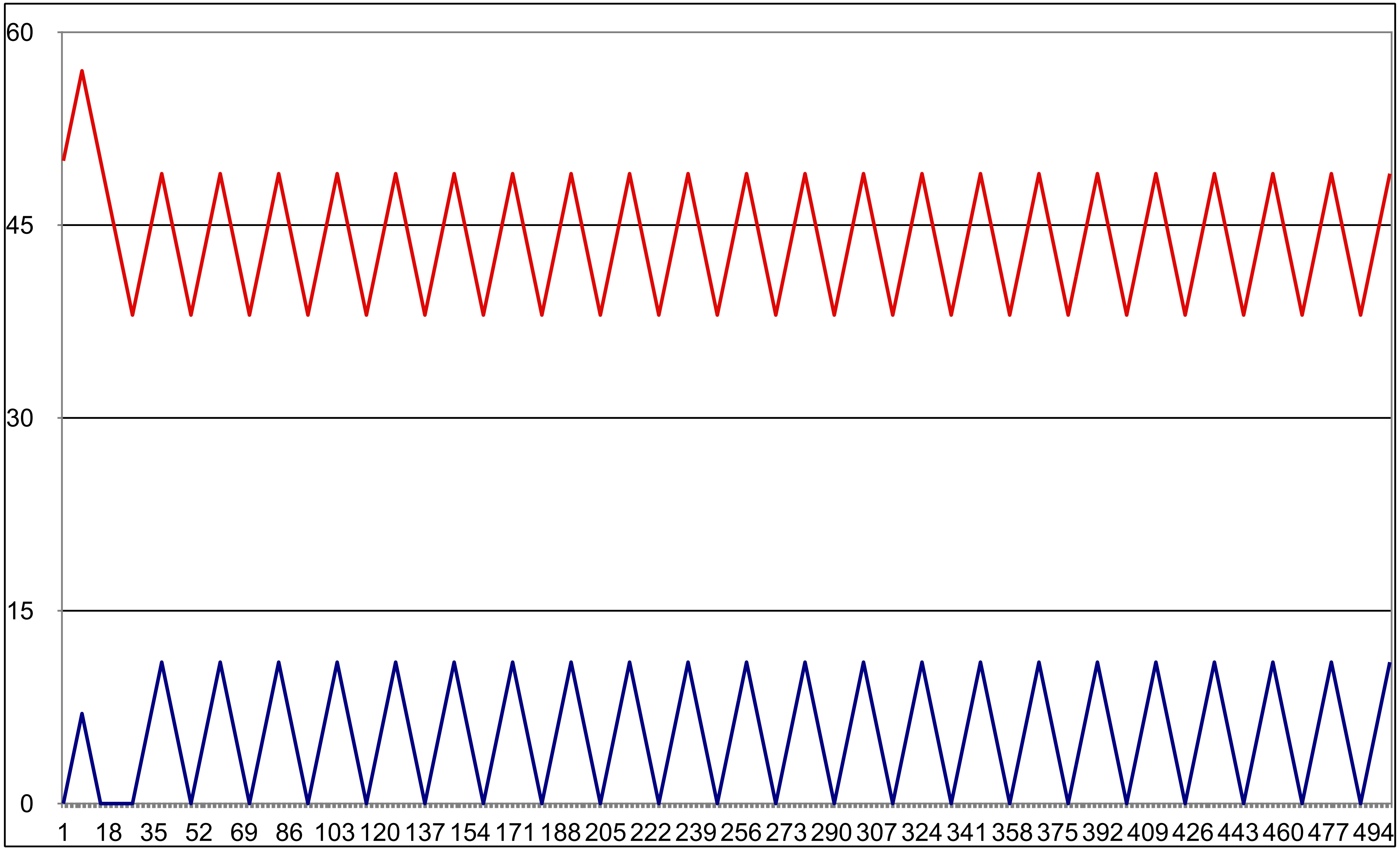
AIAD does not converge to fairness, nor efficiency:

the system fluctuates between two fairness states



AIAD does not converge to fairness, nor efficiency:

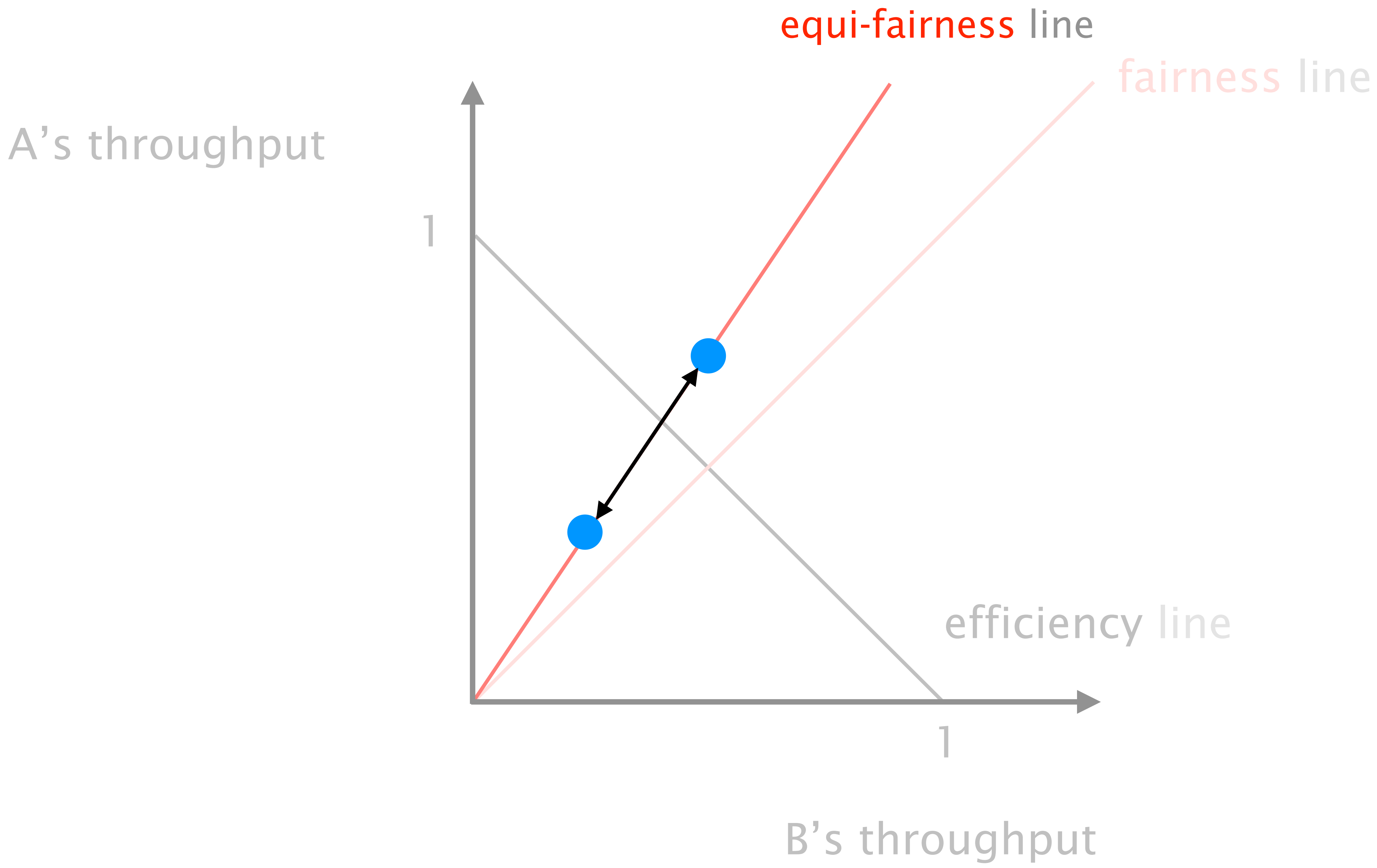
the system fluctuates between two fairness states



	increase behavior	decrease behavior
AIAD	gentle	gentle
AIMD	gentle	aggressive
MIAD	aggressive	gentle
MIMD	aggressive	aggressive

MIMD does not converge to fairness, nor efficiency:

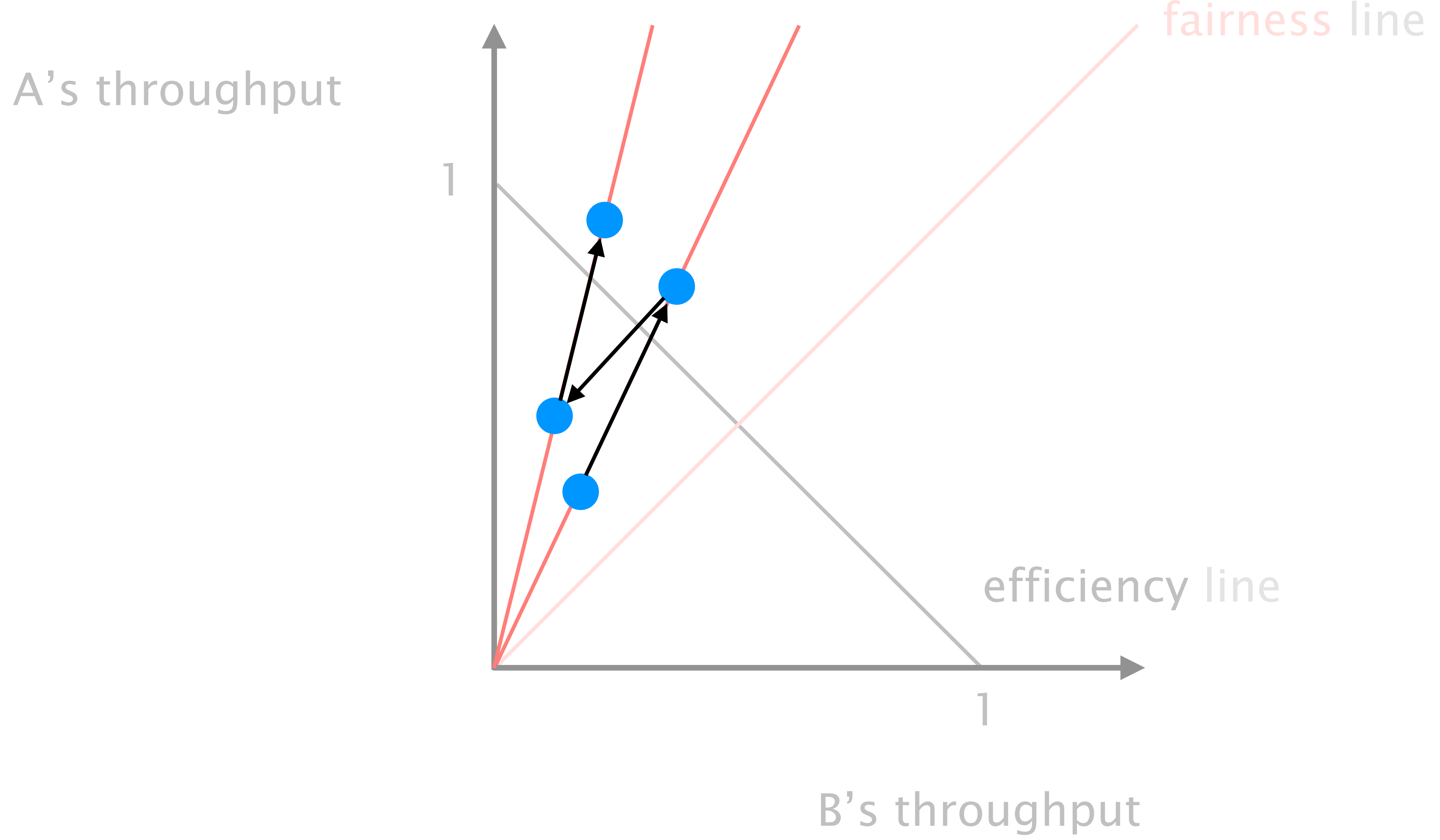
the system fluctuates along a equi-fairness line



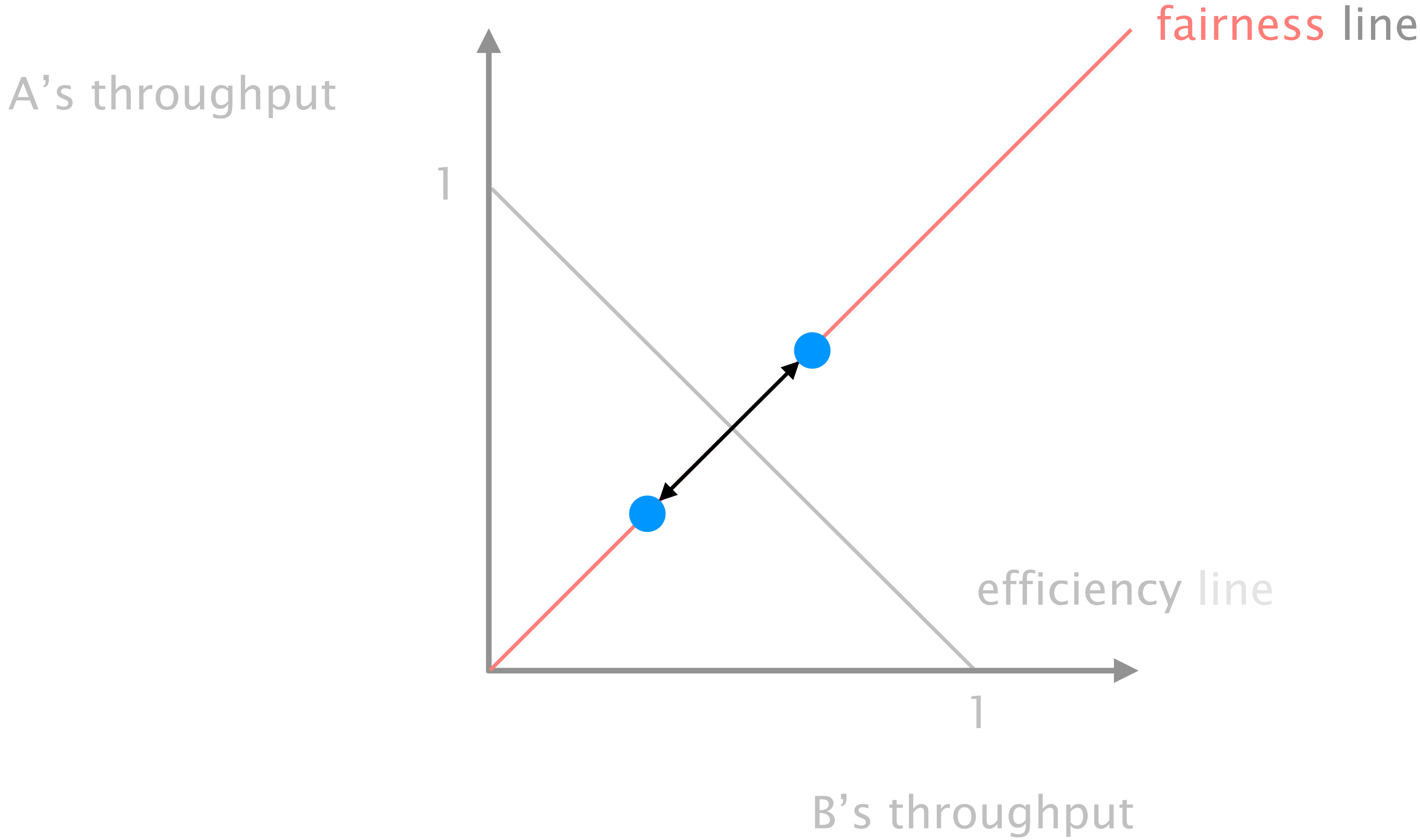
	increase behavior	decrease behavior
AIAD	gentle	gentle
AIMD	gentle	aggressive
MIAD	aggressive	gentle
MIMD	aggressive	aggressive

MIAD converges to a totally unfair allocation,

favoring the flow with a greater rate at the beginning

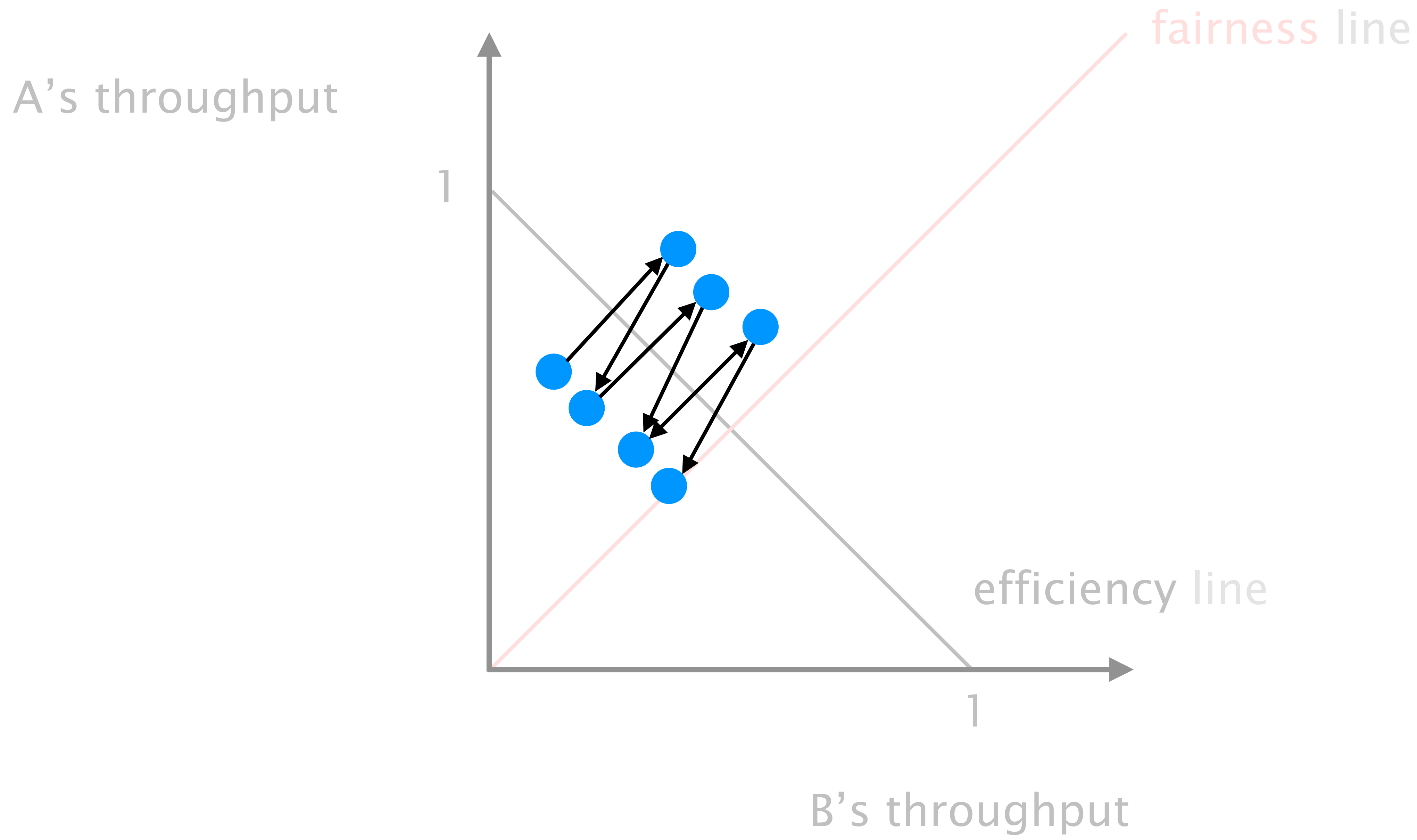


If flows start along the fairness line, MIAD fluctuates along it, yet deviating from it at the slightest change



	increase behavior	decrease behavior
AIAD	gentle	gentle
AIMD	gentle	aggressive
MIAD	aggressive	gentle
MIMD	aggressive	aggressive

AIMD converge to fairness and efficiency,
it then fluctuates around the optimum (in a stable way)



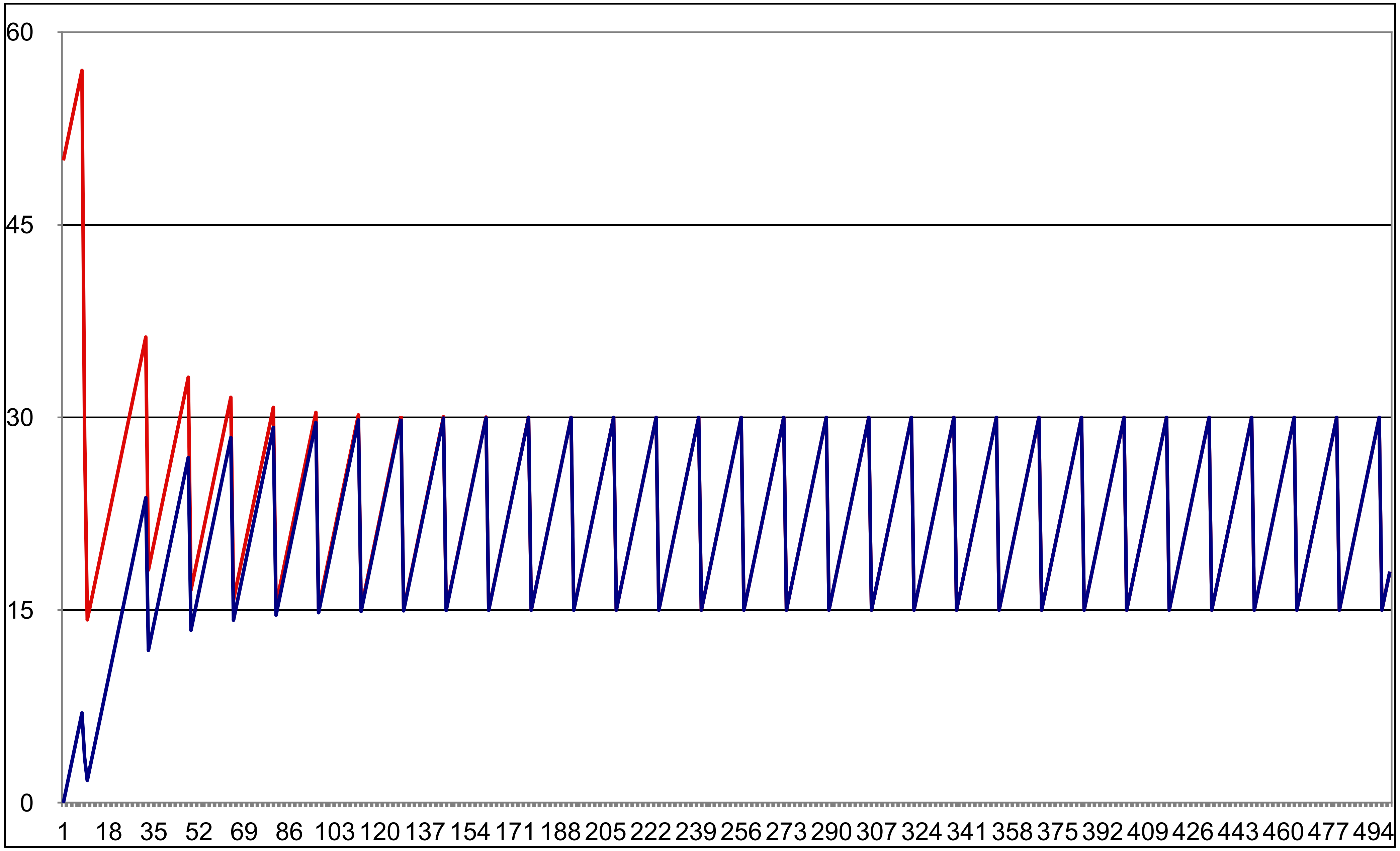
AIMD converge to fairness and efficiency, it then fluctuates around the optimum (in a stable way)

Intuition

During increase,
both flows gain bandwidth at the same rate

During decrease,
the faster flow releases more

AIMD converge to fairness and efficiency,
it then fluctuates around the optimum (in a stable way)



In practice,
TCP implements AIMD

	increase behavior	decrease behavior
AIAD	gentle	gentle
AIMD	gentle	aggressive
MIAD	aggressive	gentle
MIMD	aggressive	aggressive

In practice,
TCP implements AIMD

Implementation

After each ACK,

Increment cwnd by 1/cwnd

linear increase of max. 1 per RTT

Question

When does a sender leave slow-start and start AIMD?

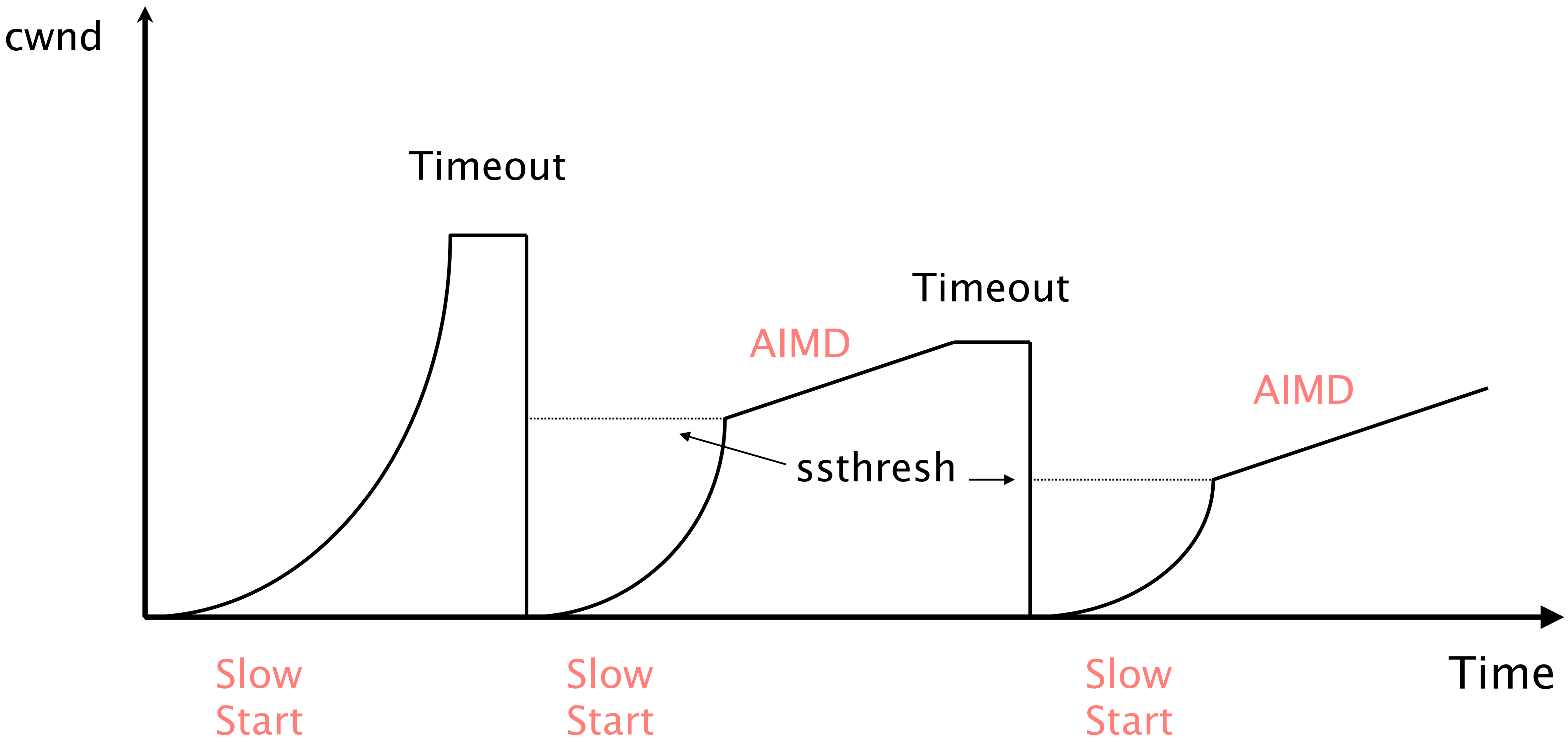
Introduce a slow start treshold, adapt it in function of congestion:

on timeout, sstresh = CWND/2

TCP congestion control in less than 10 lines of code

```
Initially:
    cwnd = 1
    ssthresh = infinite
New ACK received:
    if (cwnd < ssthresh):
        /* Slow Start*/
        cwnd = cwnd + 1
    else:
        /* Congestion Avoidance */
        cwnd = cwnd + 1/cwnd
Timeout:
    /* Multiplicative decrease */
    ssthresh = cwnd/2
    cwnd = 1
```

The congestion window of a TCP session typically undergoes multiple cycles of slow-start/AIMD



Going back all the way back to 0 upon timeout completely destroys throughput

solution

Avoid timeout expiration…

which are usually >500ms

Detecting losses can be done using ACKs or timeouts, the two signal differ in their degree of severity

duplicated ACKs	mild congestion signal packets are still making it
timeout	severe congestion signal multiple consequent losses

TCP automatically resends a segment after receiving 3 duplicates ACKs for it

this is known as a "fast retransmit"

After a fast retransmit, TCP switches back to AIMD,
without going all way the back to 0

this is known as "fast recovery"

TCP congestion control (almost complete)

```
Initially:
    cwnd = 1
    ssthresh = infinite
New ACK received:
    if (cwnd < ssthresh):
        /* Slow Start*/
        cwnd = cwnd + 1
    else:
        /* Congestion Avoidance */
        cwnd = cwnd + 1/cwnd
    dup_ack = 0
Timeout:
    /* Multiplicative decrease */
    ssthresh = cwnd/2
    cwnd = 1
```

```
Duplicate ACKs received:
    dup_ack ++;
    if (dup_ack >= 3):
        /* Fast Recovery */
        ssthresh = cwnd/2
        cwnd = ssthresh
```

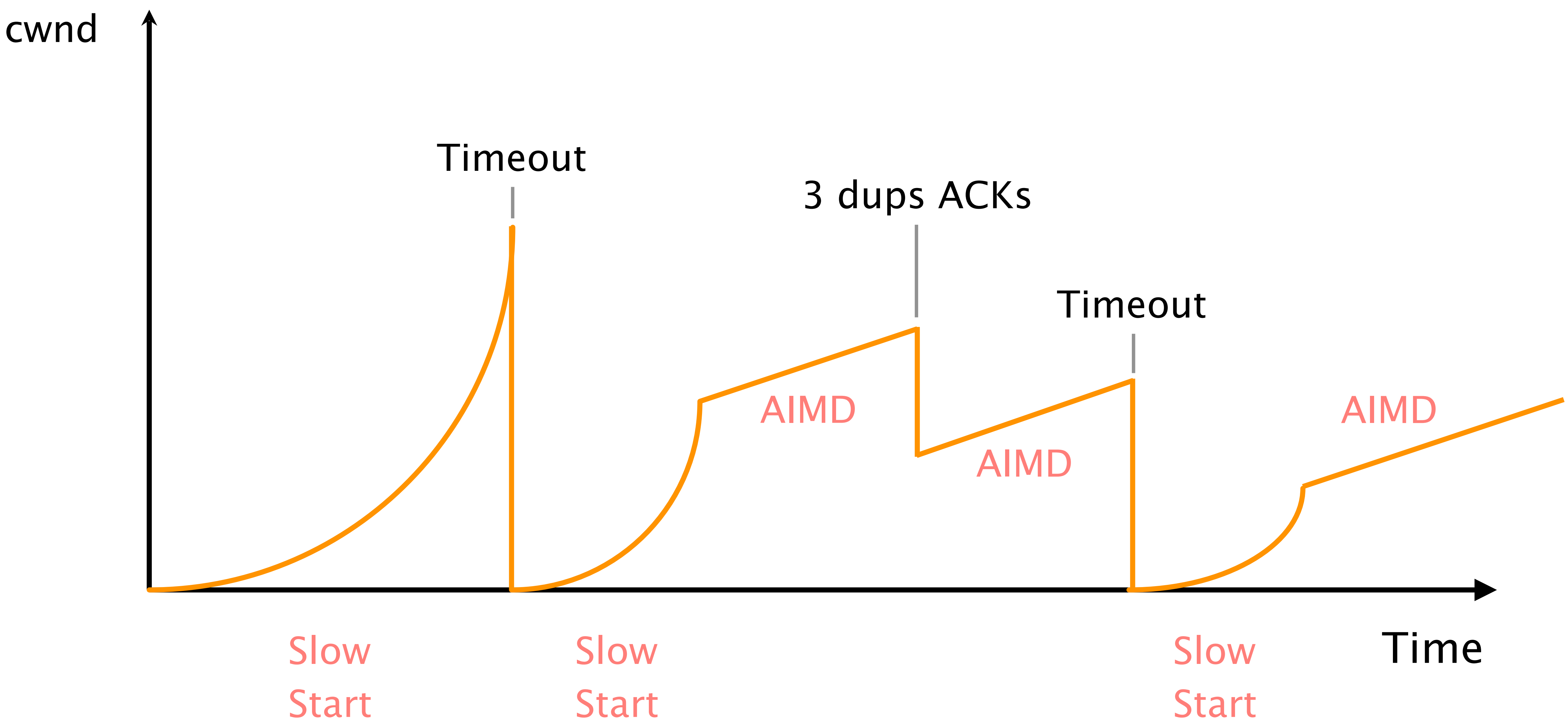
Initially:

```
    cwnd = 1
    ssthresh = infinite
New ACK received:
    if (cwnd < ssthresh):
        /* Slow Start*/
        cwnd = cwnd + 1
    else:
        /* Congestion Avoidance */
        cwnd = cwnd + 1/cwnd
    dup_ack = 0
Timeout:
    /* Multiplicative decrease */
    ssthresh = cwnd/2
    cwnd = 1
```

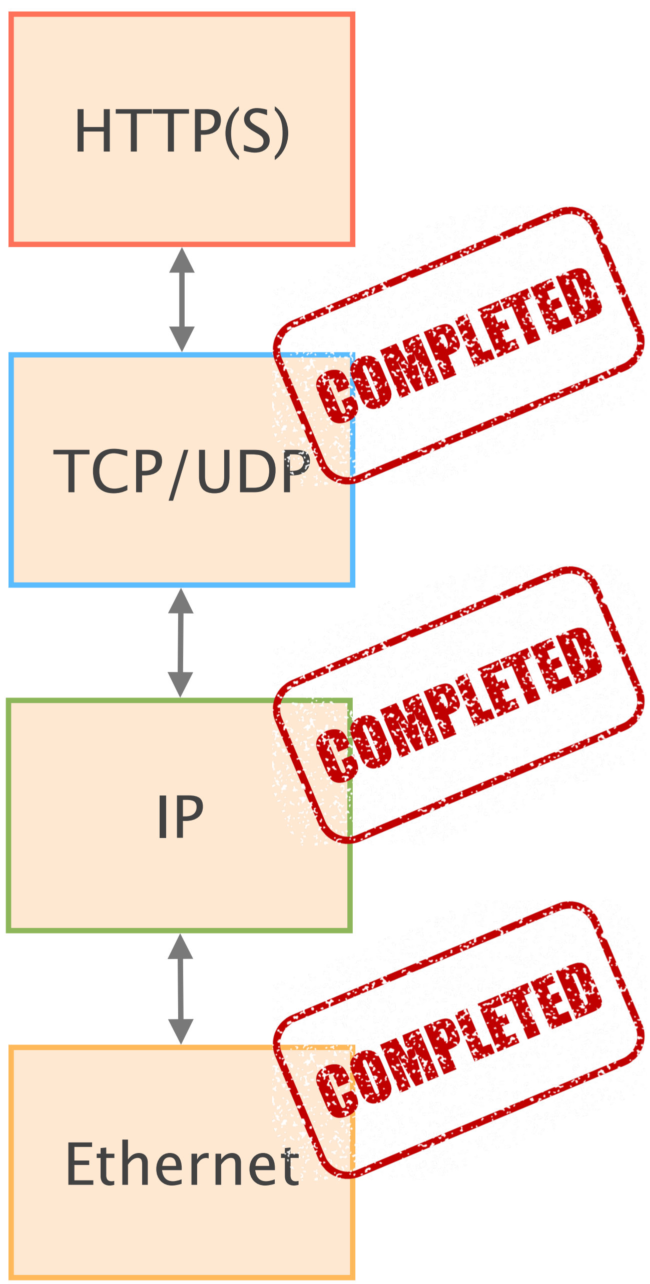
Duplicate ACKs received:

```
    dup_ack ++;
    if (dup_ack >= 3):
        /* Fast Recovery */
        ssthresh = cwnd/2
        cwnd = ssthresh
```

Congestion control makes TCP throughput look like a “sawtooth”



We now have completed the transport layer (!)



Application	HTTP(S)	
Transport	TCP/UDP	Reliable (or unreliable) transport
Network	IP	Best-effort global packet delivery
Link	Ethernet	Best-effort local packet delivery

Communication Networks

Spring 2022

Laurent Vanbever

nsg.ee.ethz.ch

ETH Zürich (D-ITET)

May 9 2022