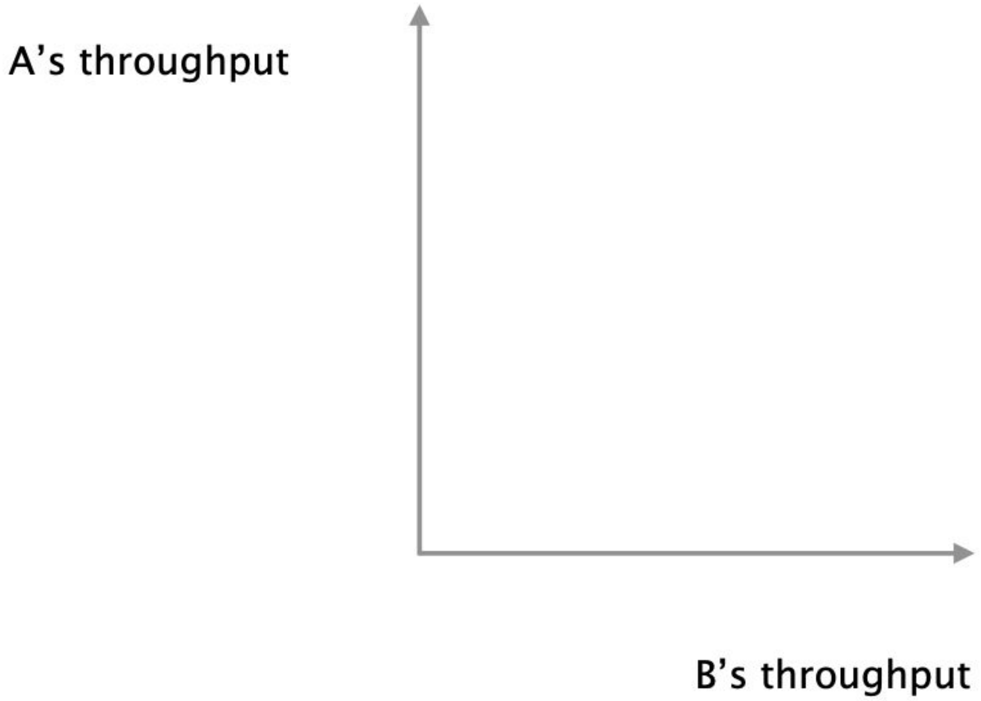
The goal here is to track the available bandwidth, and oscillate around its current value

How do we choose a scheme? Based on fairness

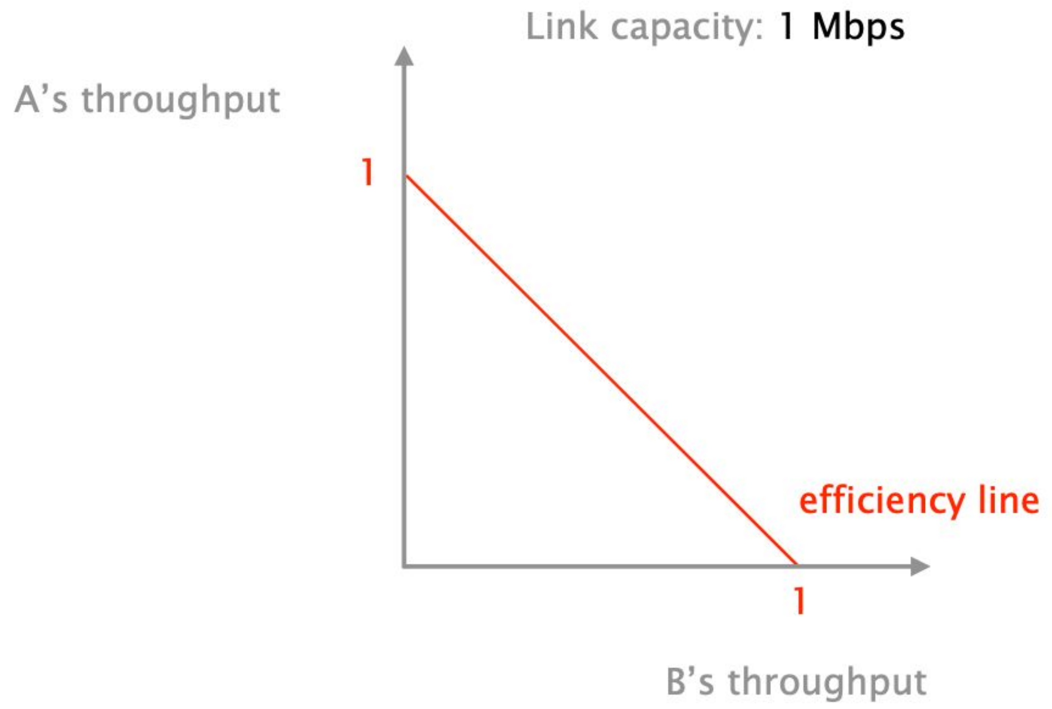
AIAD	gentle	gentle
AIMD	gentle	aggressive
MIAD	aggressive	gentle
MIMD	aggressive	aggressive

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

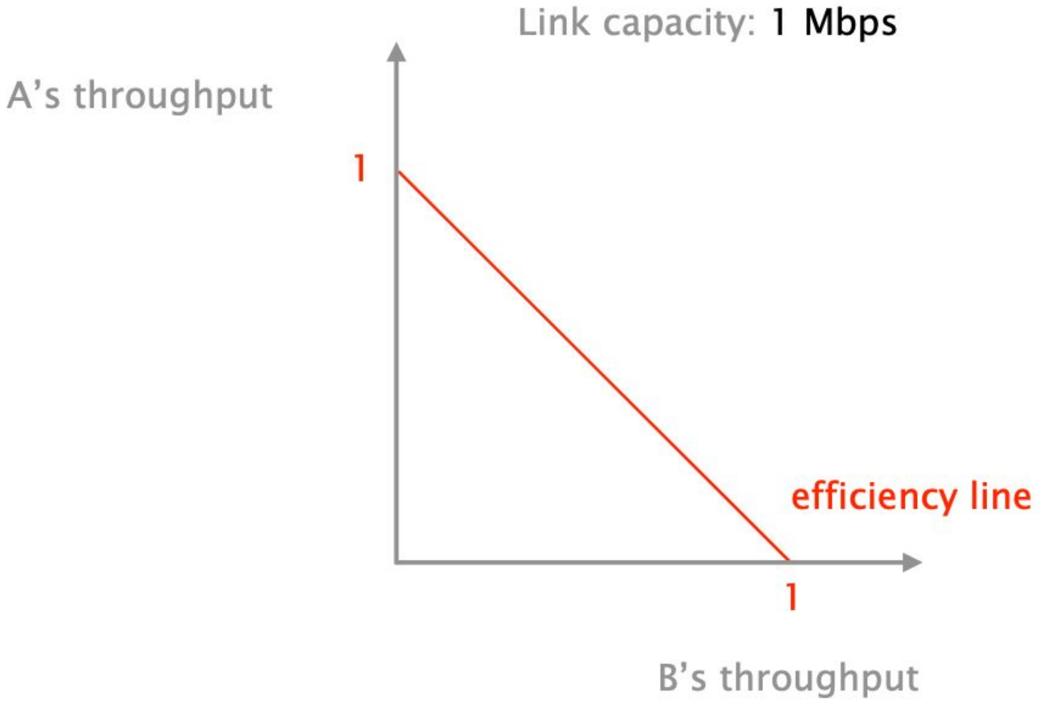
We can analyze the system behavior using a system trajectory plot



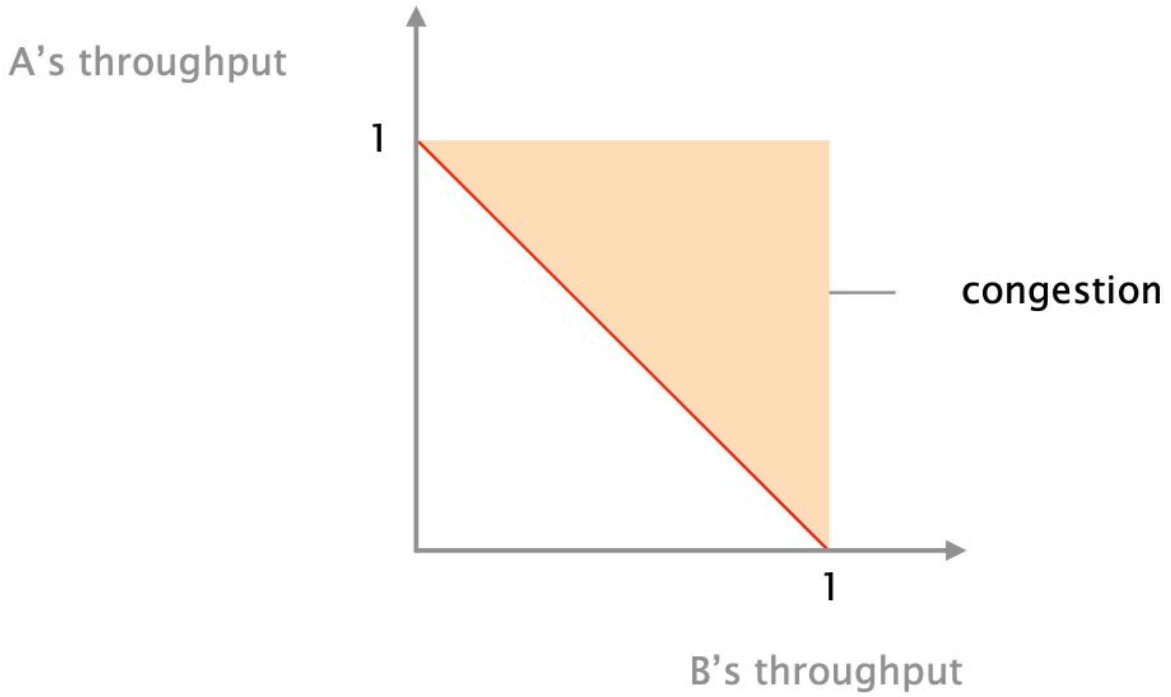
The system is efficient if the capacity is fully used, defining an efficiency line where a + b = 1



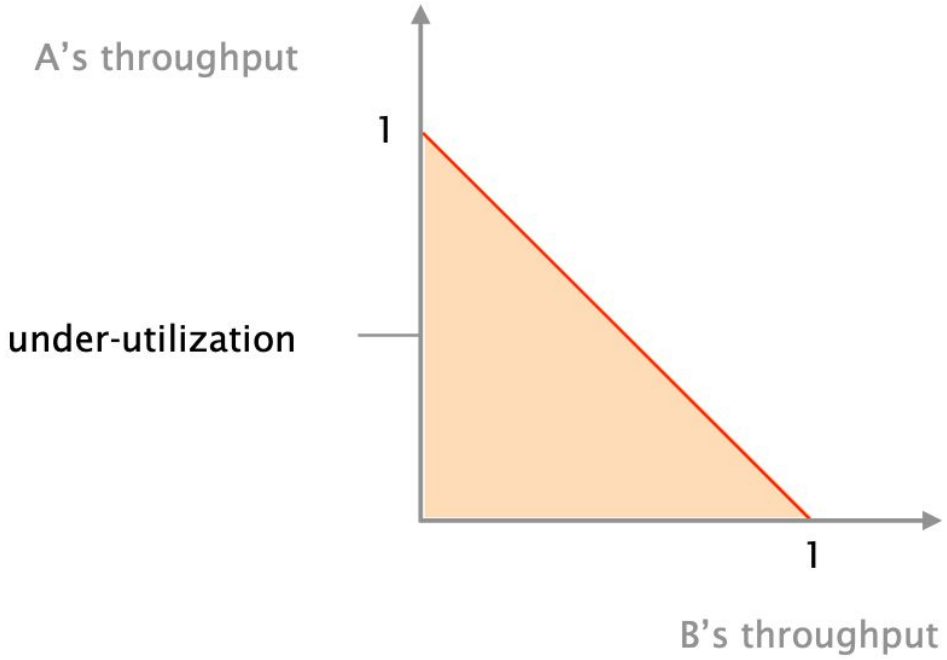
The goal of congestion control is to bring the system as close as possible to this line, and stay there



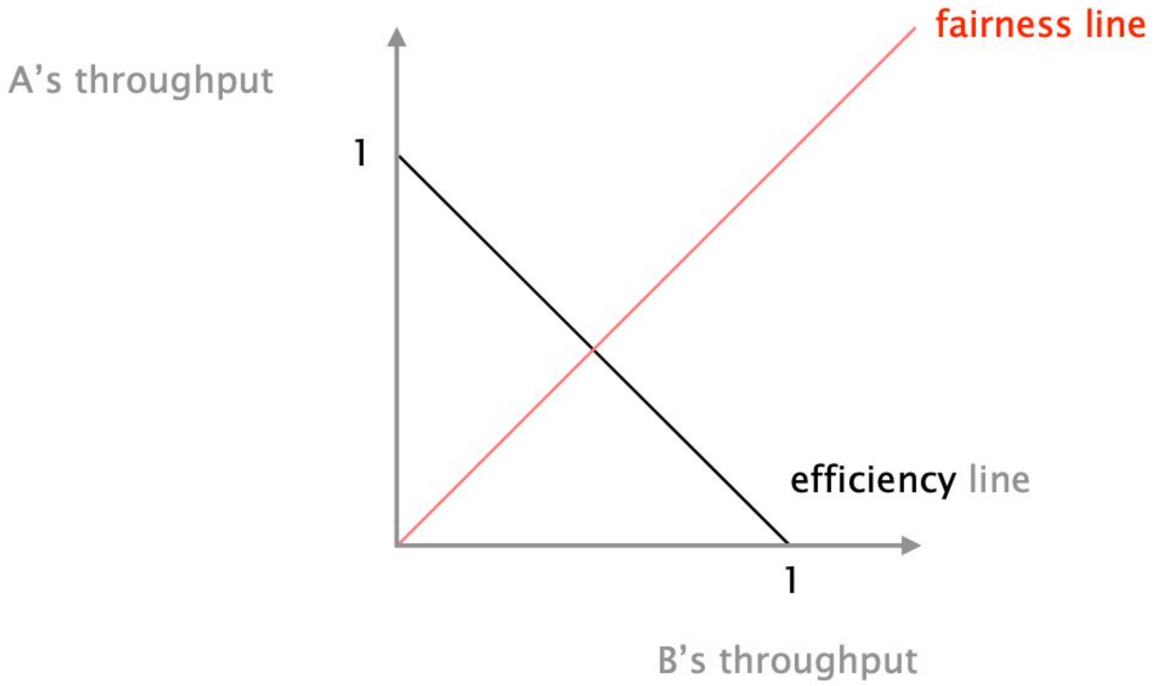
The goal of congestion control is to bring the system as close as possible to this line, and stay there



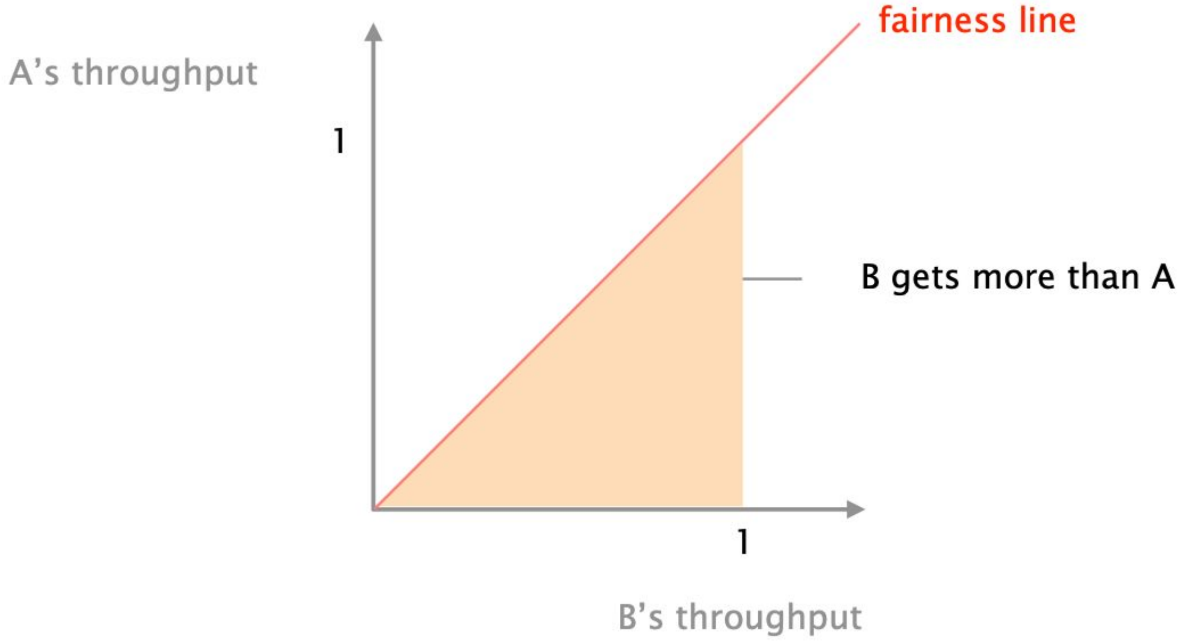
The goal of congestion control is to bring the system as close as possible to this line, and stay there



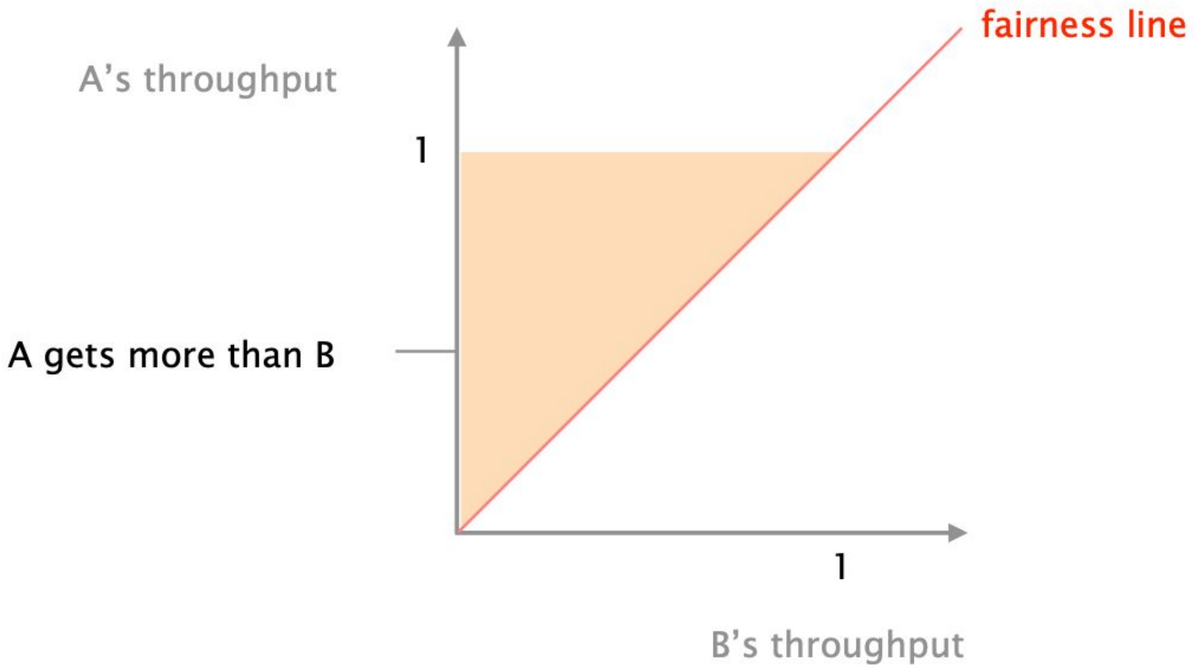
The system is fair whenever A and B have equal throughput, defining a fairness line where a = b

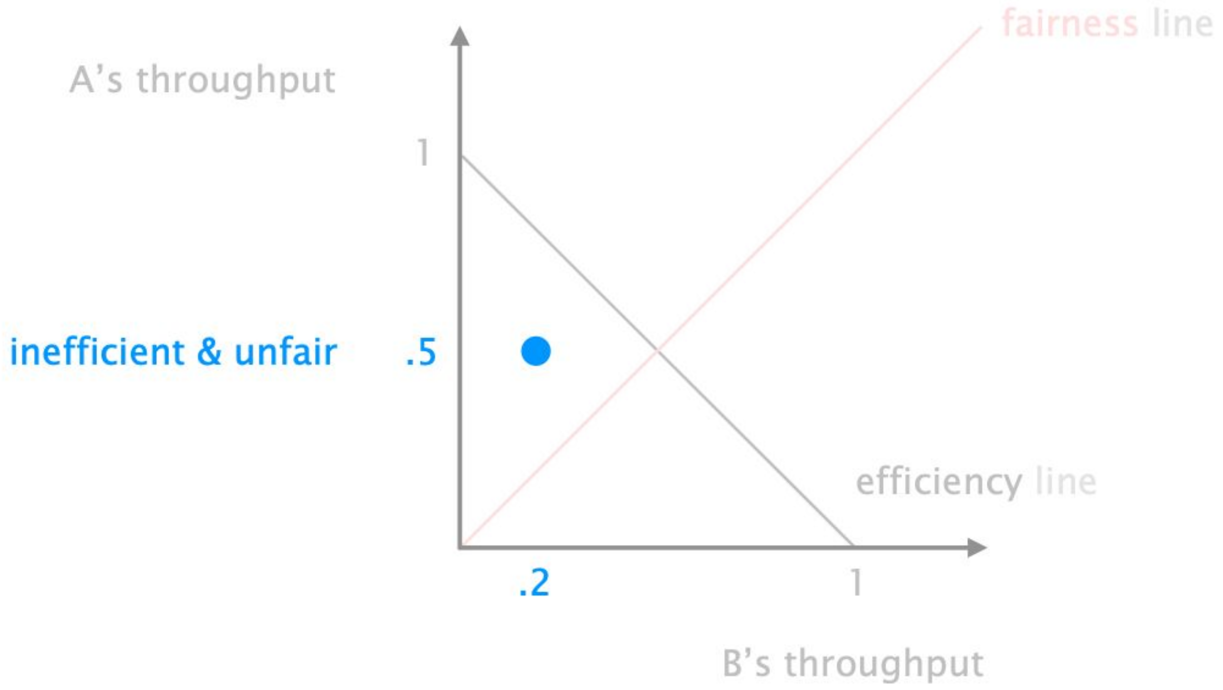


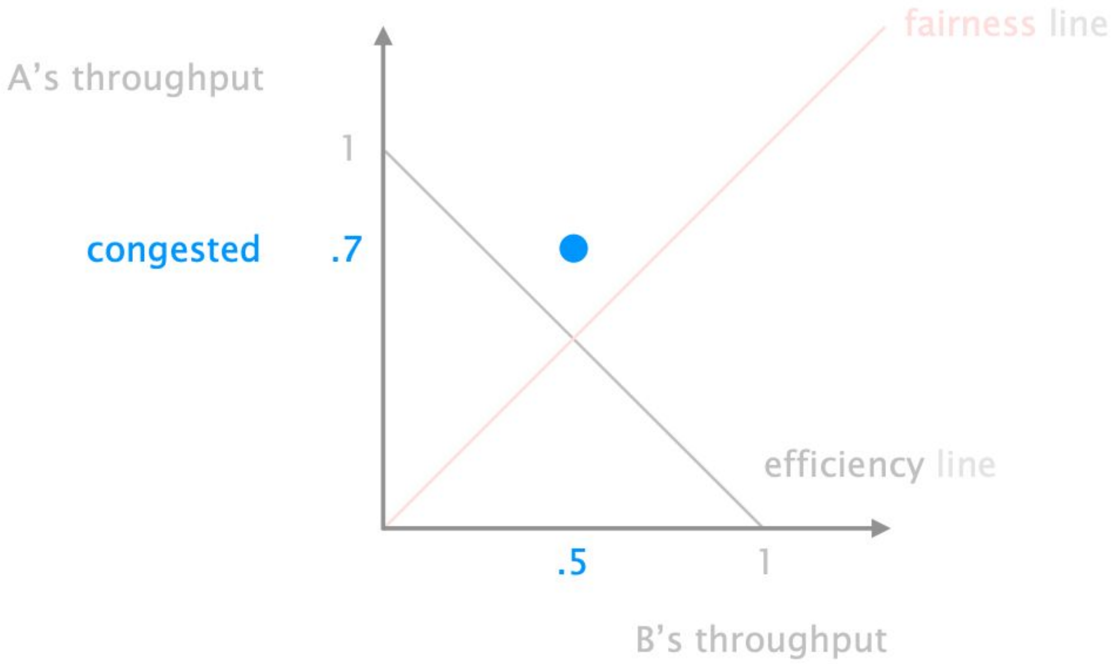
The system is fair whenever A and B have equal throughput, defining a fairness line where a = b

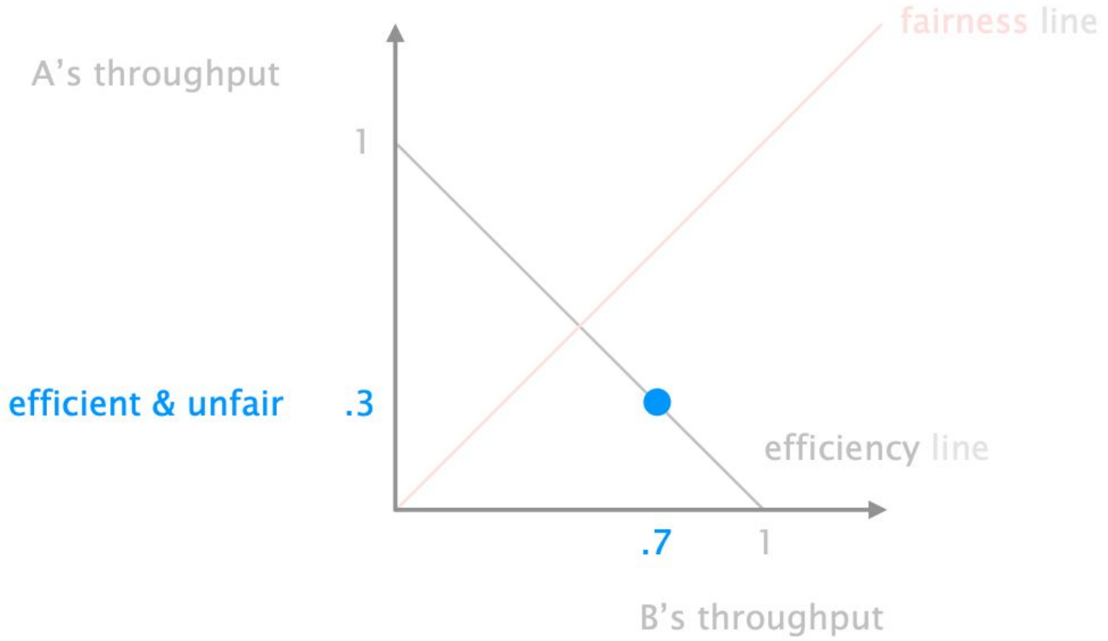


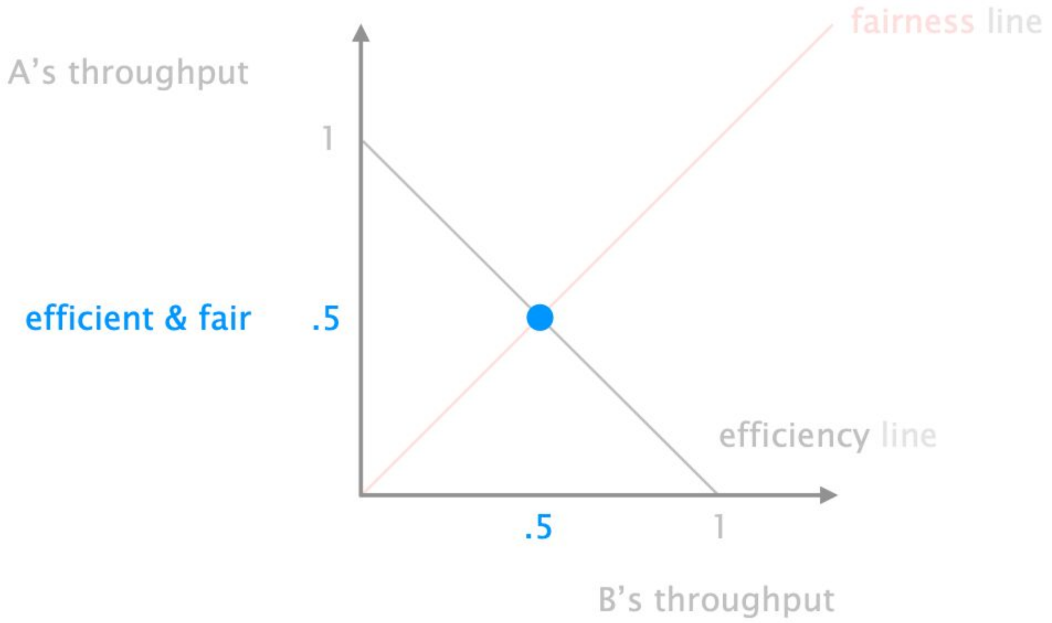
The system is fair whenever A and B have equal throughput, defining a fairness line where a = b




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

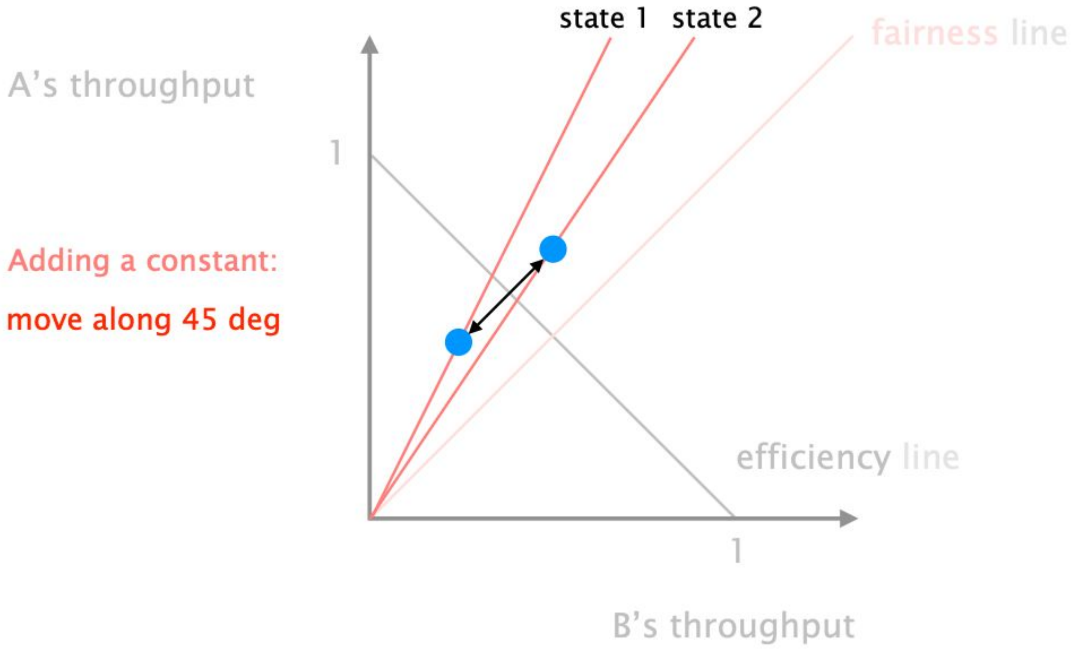

fairness line
A's throughput
1
congested
.7
efficiency line
.5
1
B's throughput


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

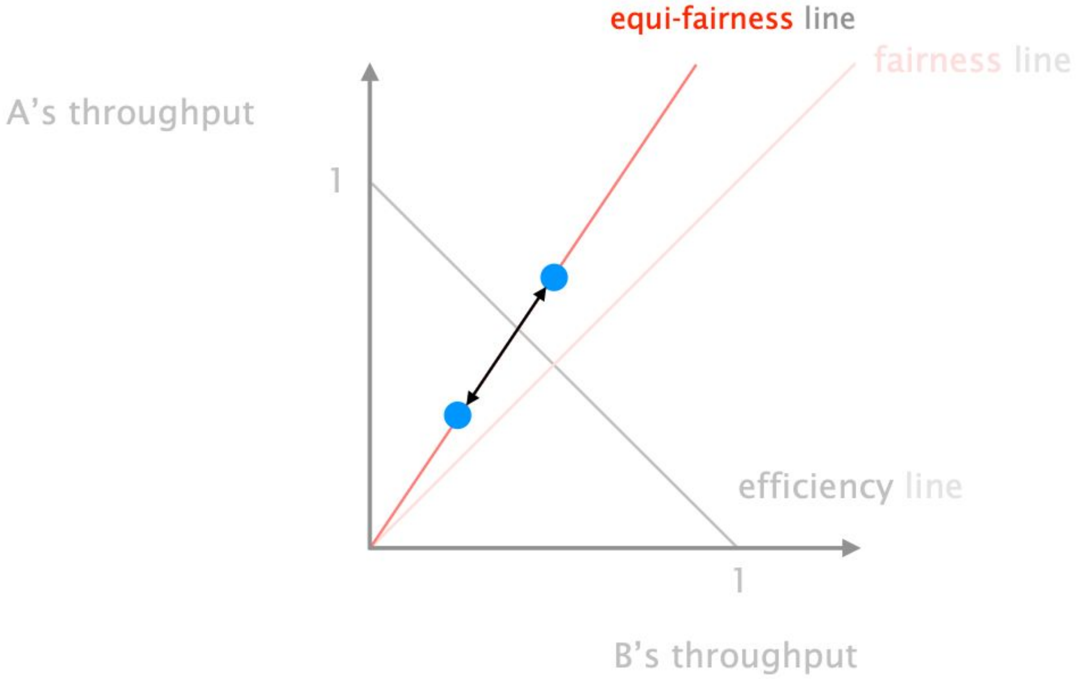
	increase behavior	decrease behavior
AIAD	gentle	gentle
AIMD	gentle	aggressive
MIAD	aggressive	gentle
MIMD	aggressive	aggressive

AIAD does not converge to fairness, nor efficiency: the system fluctuates between two fairness states



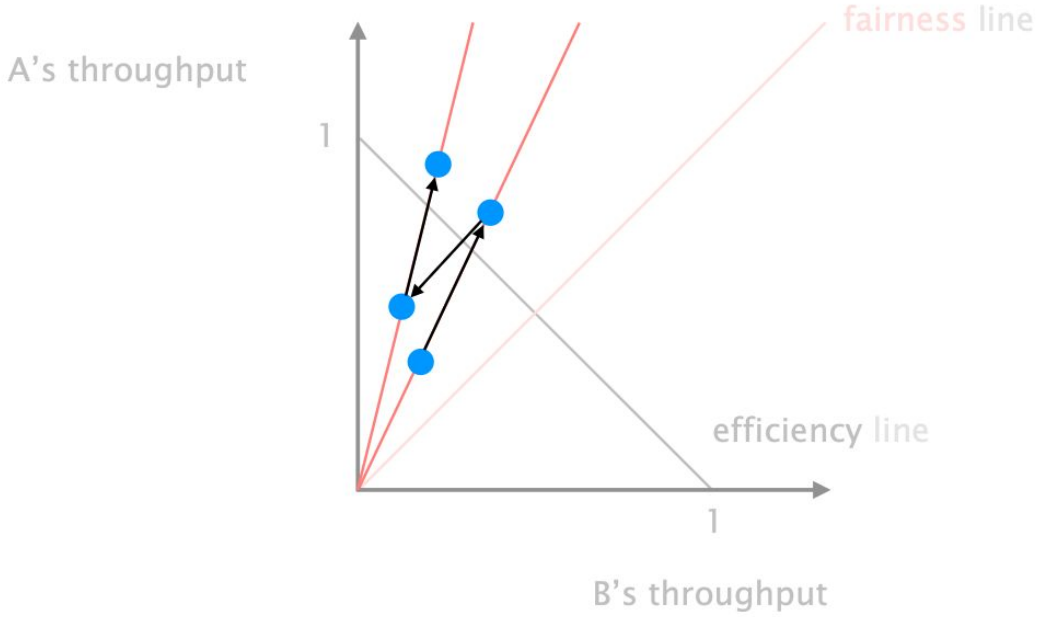
	increase behavior	decrease behavior
AIAD	gentle	gentle
AIMD	gentle	aggressive
MIAD	aggressive	gentle
MIMD	aggressive	aggressive

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

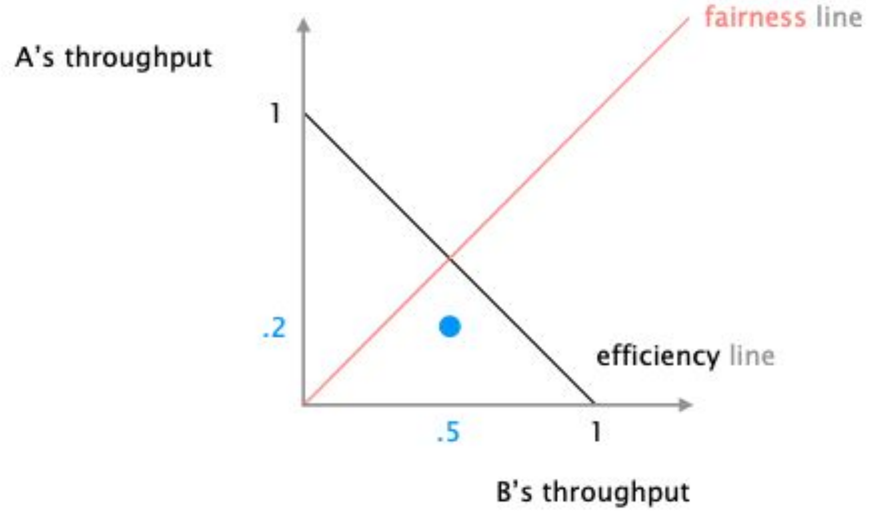


Congestion control exercise

Consider the situation in which two hosts, A and B, are concurrently using a 1 Mbps link with a Maximum Segment Size (MSS) of 100 kb.

Assuming that B starts with 500 kbps and A with 200 kbps (see left picture).

What would happen if both are using MIAD (assume both senders double their CWND MSS when there is no congestion and subtract it by 1 upon congestion).

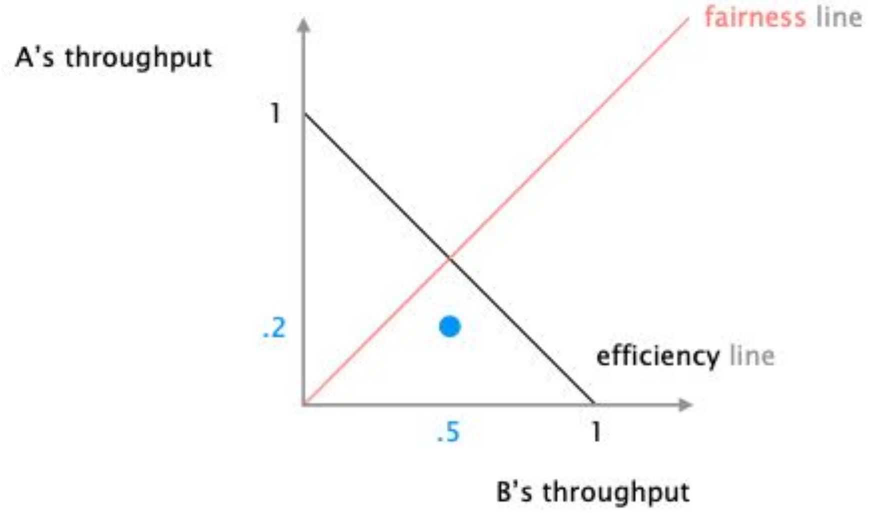


Congestion control exercise

Solution

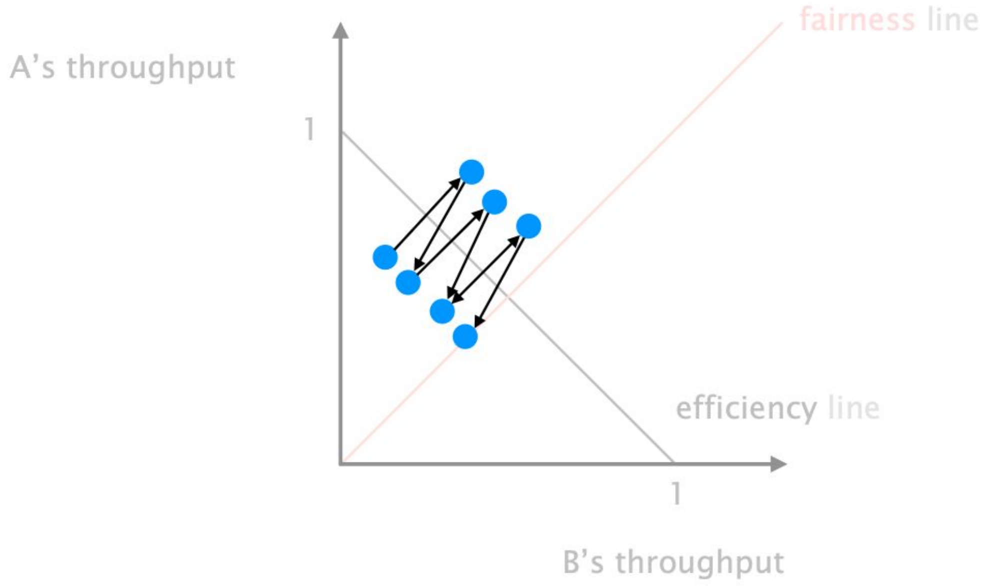
1. (.2, .5)
2. (.4, 1) > congestion!
3. (.3, .9) > congestion!
4. (.2, .8)
5. (.4, 1.6) > congestion!
6. (.3, 1.5) > congestion!
7. (.2, 1.4) > congestion!
8. (.1, 1.3) > congestion!
9. (0, 1.2) > congestion!
10. (0, 1.1) > congestion!
11. (0, 1)

The sender which benefits from a bigger initial share will end up using the entire link.



	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,

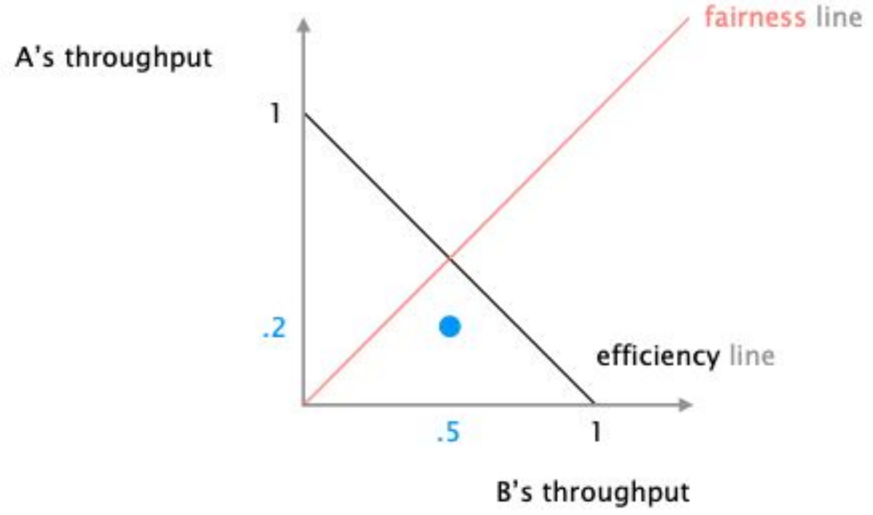
the faster flow releases more

Congestion control exercise

Consider the situation in which two hosts, A and B, are concurrently using a 1 Mbps link with a Maximum Segment Size (MSS) of 100 kb.

Assuming that B starts with 500 kbps and A with 200 kbps (see left picture).

What would happen if both are using AIMD (assume both senders <u>increase their CWND by 1</u> MSS when there is no congestion and <u>divide it by 2</u> upon congestion).

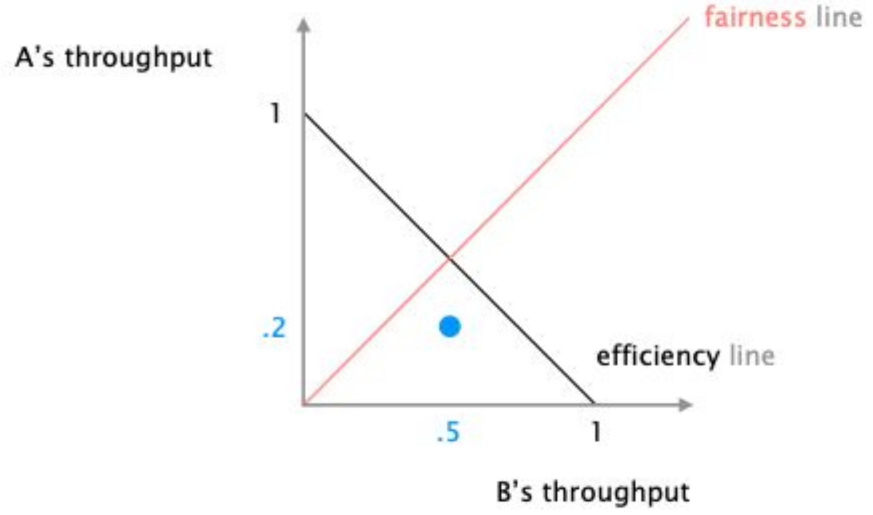


Congestion control exercise

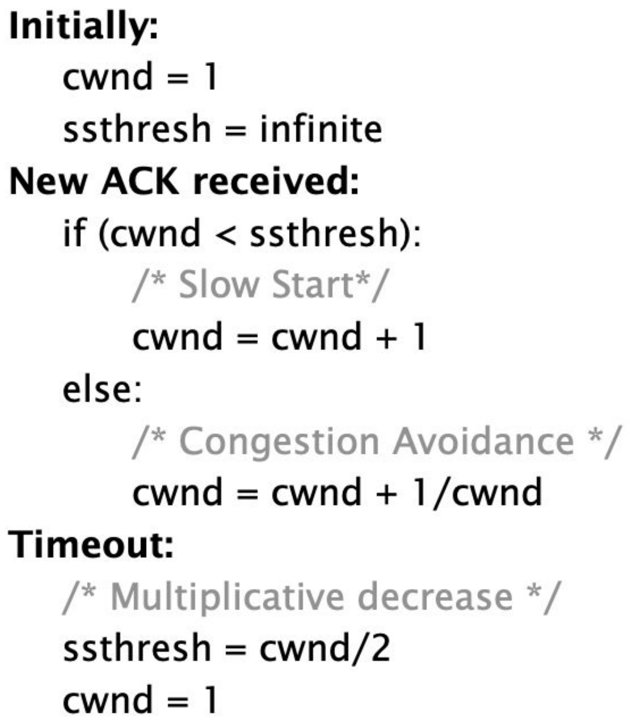
Solution

1. (.3, .6)
2. (.4, .7) > congestion!
3. (.2, .35)
4. (.3, .45)
5. (.4, .55)
6. (.5, .65) > congestion!
7. (.25, .325)

Because of its bigger share, B loses more than A because of the halving, eventually the system converges along the fairness line.

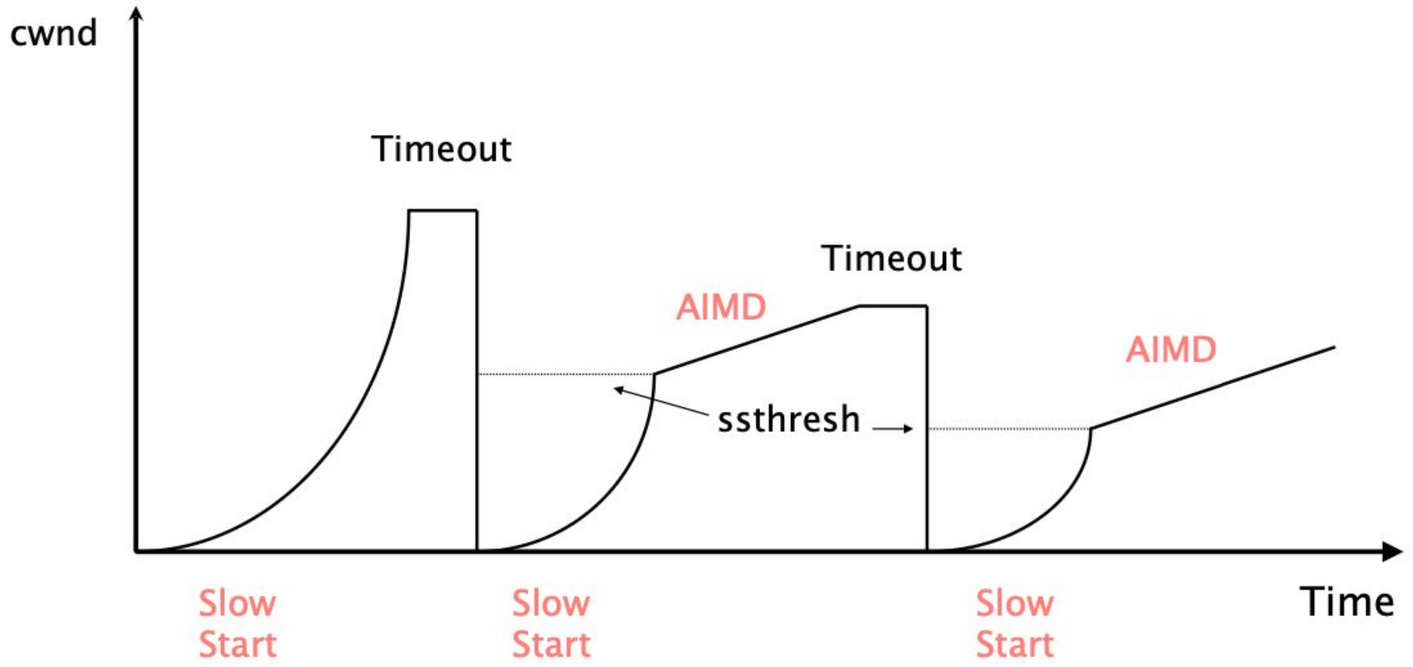


TCP uses AIMD for congestion avoidance

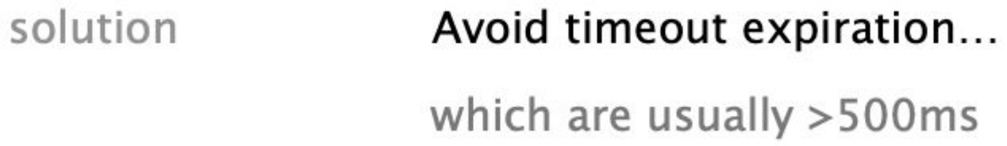


```
Initially:
    cwnd = 1
    ssthresh = infinite
New ACK received:
    if (cwnd < ssthresh):
        /* Slow Start*/
        cwnd = cwnd + 1
    else:
        /* Congestion Avoidance */
        cwnd = cwnd + 1/cwnd
Timeout:
    /* Multiplicative decrease */
    ssthresh = cwnd/2
    cwnd = 1
```

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 duplicate ACKs for it

Known as **fast retransmit**

Timeouts are slow (1 second is fastest timeout on many TCPs)

When packet is lost, receiver still ACKs last in-order packet

Use 3 duplicate ACKs to indicate a loss; detect losses quickly

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

Known as **fast recovery**

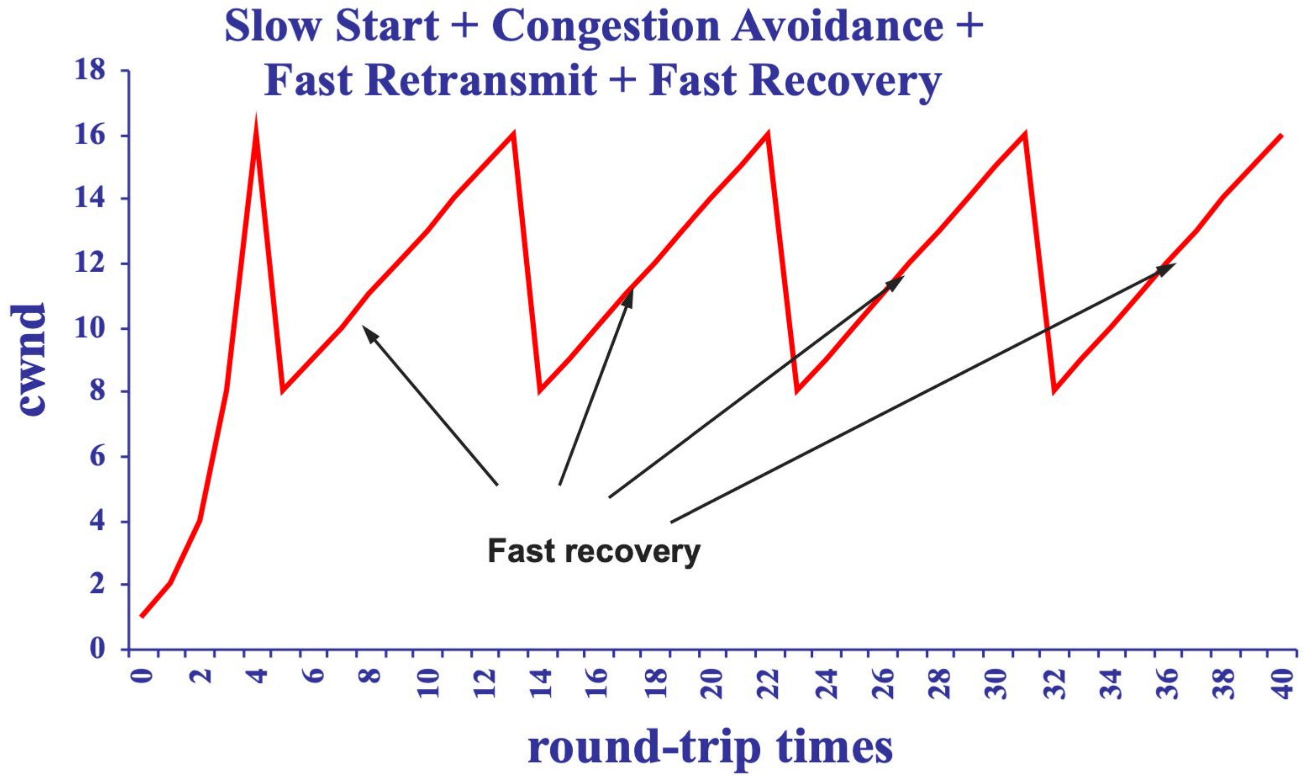
Goal: avoid stalling after loss

If there are still ACKs coming in, then no need for slow start If a packet has made it through -> we can send another one

Divide cwnd by 2 after fast retransmit

Increment cwnd by 1 full pkt for each additional duplicate ACK

More sophisticated TCP

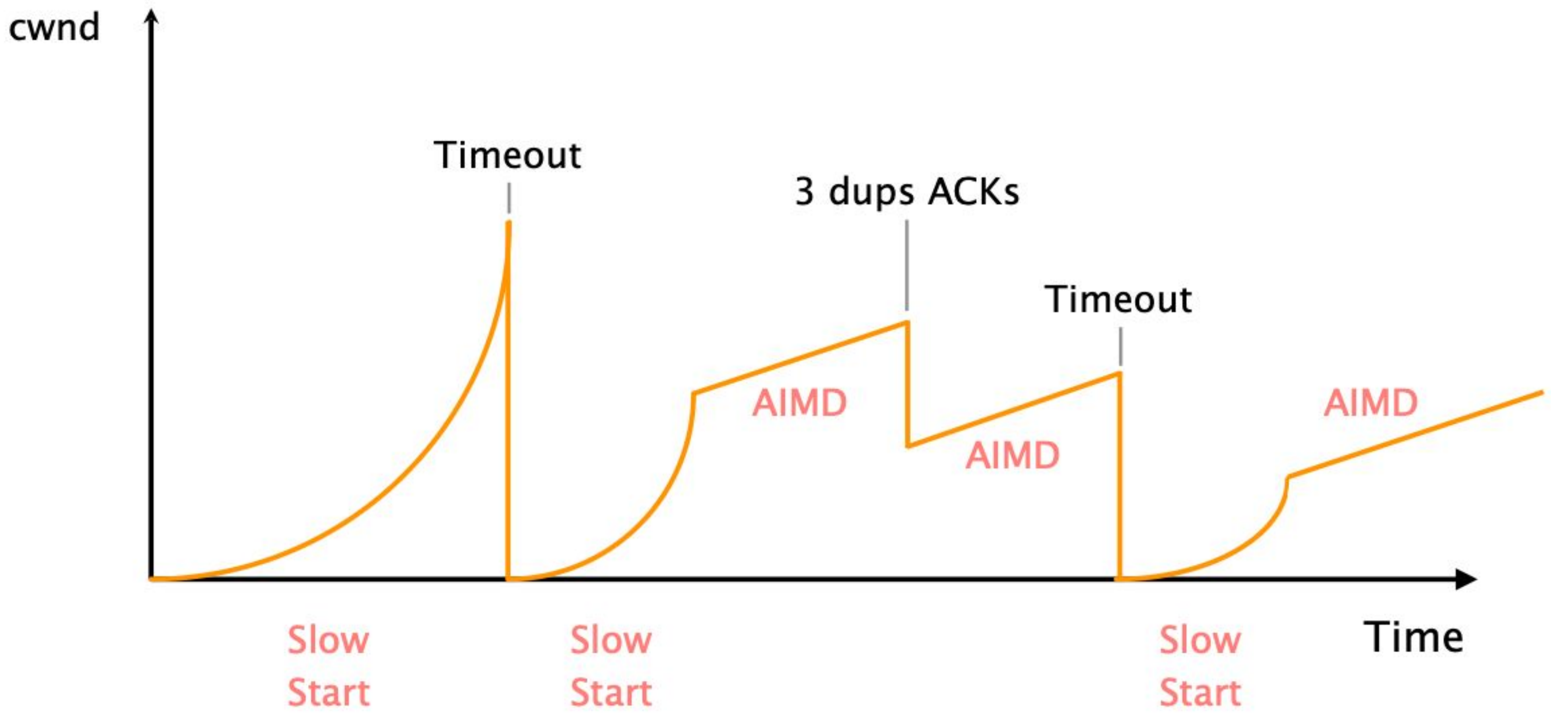


TCP congestion control

```
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”


cwnd
Timeout
3 dups ACKs
Timeout
AIMD
AIMD
AIMD
Slow Start
Slow Start
Slow Start
Time