Advanced Computer Networks

263-3501-00

Datacenter TCP

Spring Semester 2017

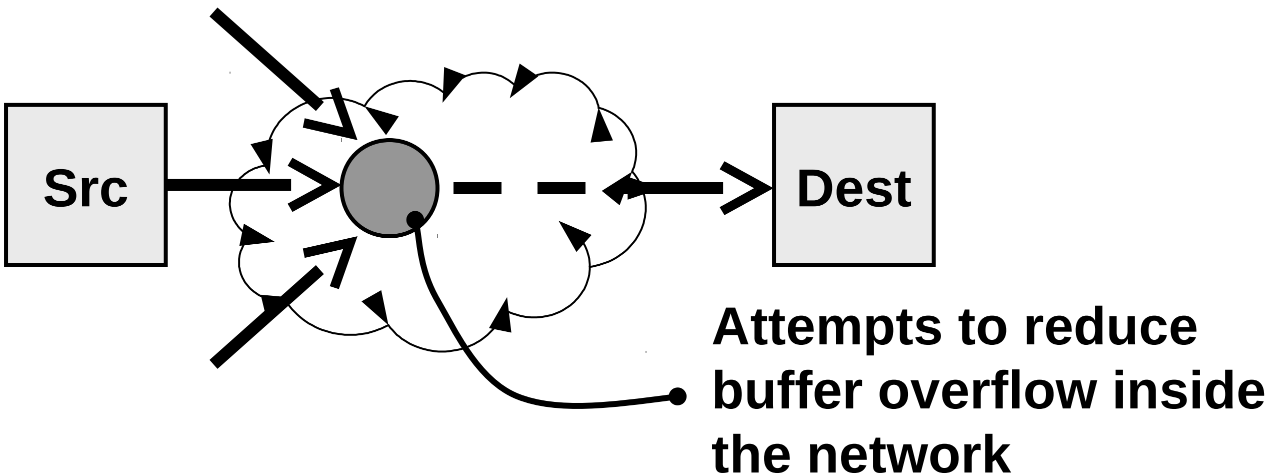
Today

- Problems with TCP in the Data Center
  - TCP Incast
  - TPC timeouts
- Improvements to TCP for the Data Center
  - Data Center TCP (DCTCP)
  - Deadline Aware TCP
  - Multipath TCP (MPTCP)

Refresh:
TCP Congestion Control

Remember: TCP congestion control

- Congestion control got added to TCP to in attempt to reduce congestion inside the network
- Must rely on indirect measures of congestion
- Implemented at the sender

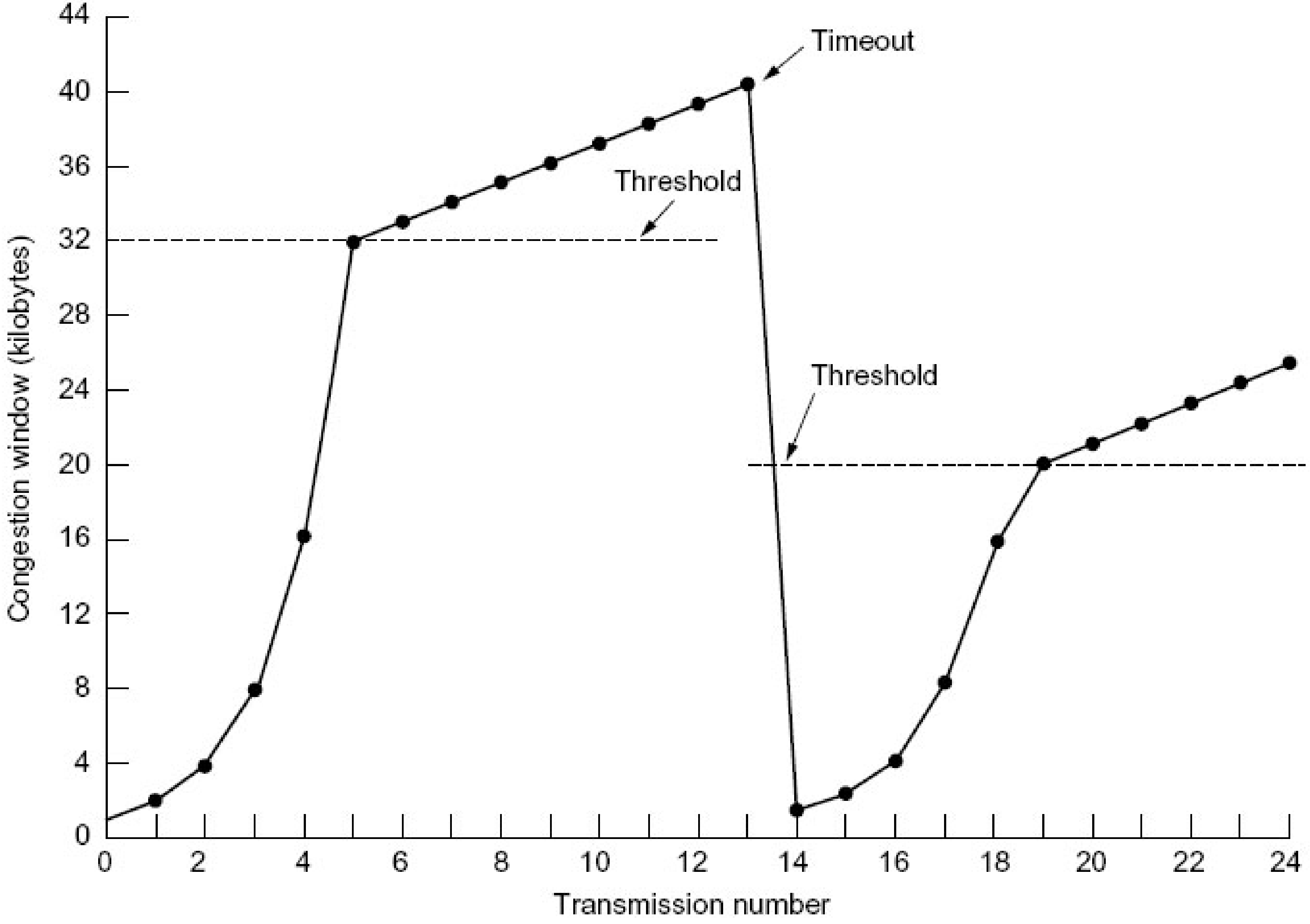


Remember: TCP Slow Start

Systems@ETH Zürich

- Congestion window (CW)
  - Number of bytes in TCP that can be transmitted without waiting for the ACK (CW always smaller than receiver window, flow ctrl)
  - Initially set to 1 TCP segment
- SSThresh
  - Initially set to 64 KB
- TCP congestion control:
  - After all ACKs corresponding to one CW have been received (typically after one RTT), the window is doubled
    - slow start (actually quite fast)
  - If CW >= SSThresh increase CW by 1 TCP segment after all ACKs corresponding to one CW have been received
    - linear increase (congestion avoidance)
  - On a timeout: Set SSThresh to half of the current CW, then set CW back to 1K

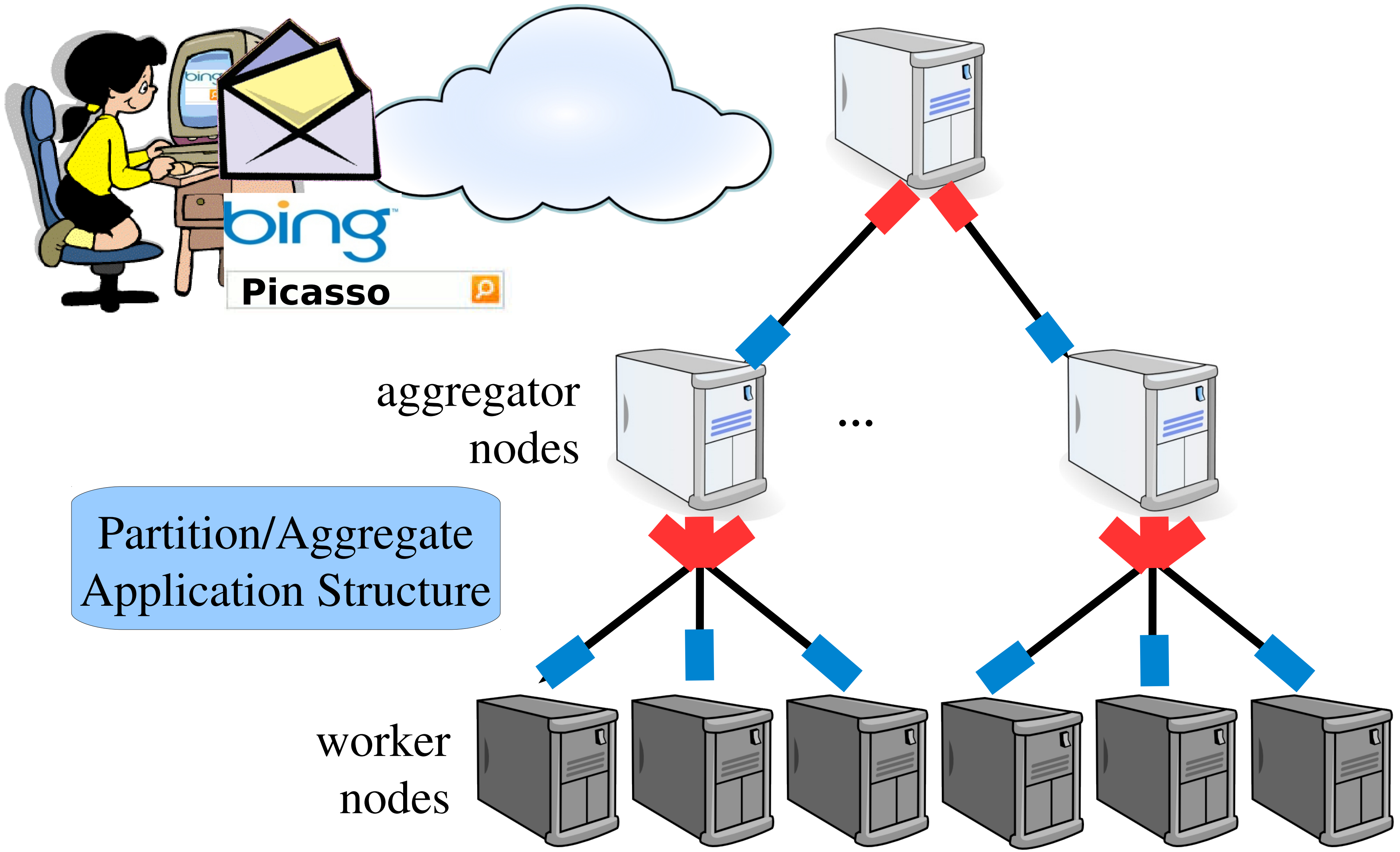
Example: Slow start



The Partition/Aggregate Pattern

How does search work?

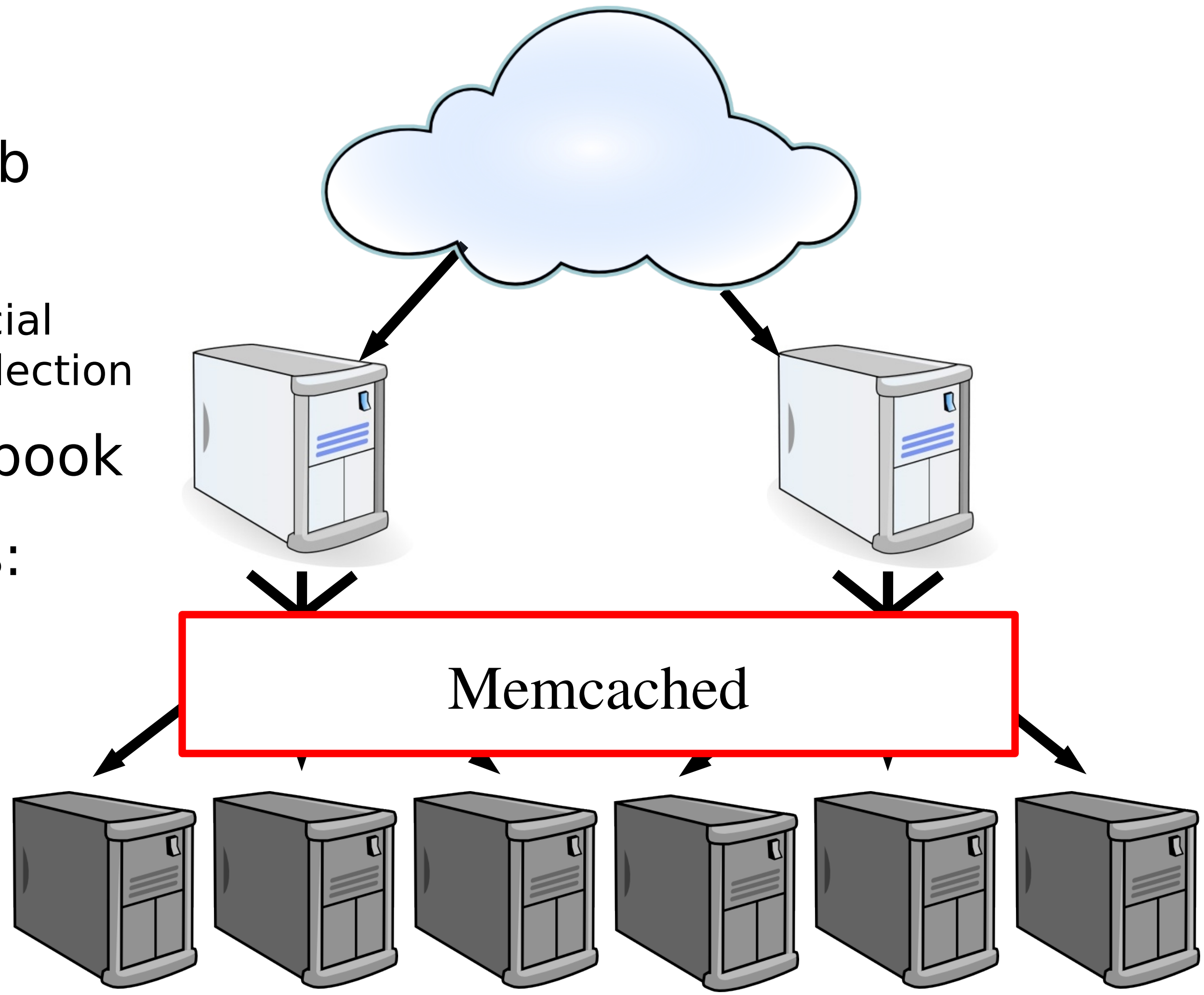
How Does Search Work?



source: stanford CS244

Partition / Aggregate

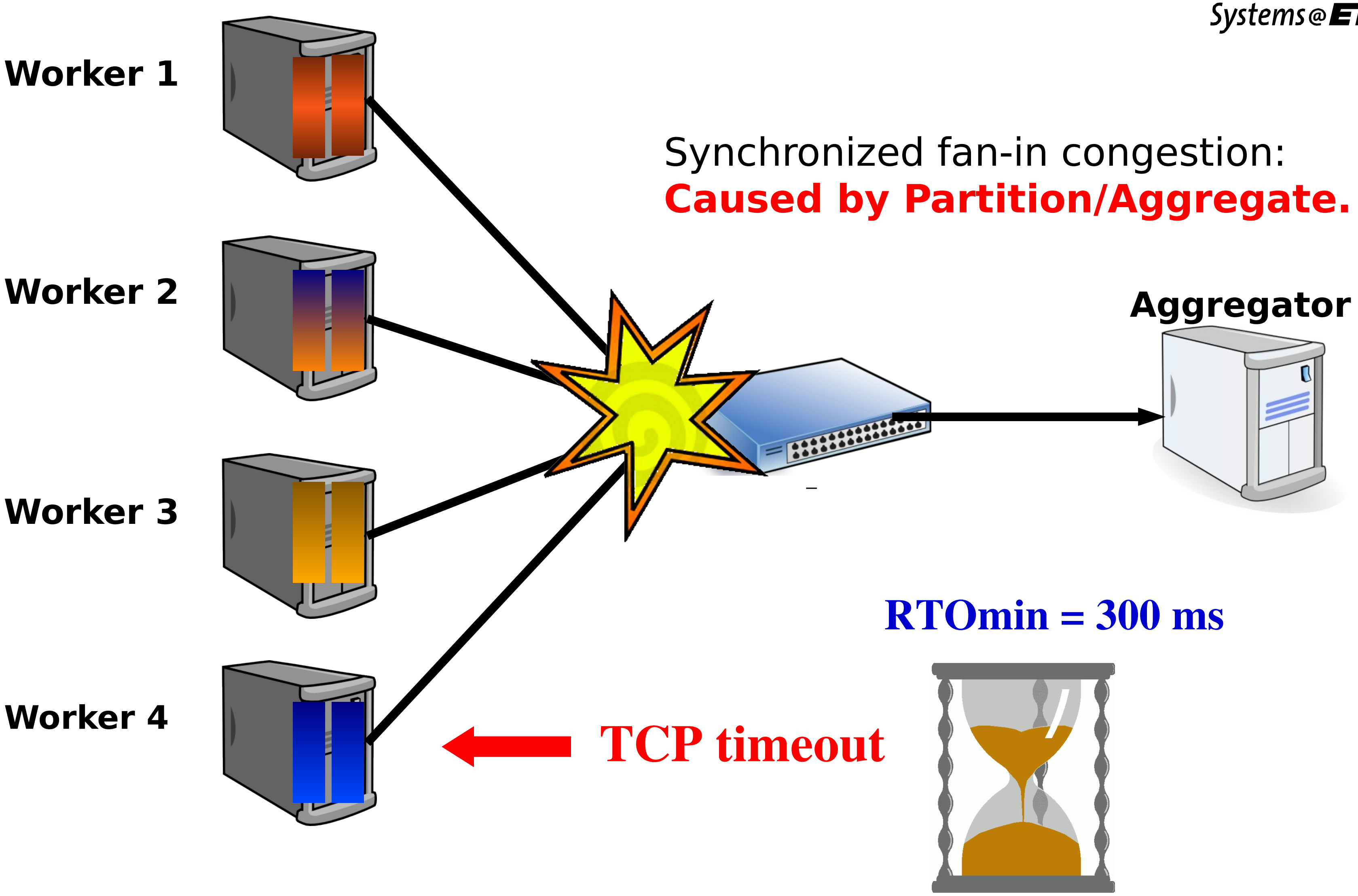
- Foundation of many large web applications
  - Web-search, social networks, ad selection
- Example: Facebook
  - Aggregators: web servers
  - Workers: Memcached servers



source: stanford CS244

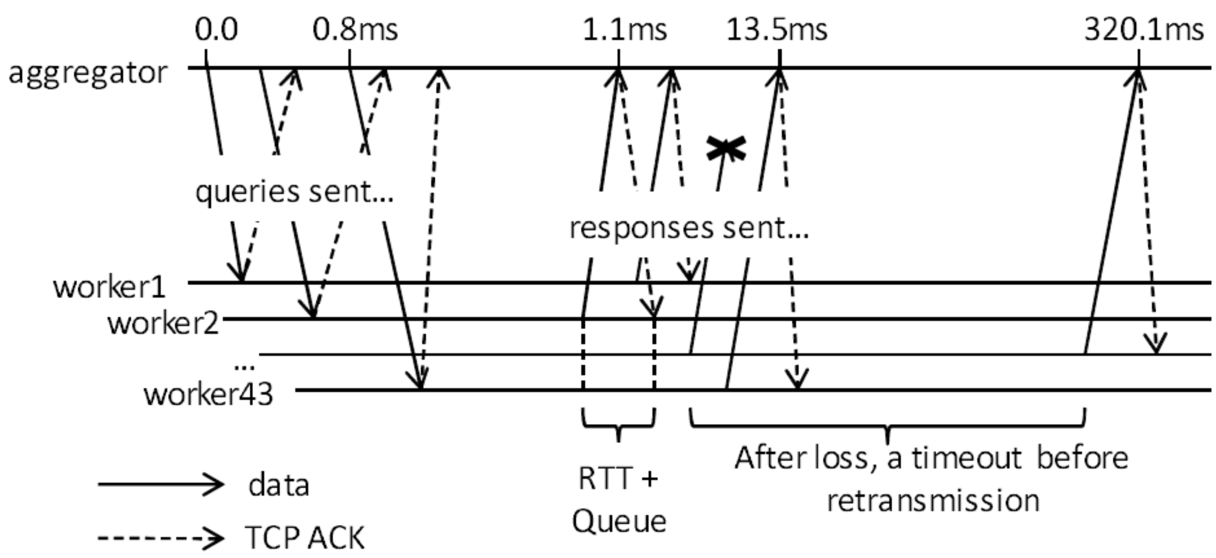
Incast

Problem: TCP Incast

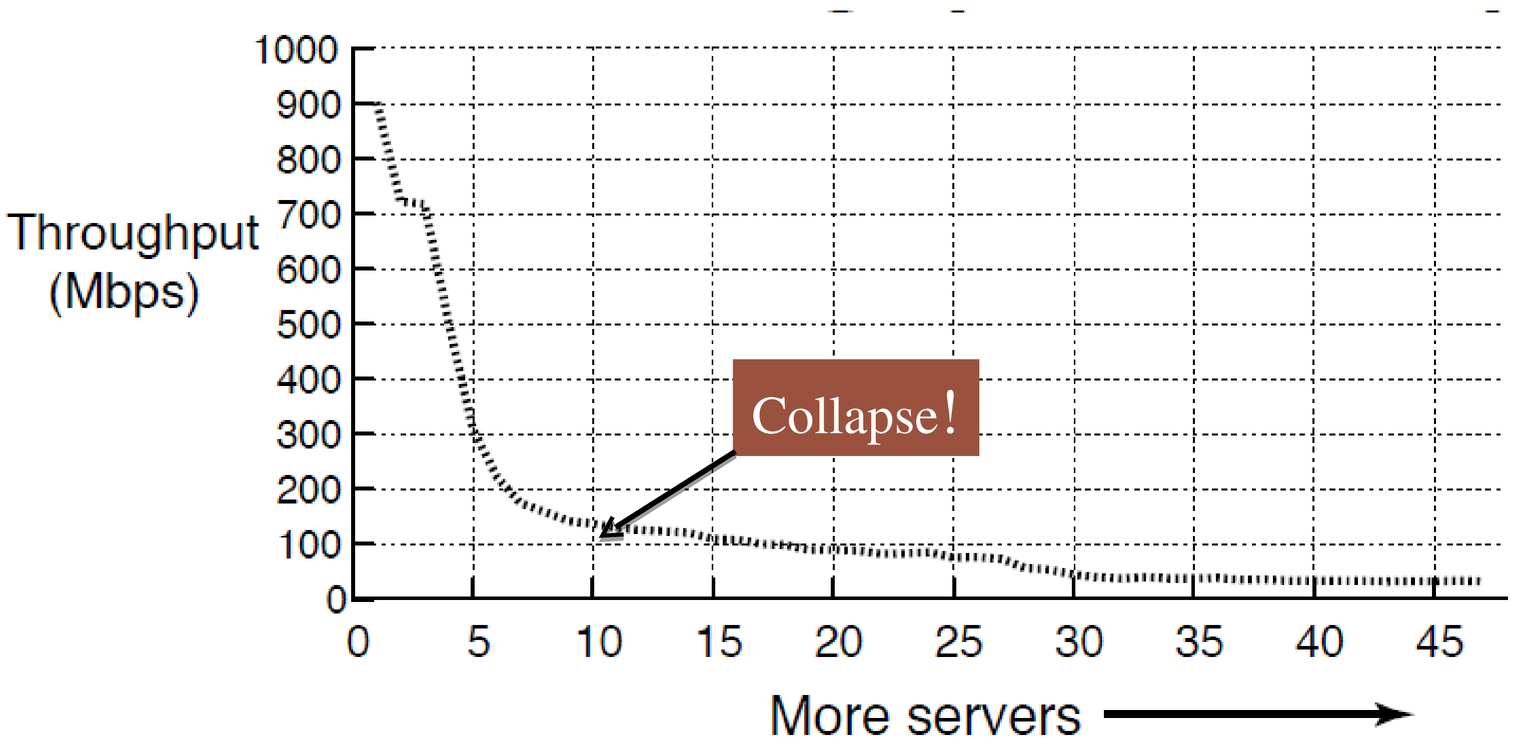


TCP Incast (2)

- Incast event measured in a production environment
  - Request forwarded in over 0.8 ms (800 microseconds)
  - All but one response returning in 12.4 ms
  - Retransmission after RTO: 300 ms



Throughput collapse



Cause of throughput collapse: coarse-grained TCP timeouts

Approach: Fine-grained timeouts

Systems@ETH zürich

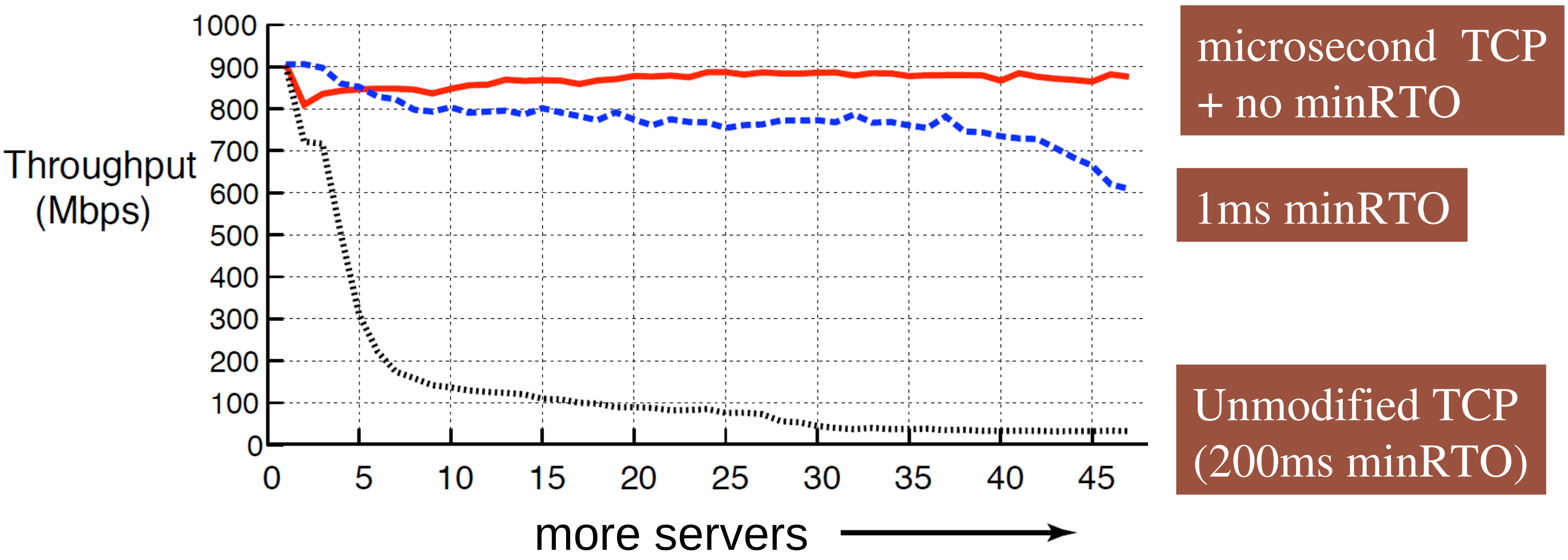
- Roundtrip timeout typically set based on measured RTT+X, with RTO >= RTO_min
- Two problems:
  - Most Linux TCP implementations do not measure RTT as fine granular as needed for datacenters (Linux jiffies updated 250-1000 per second)
  - RTO_min typically too large

Scenario	RTT
WAN	100ms
Datacenter	<1ms
SAN	<0.1ms

OS	TCP RTO_{min}
Linux	200ms
BSD	200ms
Solaris	400ms

- Idea:
  - Reduce RTO_min
  - Measure RTT using high-resolution timers in us granularity

Lower timeout helps



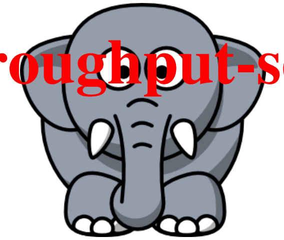
✓ High throughput for up to 47 servers

Data Center TCP (DCTCP)

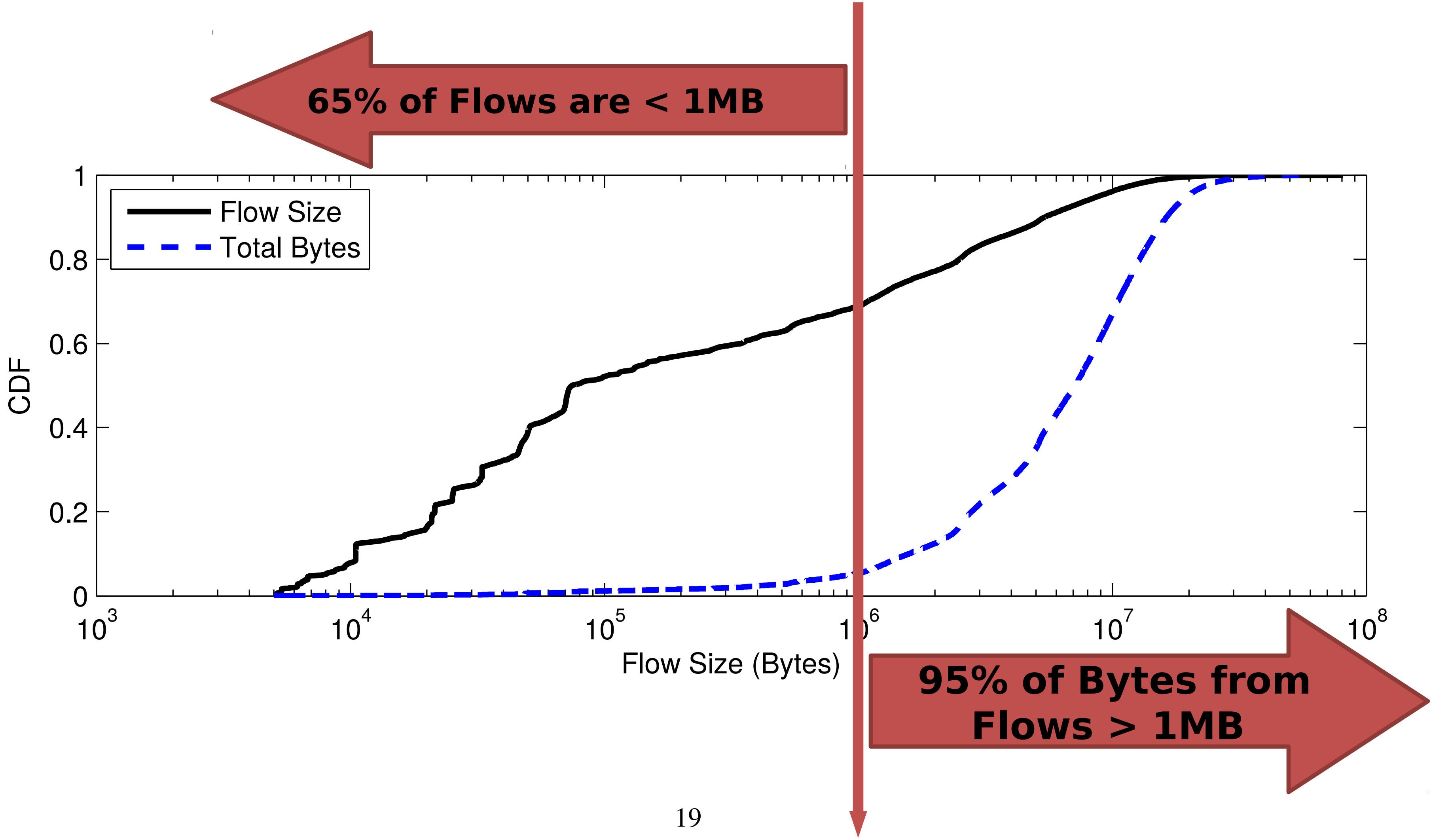
Datacenter Workloads

- Mice & Elephants
  - Short messages (50KB-1MB) (query, coordination, control state) → **Delay-sensitive**

  - Large Flows (1MB-100MB) (data update, backup) → **Throughput-sensitive**

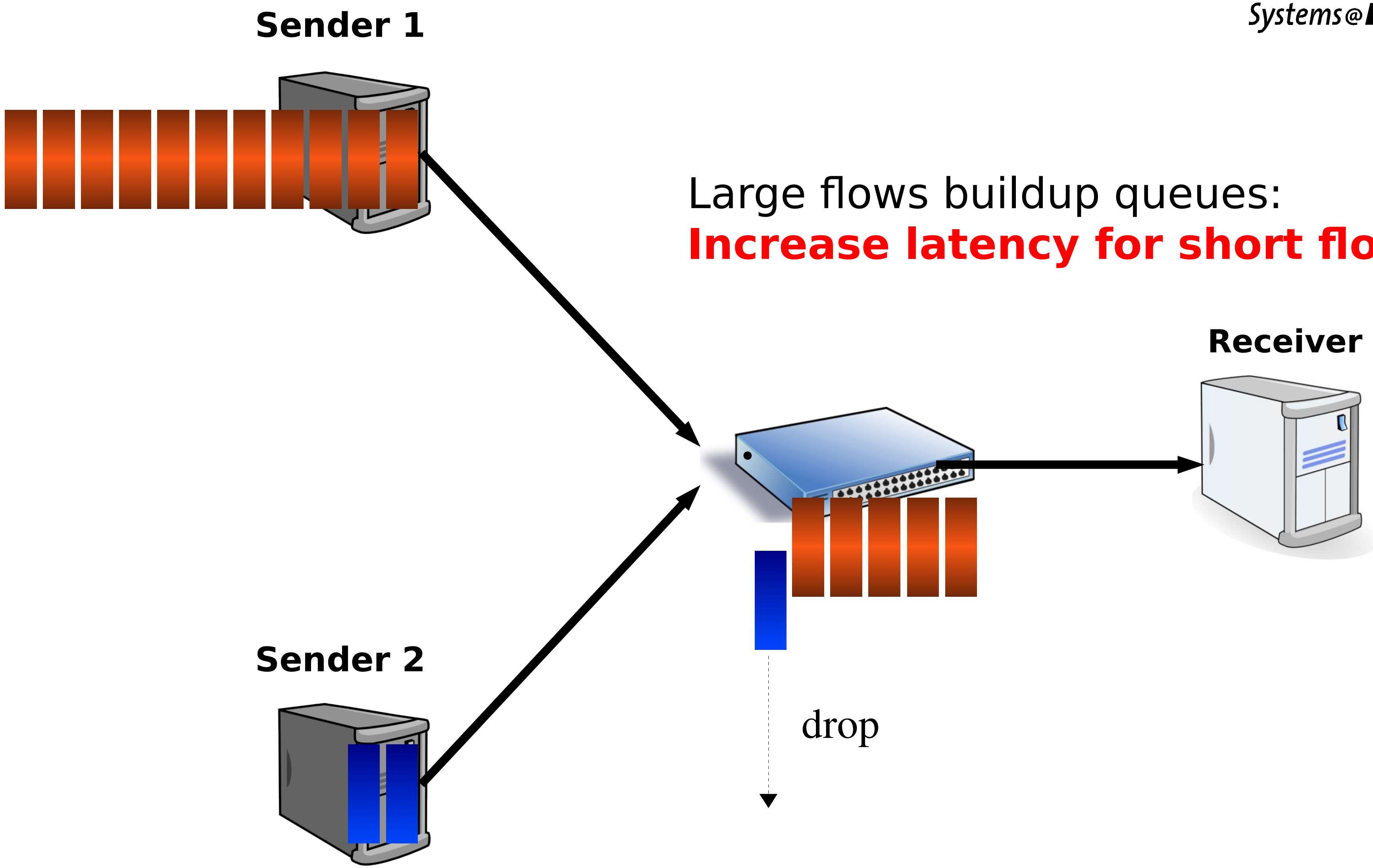


Flow size


65% of Flows are < 1MB
Flow Size
Total Bytes
CDF
0
0.2
0.4
0.6
0.8
1
10^3
10^4
10^5
10^6
10^7
10^8
Flow Size (Bytes)
95% of Bytes from Flows > 1MB

Queue Buildup

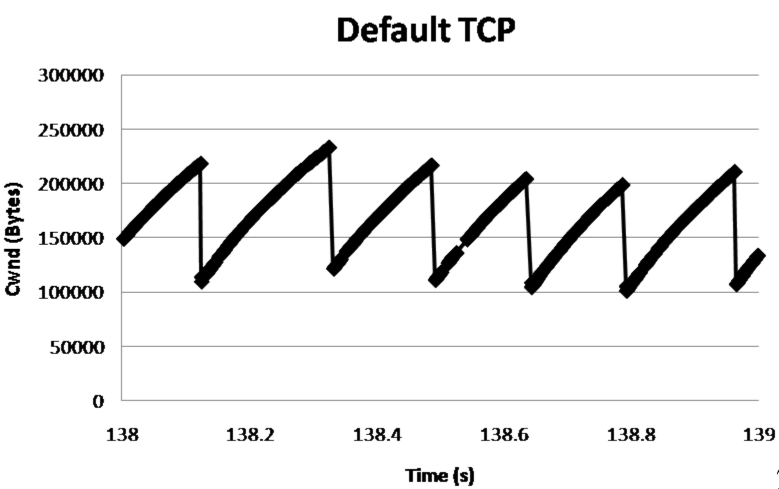
Large flows buildup queues:
Increase latency for short flows.

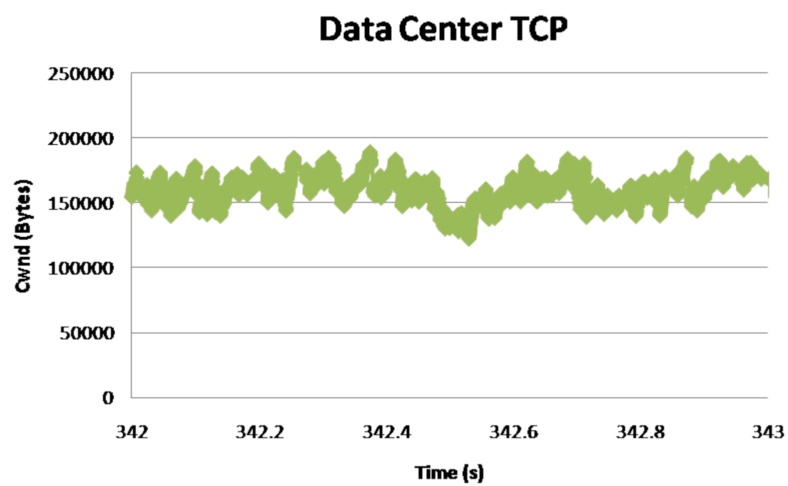


Approach: Datacenter TCP

- Datacenter TCP (DCTCP):
  - Mark packets in switches using Explicit Congestion Notification (ECN) if they experience congestion
  - Scale the TCP window down proportionally to the number of packets with ECN bit set

ECN Marks	TCP	DCTCP
1 0 1 1 1 1 0 1 1 1	Cut window by 50%	Cut window by 40%
0 0 0 0 0 0 0 0 0 1	Cut window by 50%	Cut window by 5%



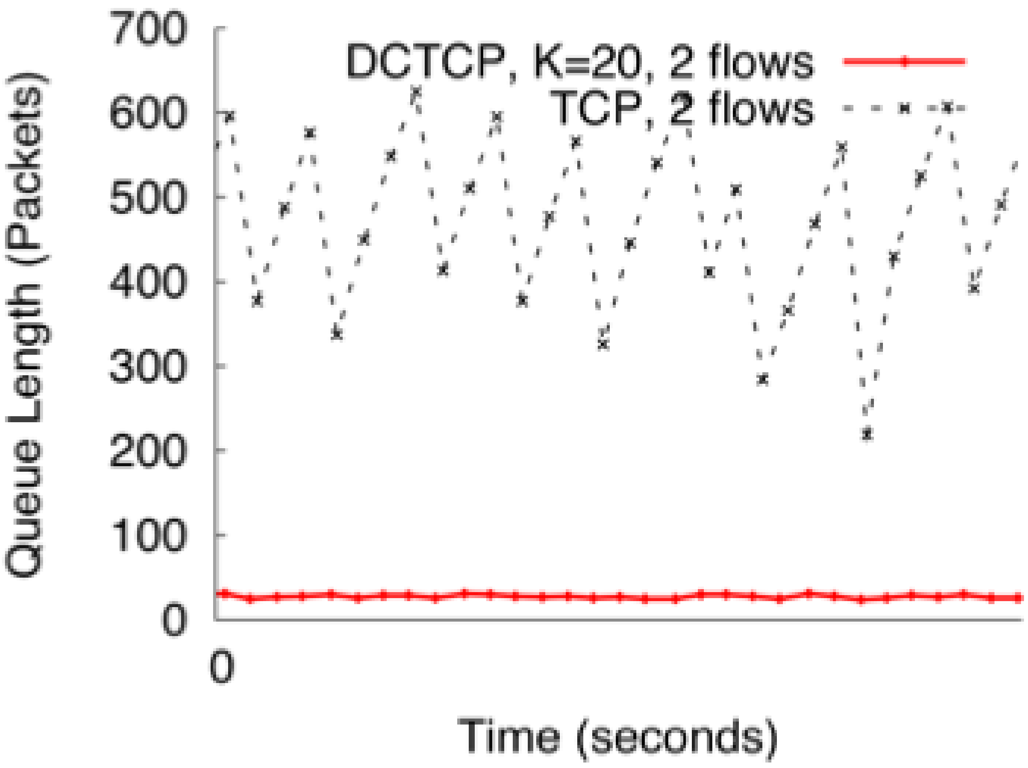


DCTCP Algorithm

- Switch-side:
  - Mark packets if Queue length > K using ECN bit
- Receiver-side:
  - Echo bit back to sender with delayed ACKs
- Sender-side:
  - Maintain running average ***a*** of fraction of packets marked (value of 'a' between 0 and 1)
    - $a = (1-g)*a + g*F$
      F: fraction of packets marked in last window
      $0 > g < 1$: weight given to new samples

    - a close to 0 means low congestion
    - a close to 1 means high congestion
  - Window decrease in case of ECN-marked ACK: $w = w \times (1-a/2)$

DCTCP in Action

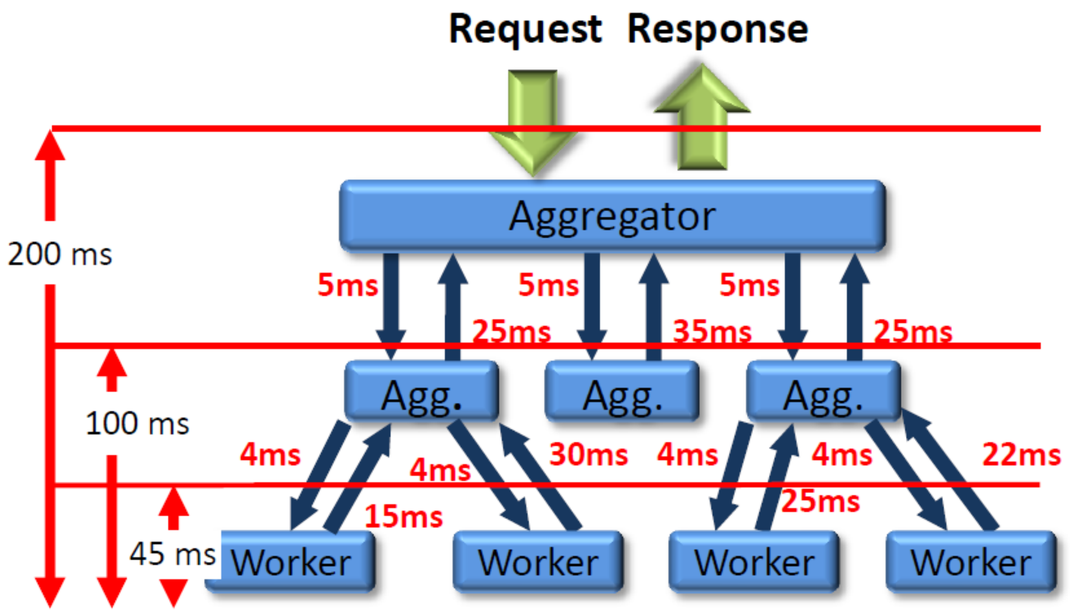


- DCTCP achieves full throughput (not shown in Figure) while taking up a very small footprint in the switch

Deadline-aware TCP

User-facing online services

Partition/aggregate workflow



Application SLAs

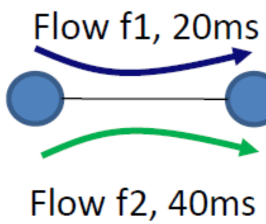
Cascading SLAs

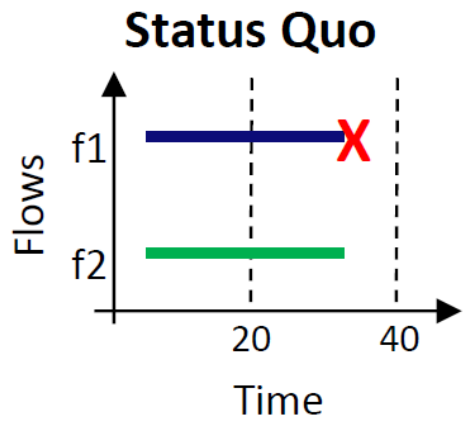
Network SLAs

Flow deadlines

A flow is useful **if and only if** it satisfies its deadline

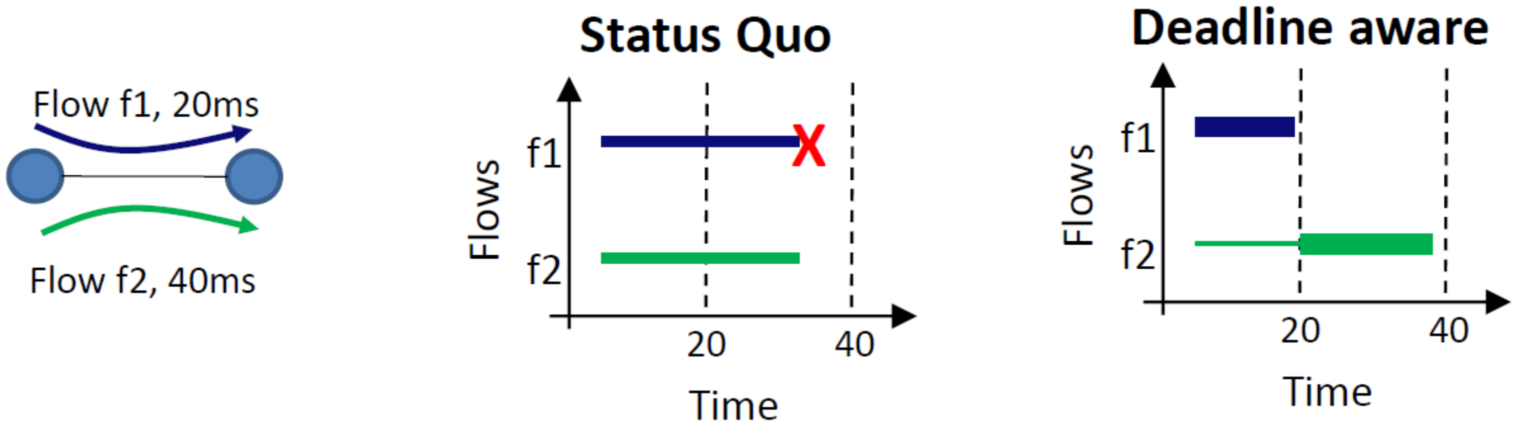
Limitations of fair sharing (1)





- Flows f1 and f2 get a fair share of bandwidth
- Flow f1 misses its deadline (incomplete response to user)

Limitations of fair sharing (1)

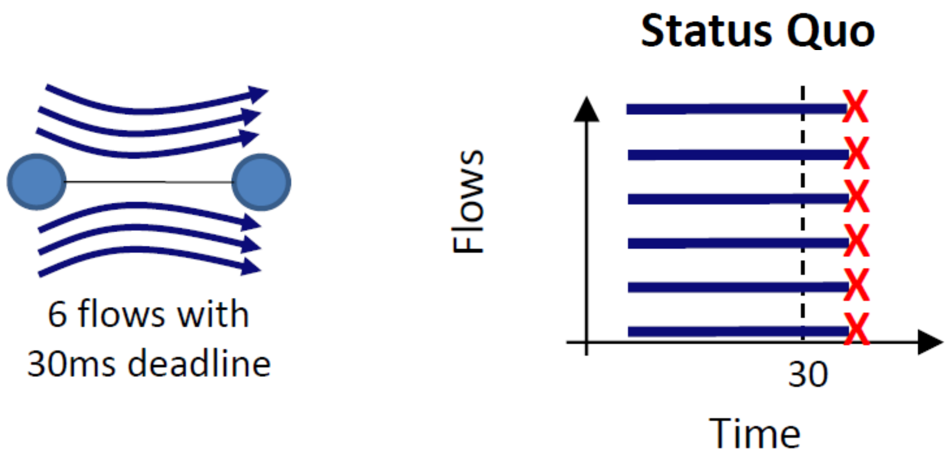


- Flows f1 and f2 get a fair share of bandwidth
- Flow f1 misses its deadline (incomplete response to user)

Case for unfair sharing:

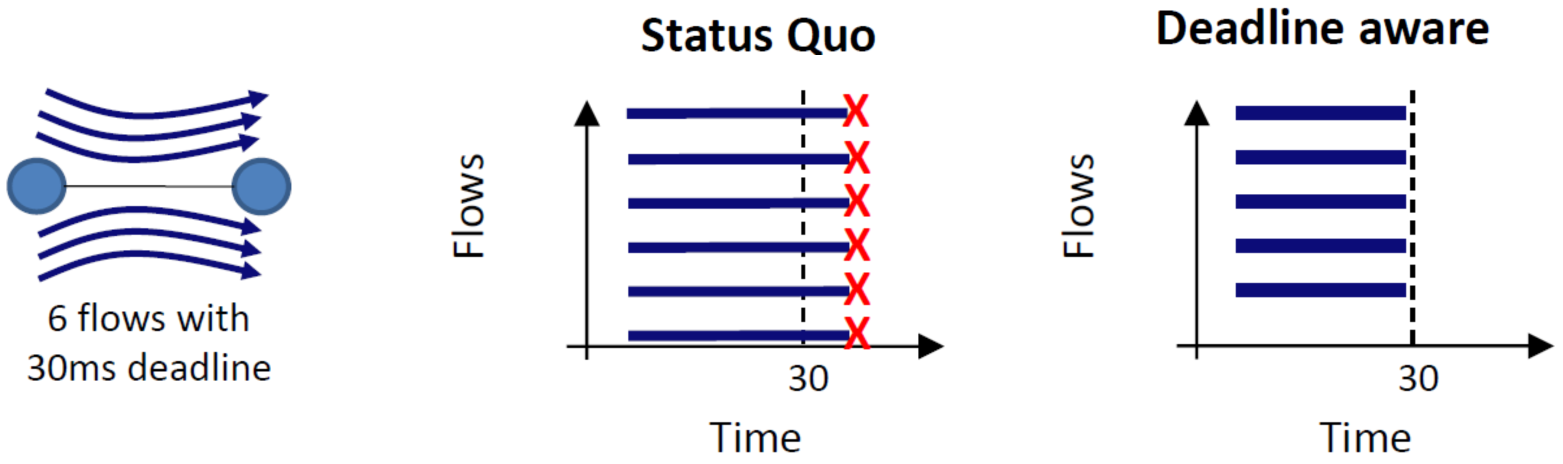
- Flows get bandwidth in accordance to their deadlines
- Deadline awareness ensures both flows satisfy deadlines

Limitations of fair sharing (2)



- Insufficient bandwidth to satisfy all deadlines
- With fair share, all flows miss the deadline (empty response)

Limitations of fair sharing (2)



- Insufficient bandwidth to satisfy all deadlines
- With fair share, all flows miss the deadline (empty response)

Case for flow quenching:

- With deadline awareness, one flow can be quenched
- All other flows make their deadline

D^3: Deadline driven delivery

Systems@ETH zürich

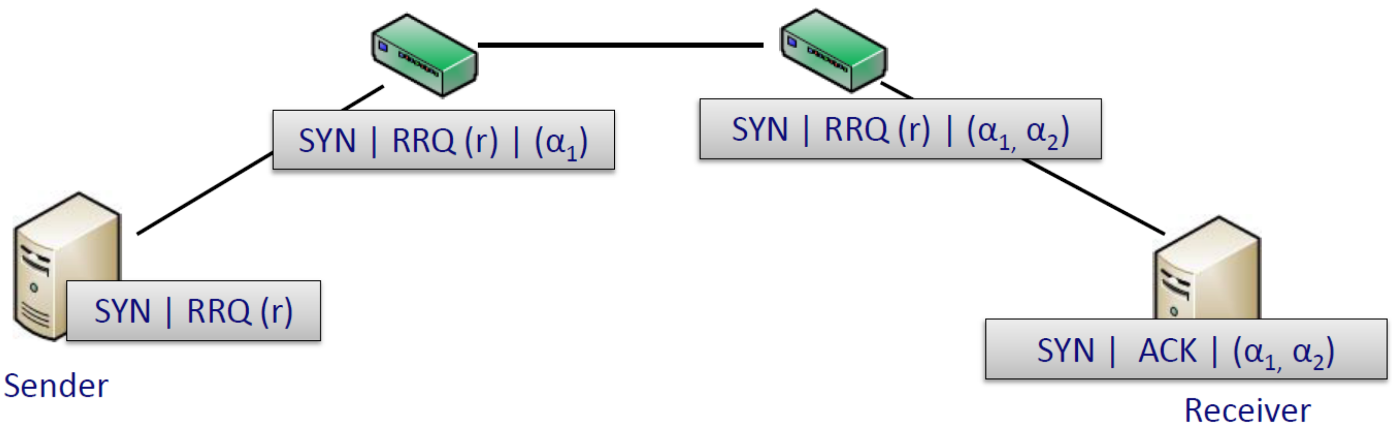
- Main idea: make the network aware of flow deadlines
  - Prioritize flows based on deadlines

- Key insight:
  - Rate required to satisfy a flow deadline: **r = s / d**

    s: flow size

    d: deadline

How it works (1)

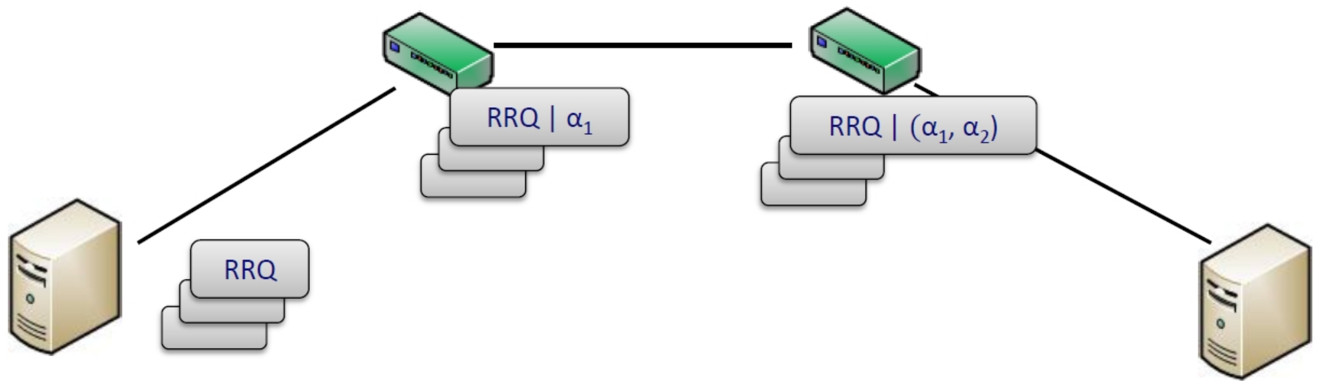


- Application exposes (s,d)
- Desired rate r = s / d
- Routers allocate rates (α) based on traffic load
- Sending rate for next RTT : sr = $\min(\alpha_1,\alpha_2)$

s: flow size
d: deadline
RRQ: rate request
α: allocated rate

How it works (2)

Systems@ETH zürich

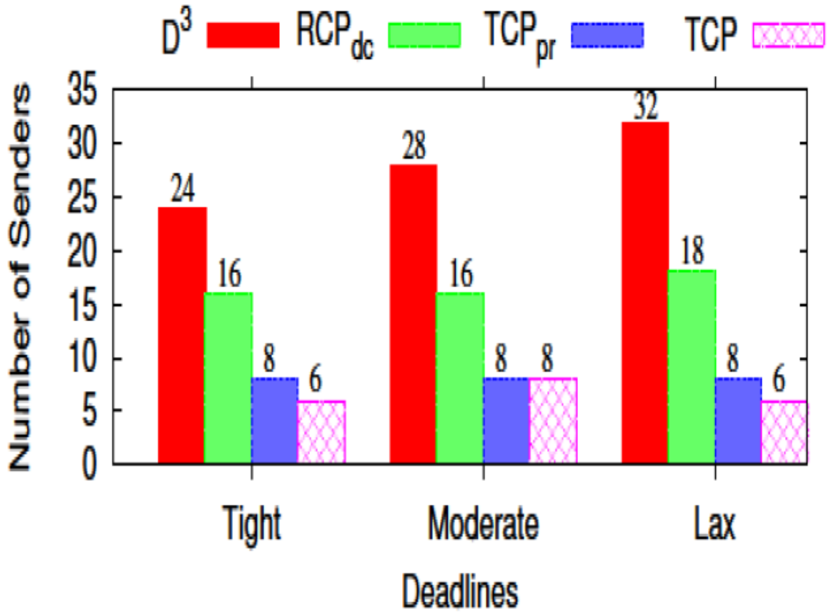


- Application exposes (s,d)
- Desired rate r = s / d
- Routers allocate rates (α) based on traffic load
- Sending rate for next RTT : sr = min(α_1,α_2)

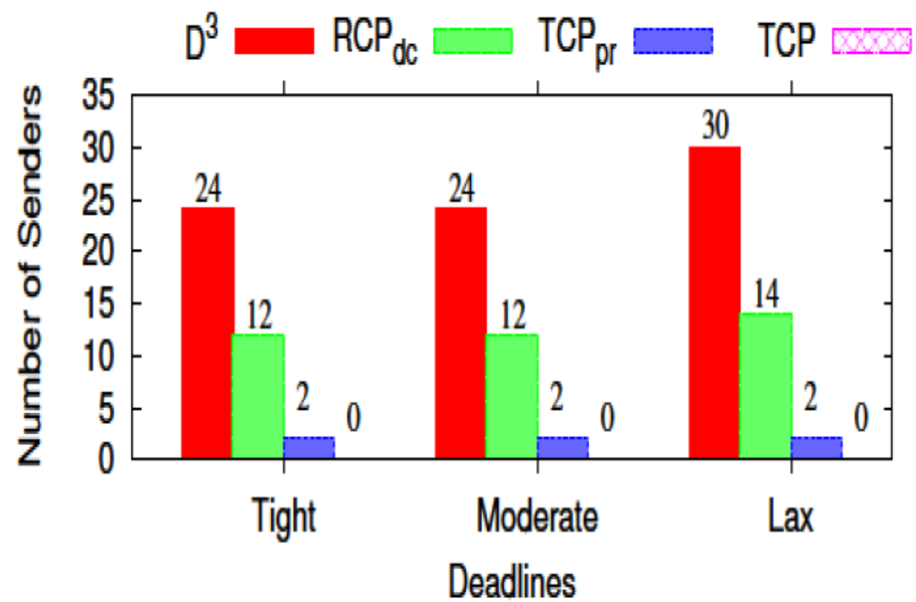
s: flow size
d: deadline
RRQ: rate request
α: allocated rate

- One of the packets contains and updated RRQ based on the remaining flow size and the deadline

Flow microbenchmarks

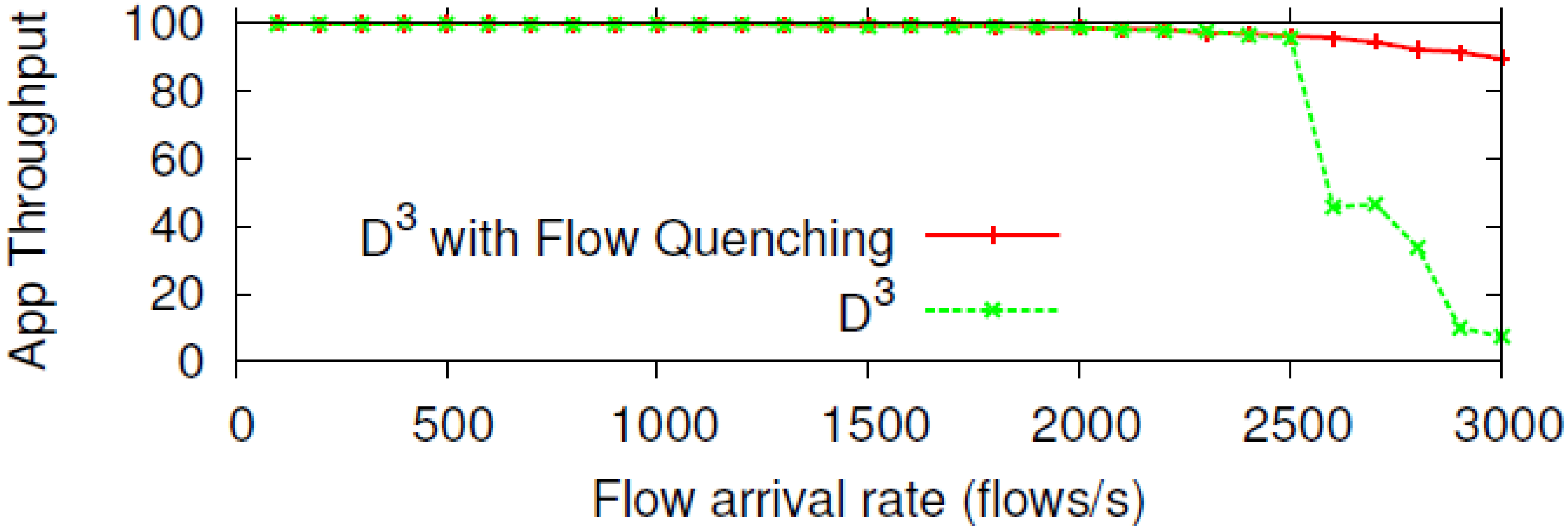


With background flows



- Experiment: multiple workers sending traffic to aggregator (within a single rack)
- How many workers can be supported while satisfying deadlines?
- D^3 can support roughly twice as many workers than RCP (~DCTCP) while satisfying application deadlines

Flow quenching



- Terminate useless flows when:
  - Desired rate exceeds link capacity
  - Deadline has expired

D^3 Summary

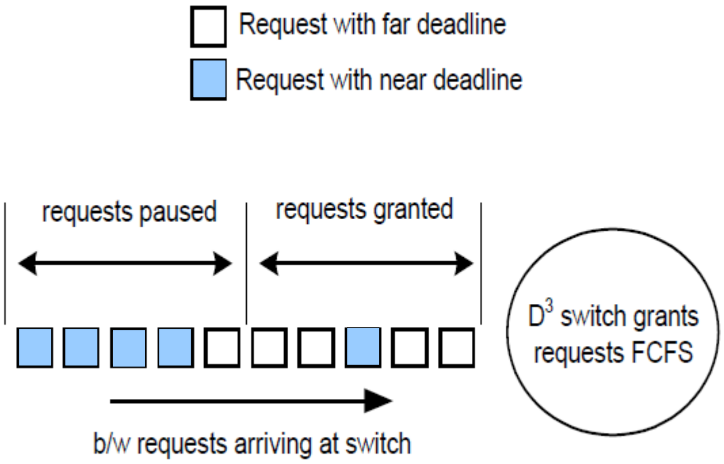
Systems@ETH Zürich

- Traditional TCP flow-sharing leads flows missing deadlines
- D3 allocated rates at switches based on the deadlines of the flows
- D3 can support many more flows with deadlines than TCP

Limitations of D^3

- Needs router support
  - User-space PC-based implementation for paper
- Violates end-to-end argument
- Greedy rate allocation leads to priority inversion

  … and later to missed deadlines



Deadline-aware Datacenter TCP (D^2TCP)

- Key idea:
  - Vary sending rate based on both deadline and extent of congestion
- Built on top of DCTCP
- Per-flow state at end-hosts (not routers/switches)
- Reactive: senders react to congestion
  - No knowledge of other flows

D^2TCP Congestion Avoidance (1)

- Remember: DTCP congestion control
  - TCP window: w = w x (1-a/2)
  - Running average of marked packets: a = (1-g)*a + g*F
    (a ~ 0: low congestion, a ~ 1: high congestion)
- D2TCP extends DTCP to integrate deadline
  - Deadline factor d: larger d implies closer deadline
  - Penalty function: $p = a^d$
  - TCP window: w = w * (1 – p/2) if p > 0
    w = w + 1 if p = 0
- Note:
  - a = 0 => p = 0 => w = w +1 (similar to TCP)
  - a = 1 => p = 1 => w = w/2 (similar to TCP)

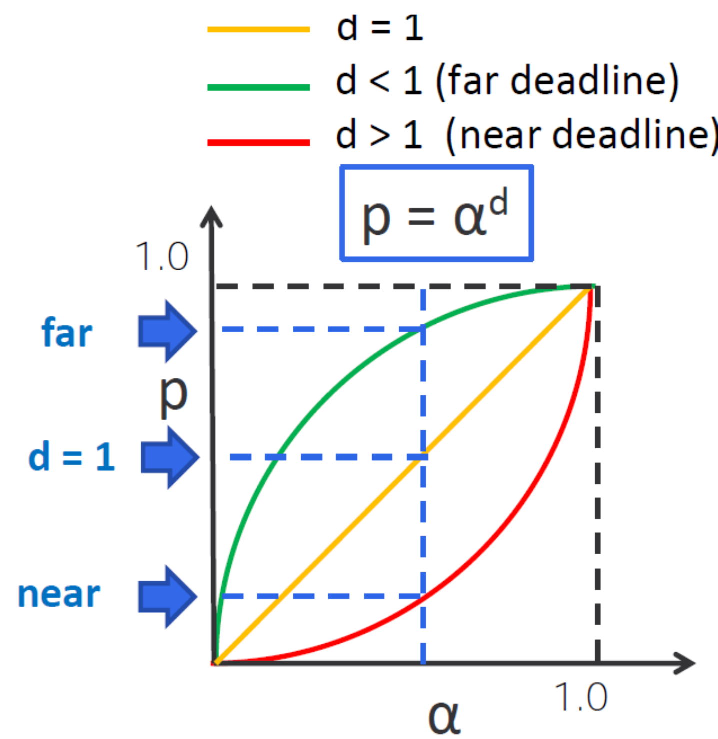
D^2TCP Congestion Avoidance (2)

- d < 1 for far deadline flows

  => p large => shrink window
- d > 1 for near deadline flows

  => p small => retain window
- d = 1 for long lived flows

  => DTCP behavior



Near-deadline flows back off less
while far-deadline flows back off more

How to determine d?

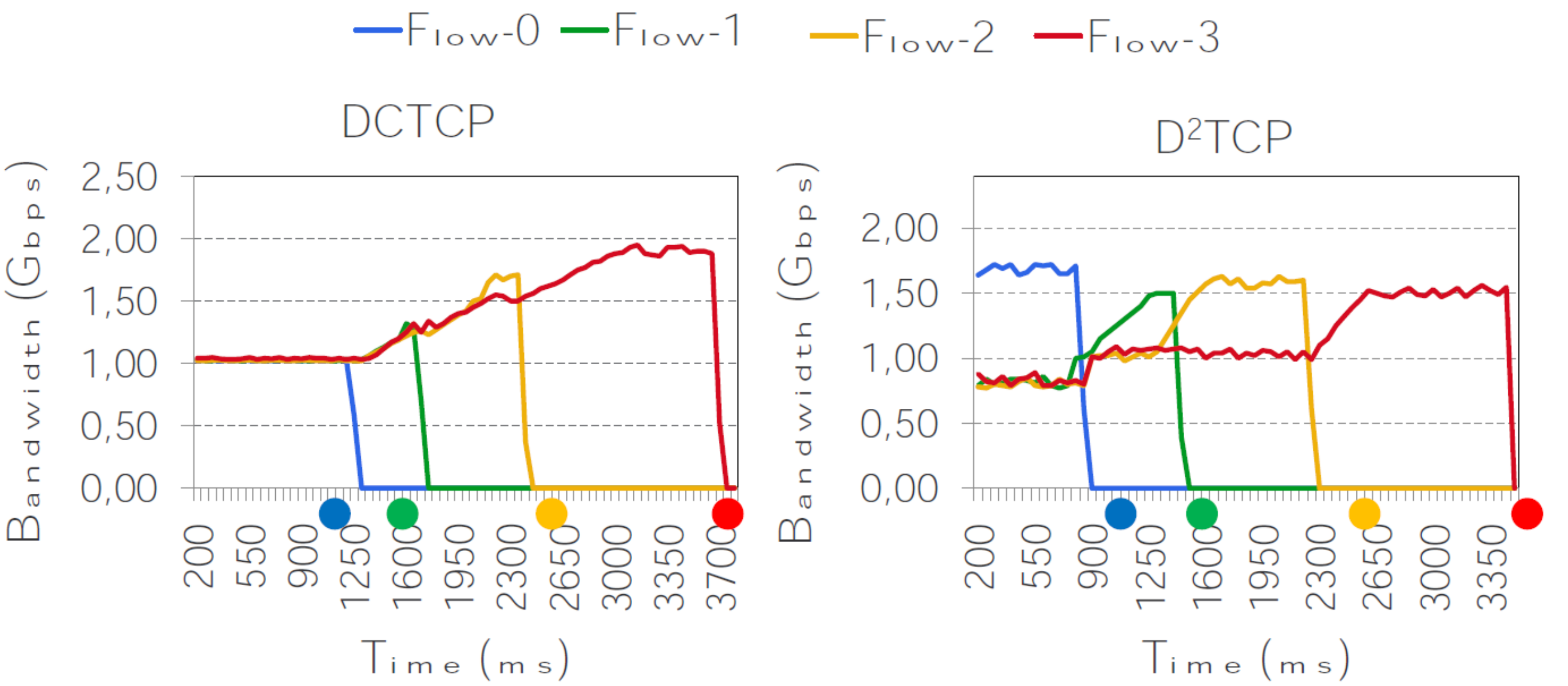
- $d = T_c / D$

  T_c: time needed for a flow to complete under deadline-agnostic congestion behavior (based on the current window w)

  D: remaining time until deadline expires

- Flow is on track: $T_C \approx D \Rightarrow d \approx 1$ ✓
- Flow is about to miss deadline: $T_C > D \Rightarrow d > 1$ ✓
- Flow is ahead of deadline $T_C < D \Rightarrow d < 1$ ✓

D²TCP versus DCTCP

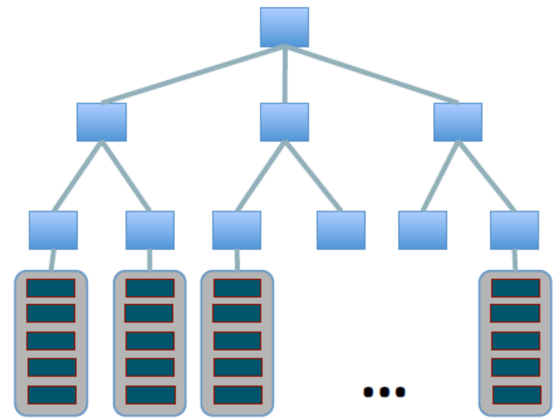


- Flow sizes: 150MB, 220MB, 350MB, 500MB
- Flow deadlines: 1000ms, 1500ms, 2500ms, 4000ms
- DTCP: all flows get same b/w irrespective of deadline
- D²TCP: Near deadline flows get more bandwidth

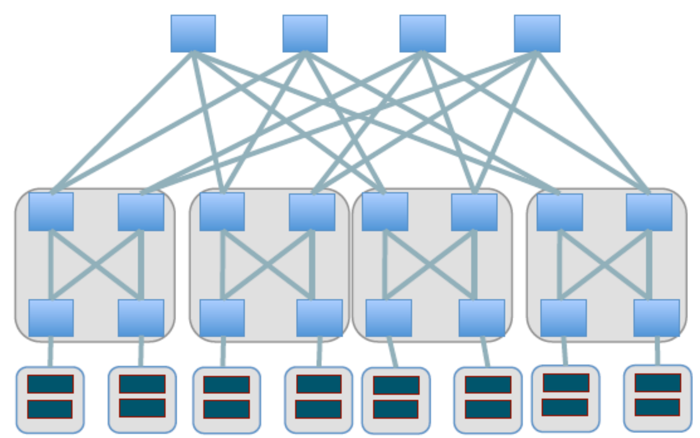
Multipath TCP

Modern datacenters provide many parallel paths

- Traditional topologies are tree-based
  - Poor performance
  - Not fault-tolerant

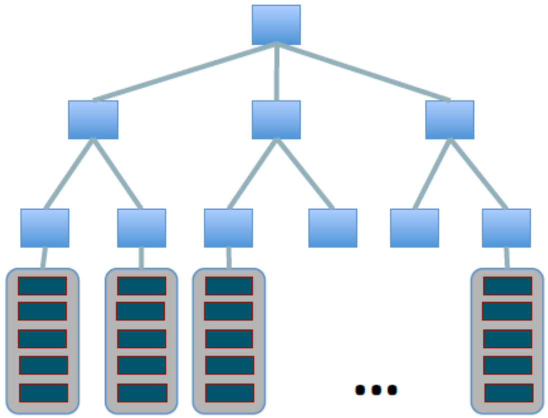


- Shift towards multipath topologies
  - FatTree (Portland, VL2)
  - BCube

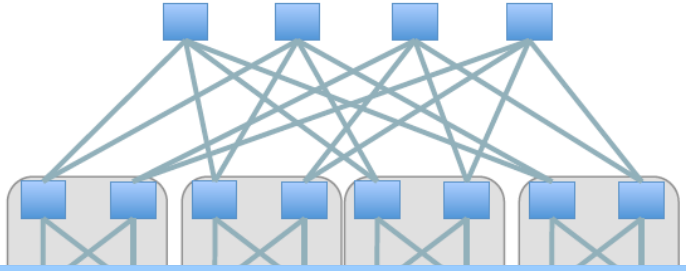


Modern datacenters provide many parallel paths

- Traditional topologies are tree-based
  - Poor performance
  - Not fault-tolerant

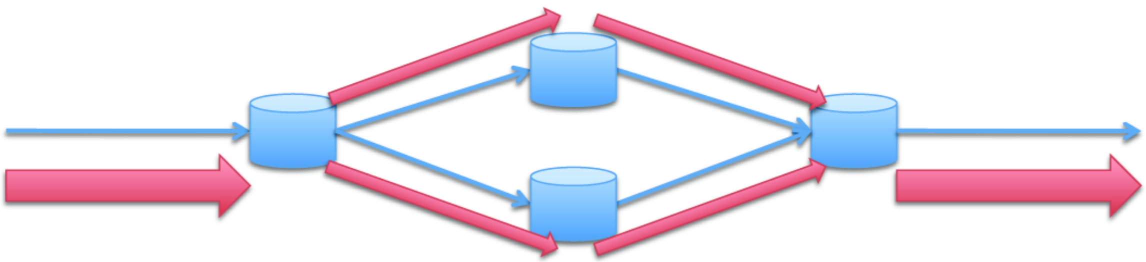


- Shift towards multipath topologies
  - FatTree (Portland, VL2)
  - BCube



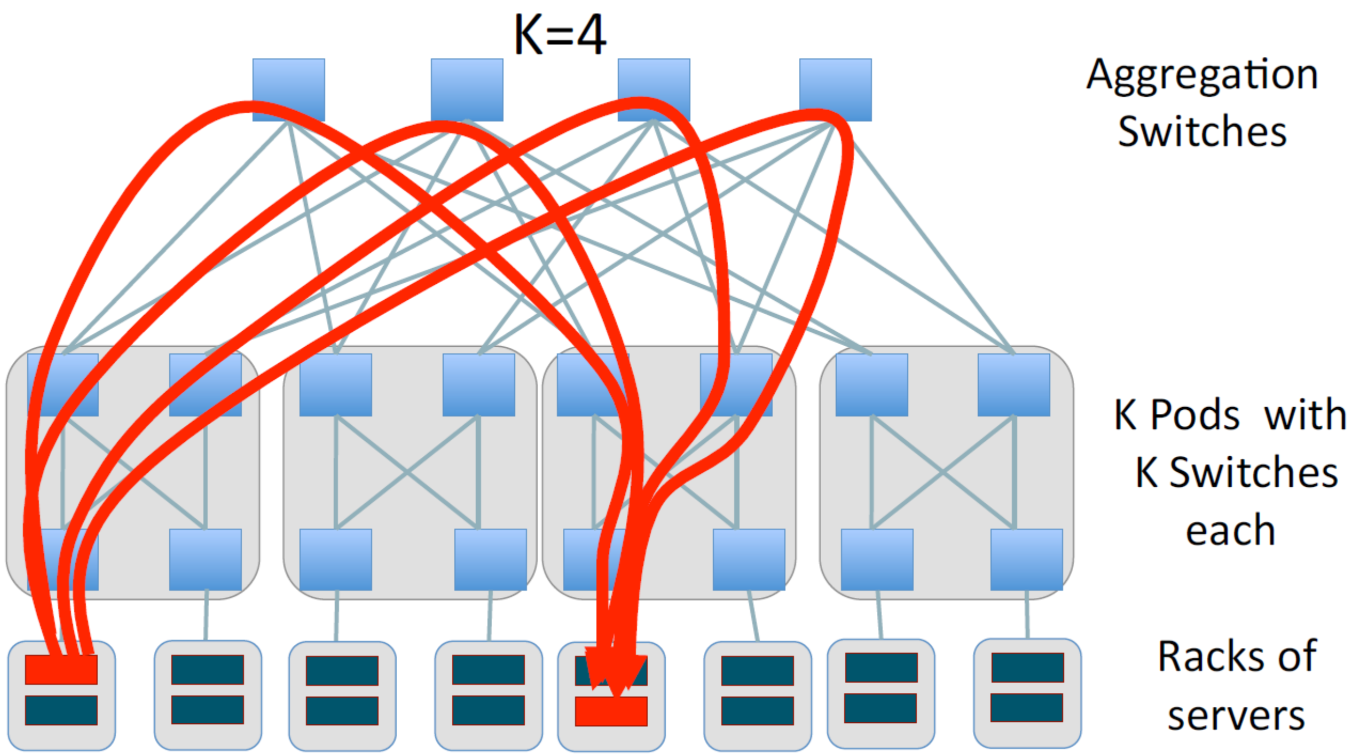
How to effectively use all the bandwidth in a multipath topology?

Equal-cost multipath routing (ECMP)



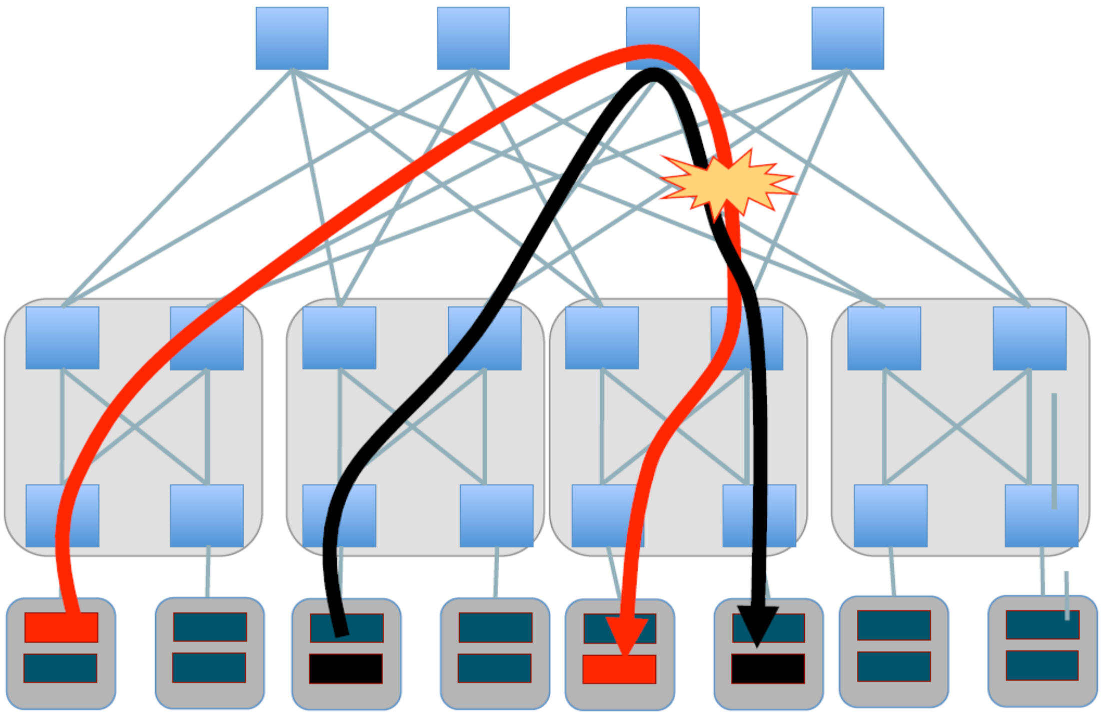
- ECMP
  - Multipath routing strategy that splits traffic over multiple paths for loadbalancing
- Path selection via hashing
  - #buckets = #outgoing links
  - Hash network information (src address/port, dst address/port, protocol type) to select outgoing link: preserves flow affinity
- Why not just round-robin packets?
  - Different RTT per path
  - Different MTUs per path
  - Reordering: triple TCP ACKs, TCP fast retransmit, reduced TCP window

ECMP in Fat Tree Network



Hash flows to different paths

ECMP collision



Multiple flows may hash to same path

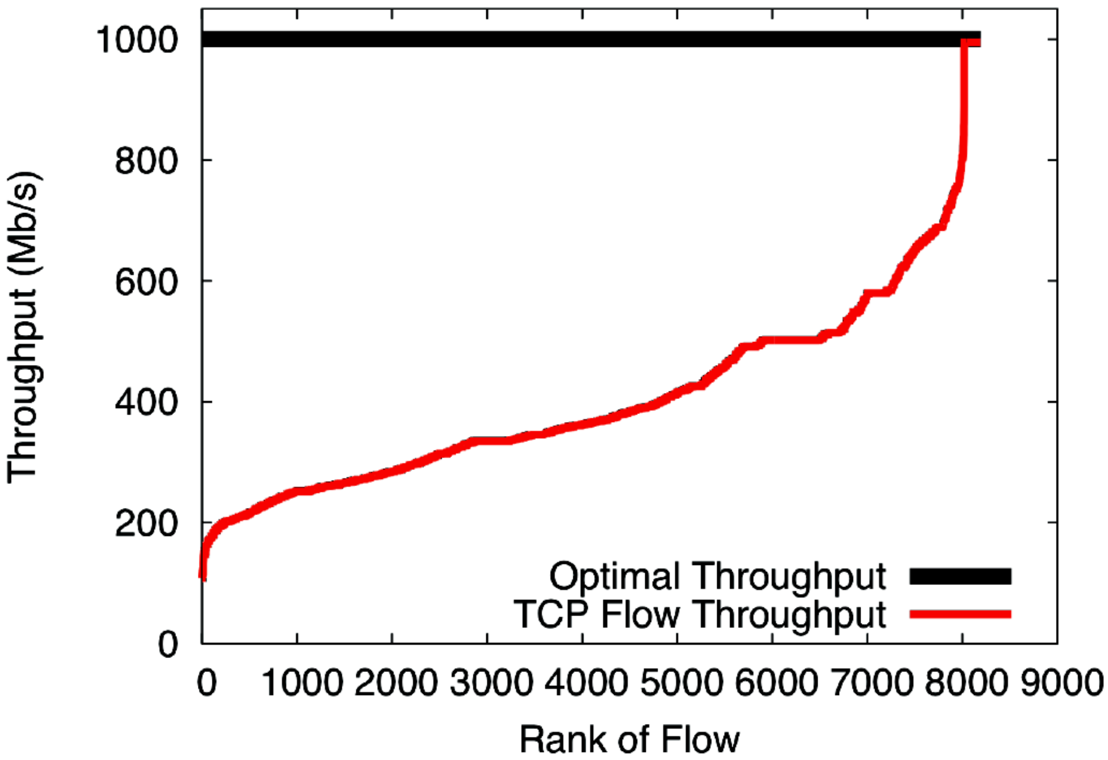
Limitations of ECMP

Systems@ETH Zürich

- ECMP may not utilize the links uniformly
  - Many flows hashed to same link

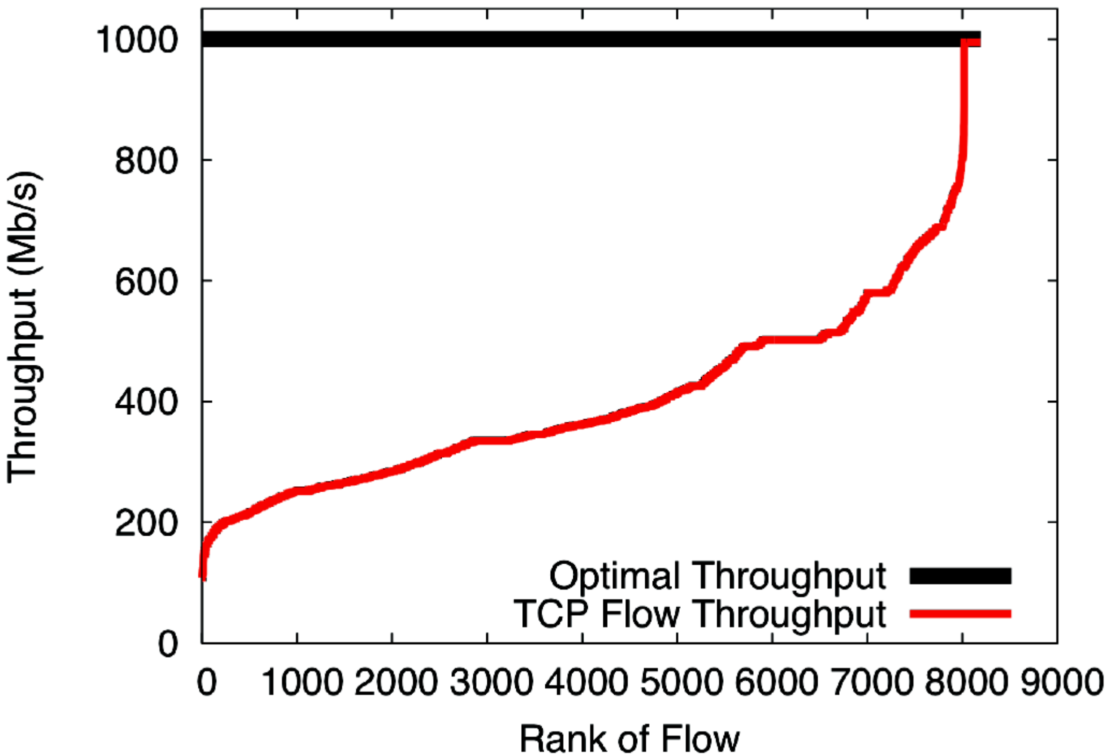
- ECMP is static
  - No knowledge about current traffic on a link

Limitations of ECMP (2)



- 8192 node fat tree toplogy
- Random traffic pattern:
  - Flow: every host chooses a random destination
  - No destination is used twice
- Figure: flows ranked according to their throughput

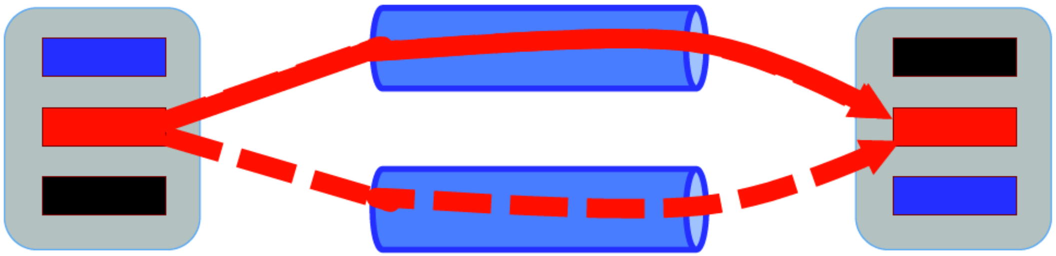
Limitations of ECMP (2)



- 8192 node fat tree toplogy
- Random traffic pattern:

Vast majority of flows does not achieve more than 50% of the throughput possible

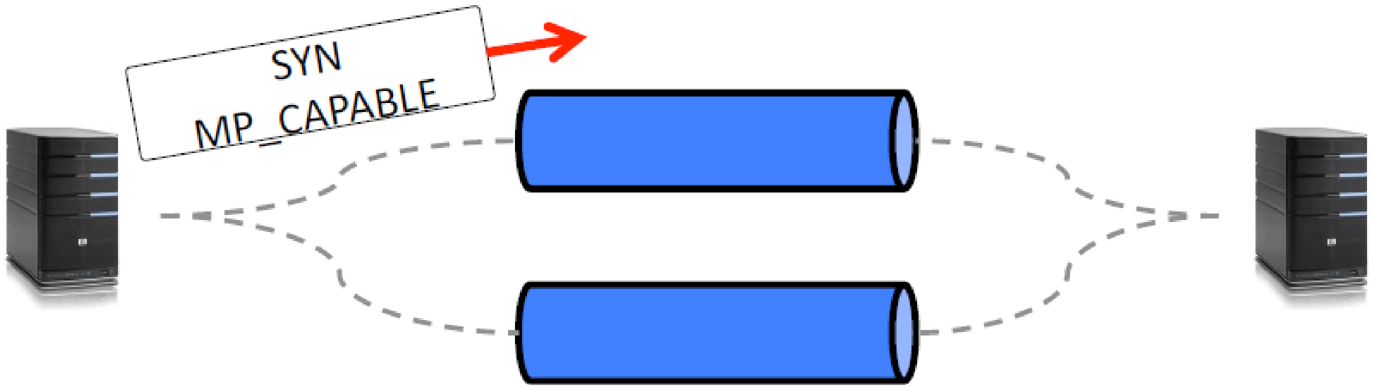
Multipath TCP (MPTCP)



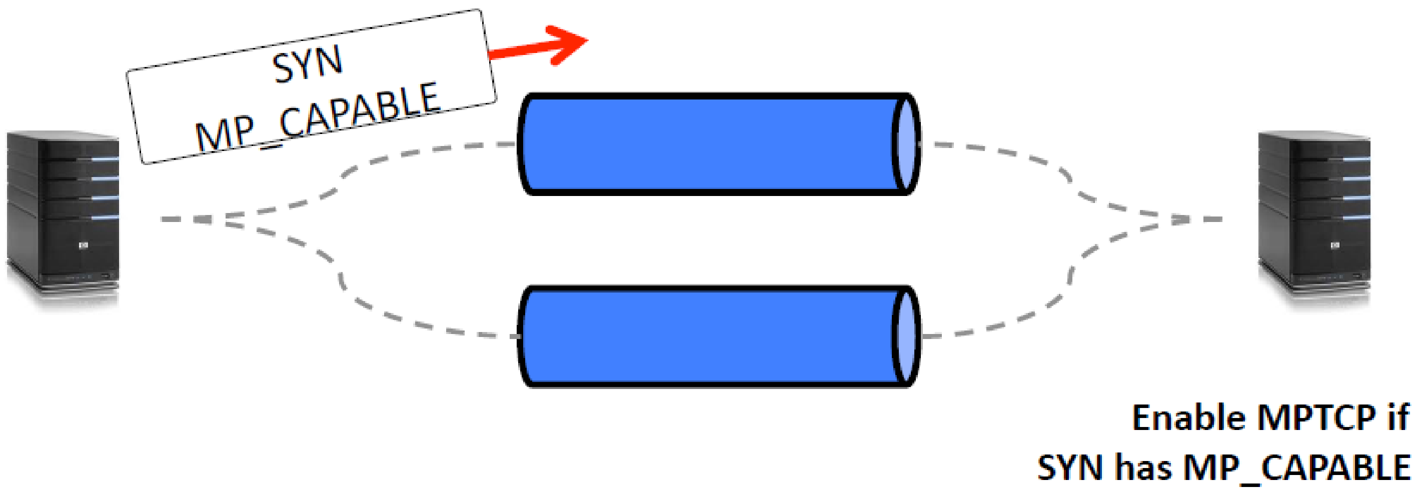
- Instead of using one path for each TCP flow, use many subflows per TPC flow, each on a random path
- Don't worry about collisions
- Just don't send (much) traffic on colliding path
- Improves bandwidth (aggregation), fairness and robustness

MTCP: Connection Management

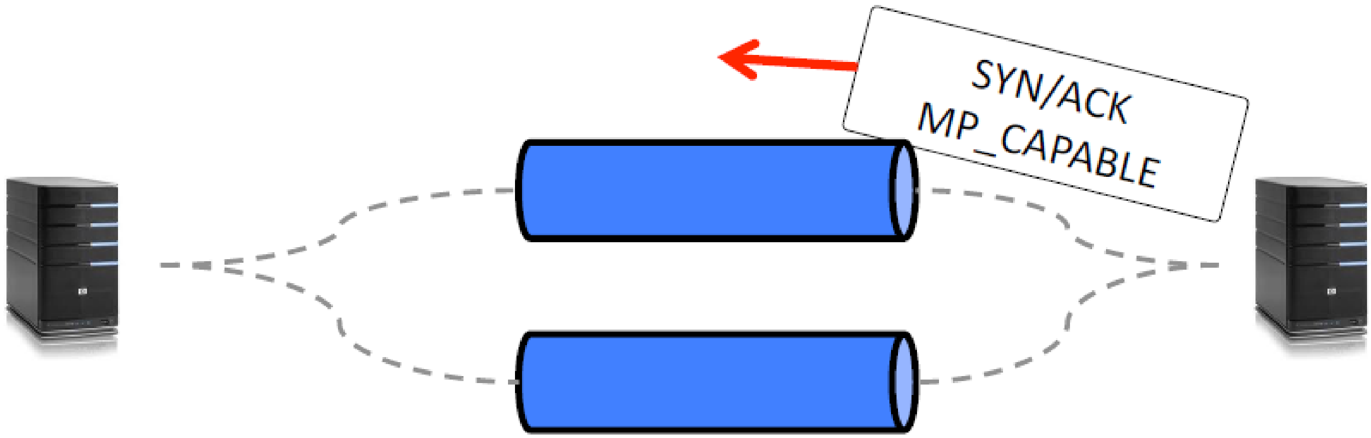
MTCP: Connection Management



MTCP: Connection Management



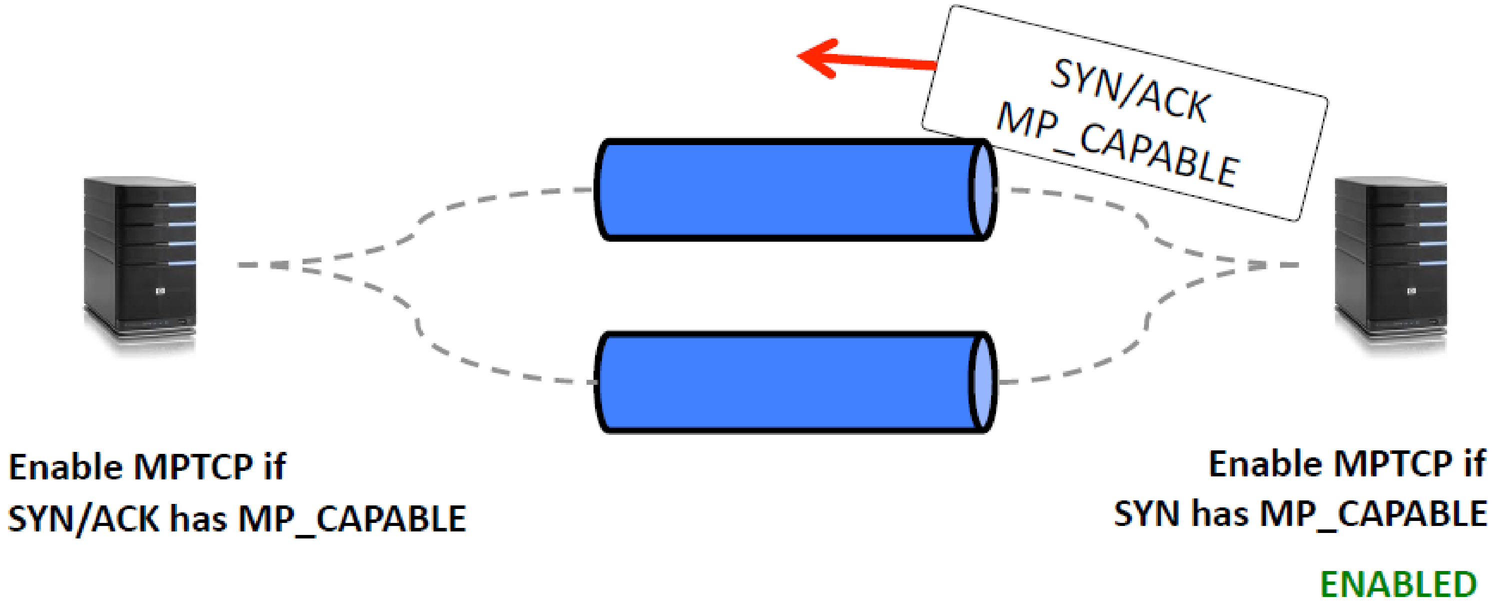
MTCP: Connection Management



Enable MPTCP if
SYN has MP_CAPABLE

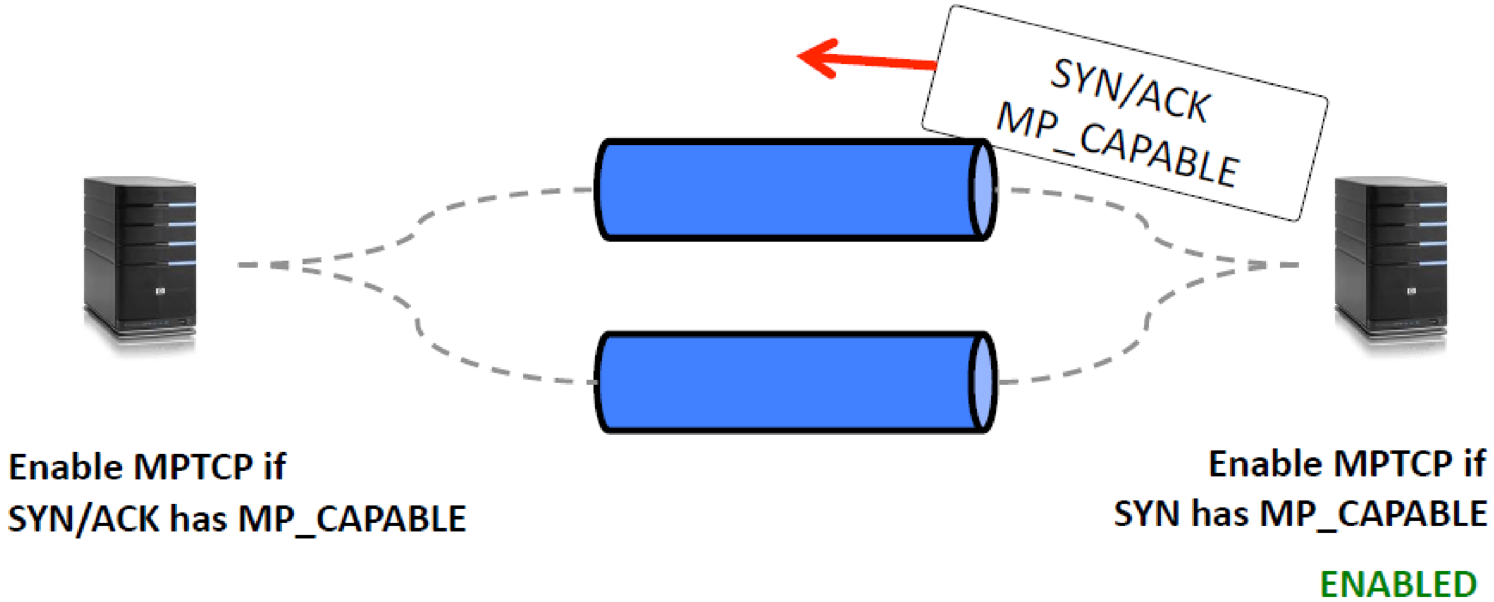
ENABLED

MTCP: Connection Management



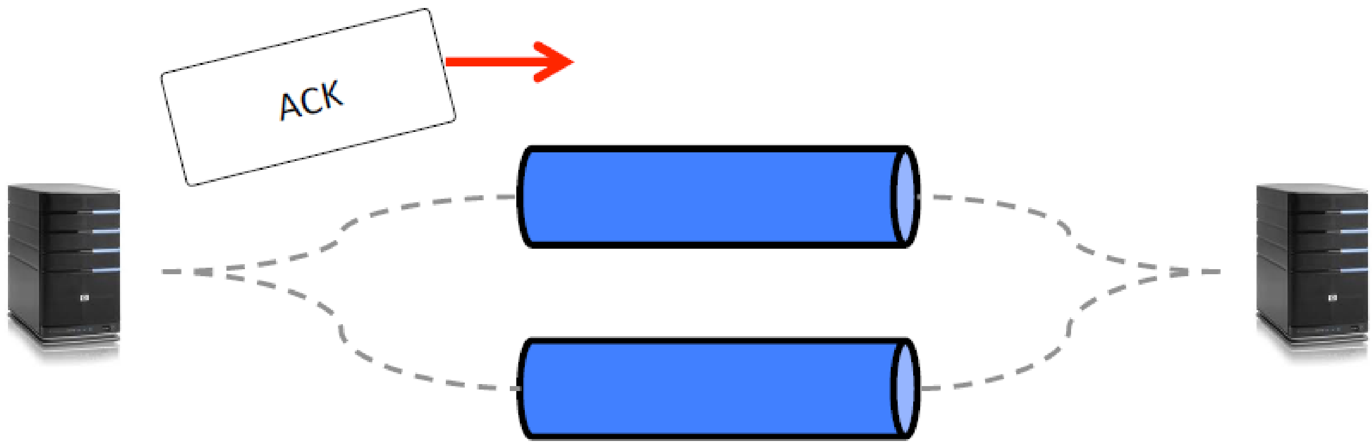
client learns about
additional interfaces
of server

MTCP: Connection Management



Works in data centers, problem when using MPTCP across the Internet: 6% of access networks remove unknown options

MTCP: Connection Management



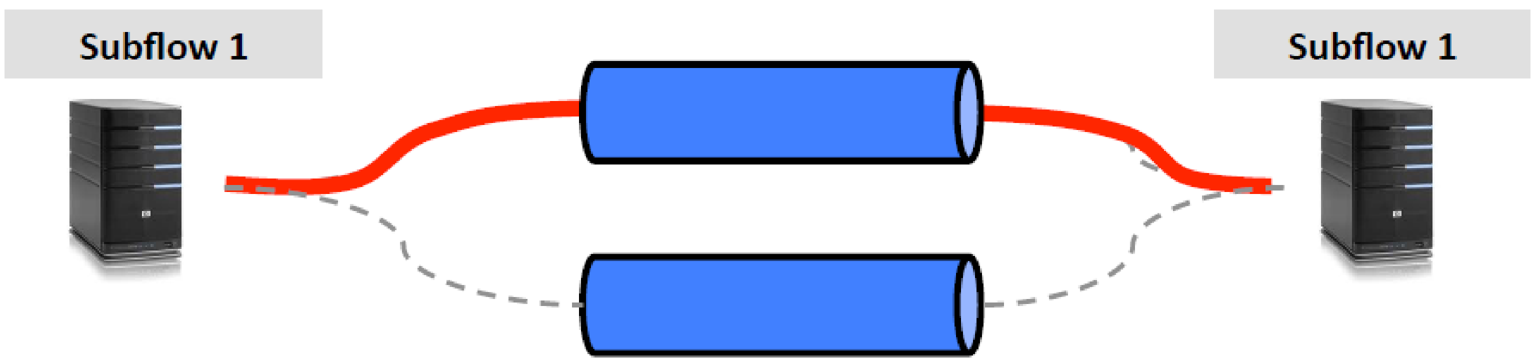
Enable MPTCP if
SYN/ACK has MP_CAPABLE

ENABLED

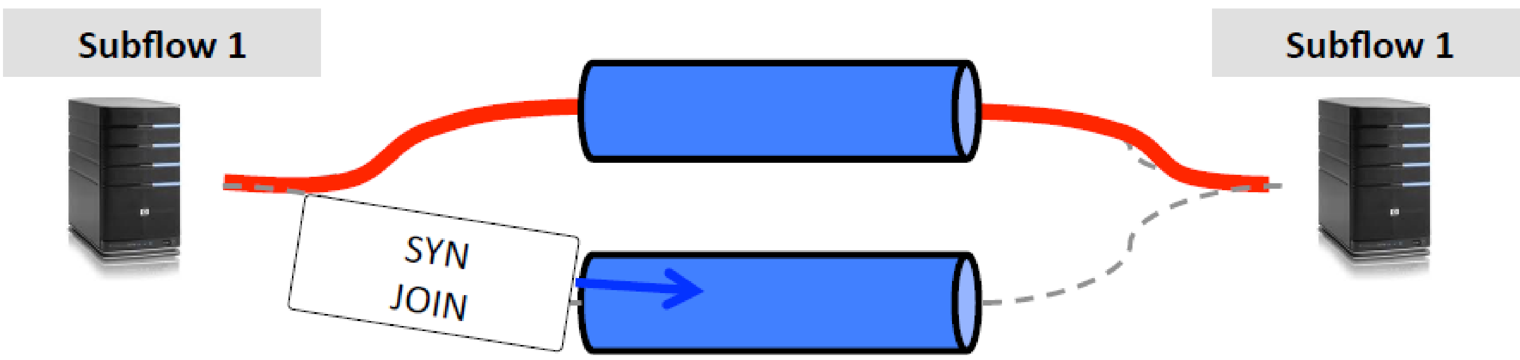
Enable MPTCP if
SYN has MP_CAPABLE

ENABLED

MTCP: Connection Management

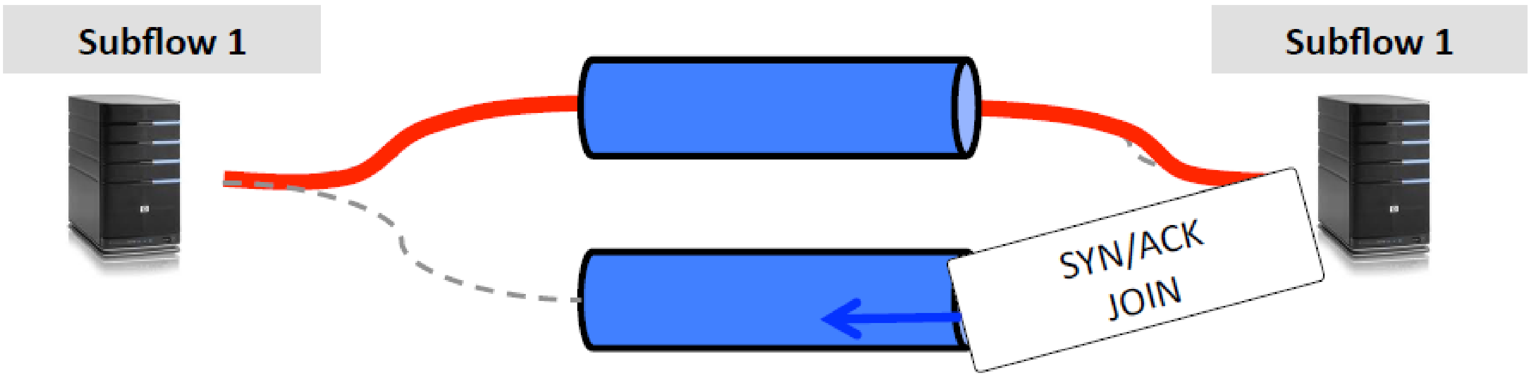


MTCP: Connection Management



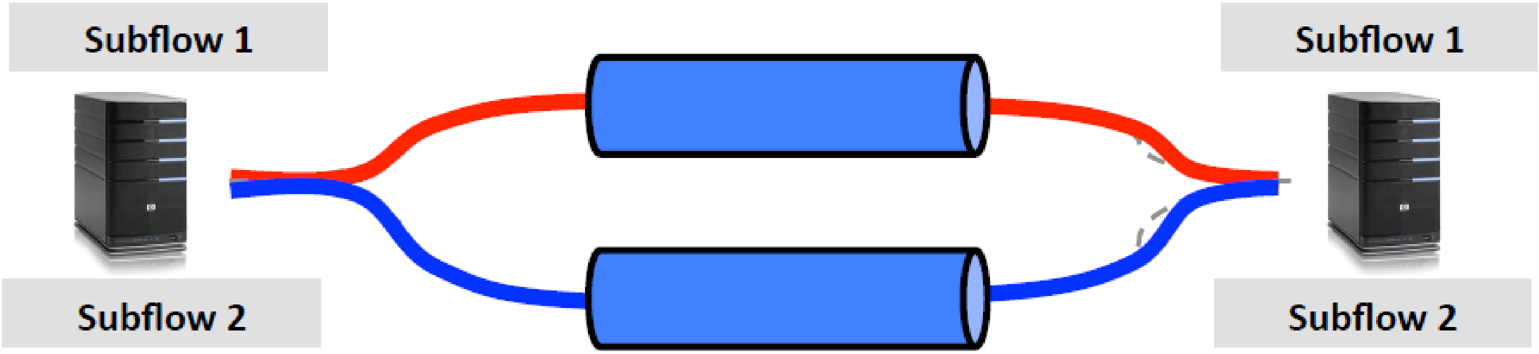
Subflows can be between different interfaces or between the same pair of IP addresses but different ports

MTCP: Connection Management



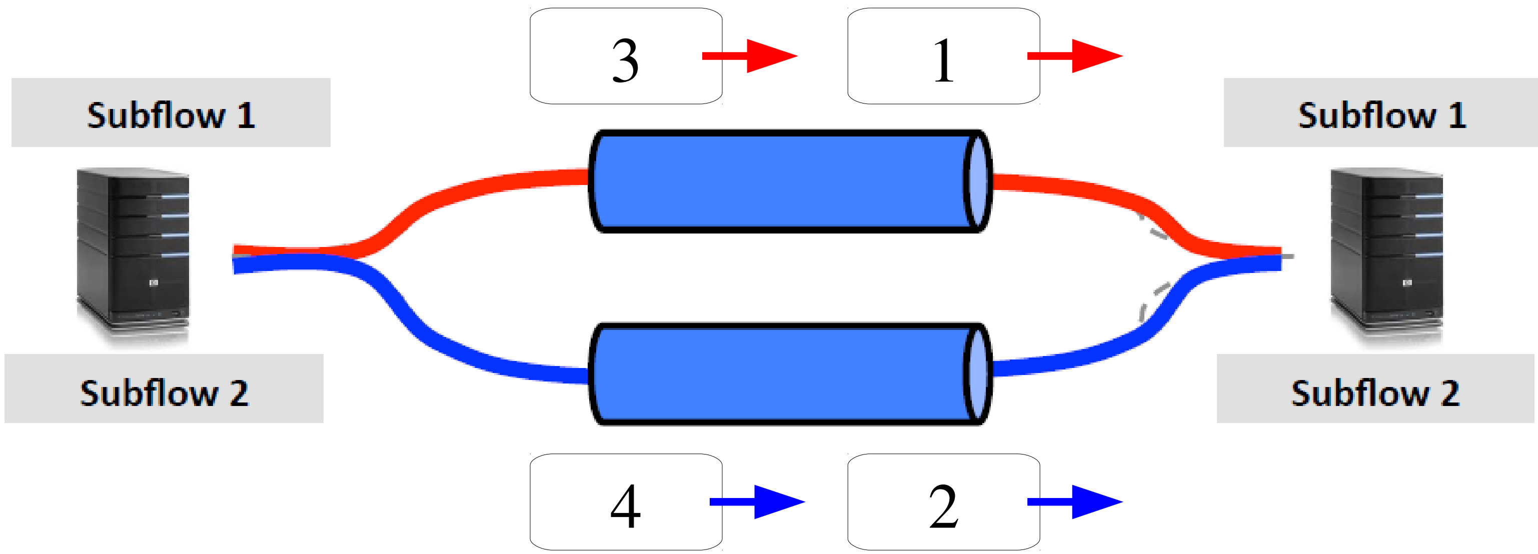
MTCP: Connection Management

MTCP: Connection Management



MPTCP relies on ECMP to hash different subflows to different paths

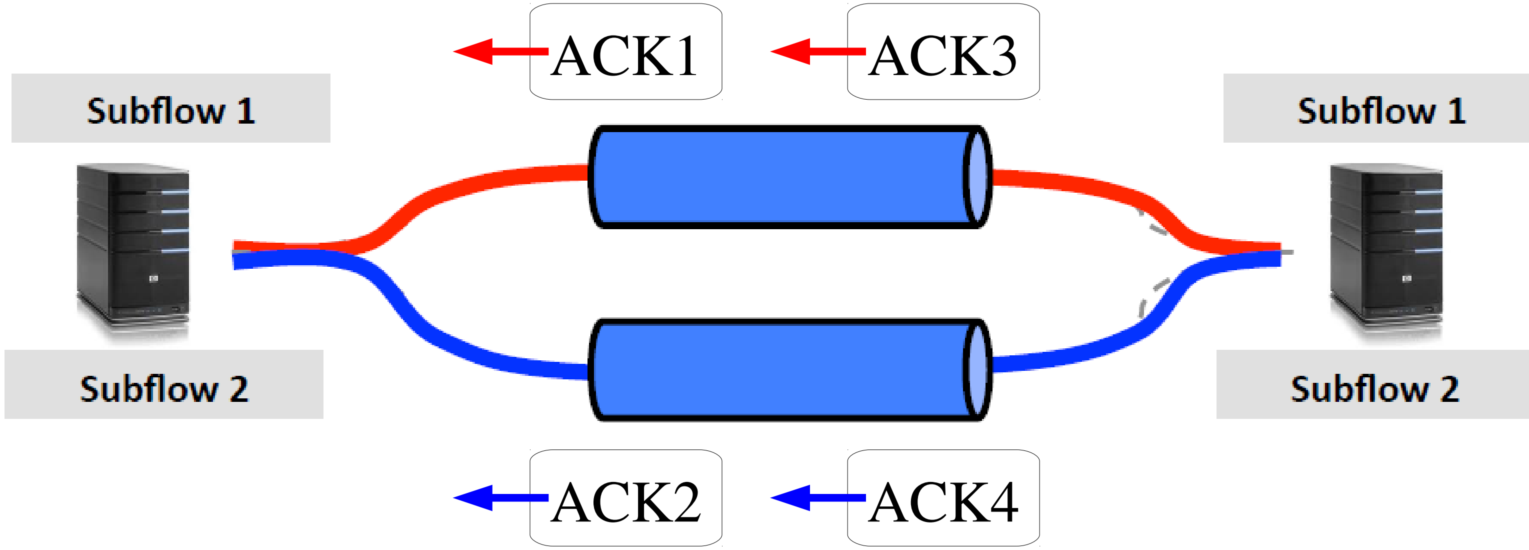
MTCP: Sending Data



MPTCP stripes TCP across the subflows

Additional TCP options allow the receiver to reconstruct the received data in the original order

MTCP: Sending Data



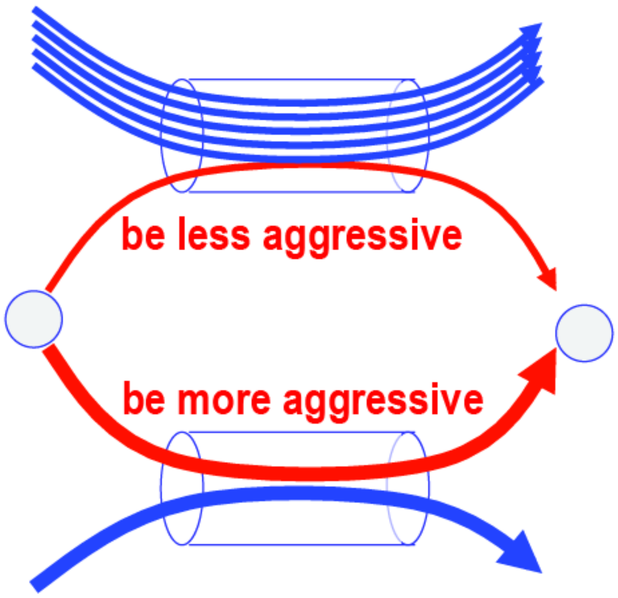
MPTCP stripes TCP across the subflows

When used over the Internet, middleboxes
may drop ACKs of unseen data packets

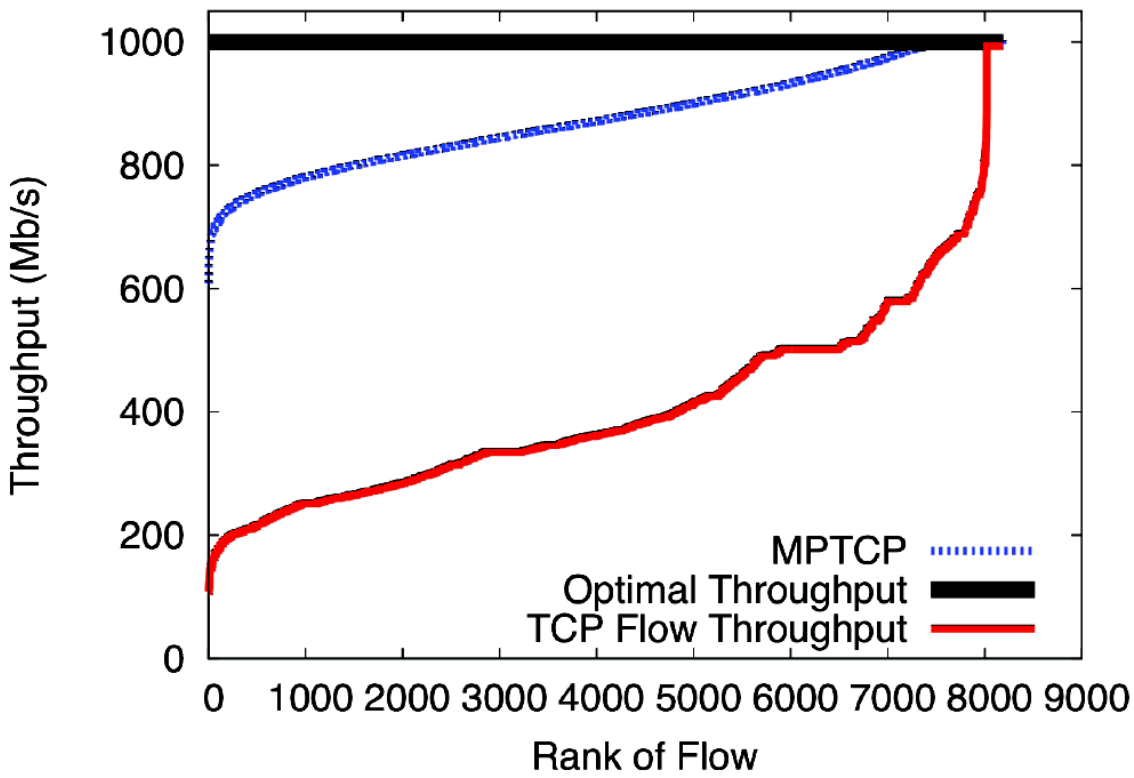
MPTCP Congestion Control

- Each path runs its own congestion control, to detect and respond to the congestion it sees

- But link congestion control parameters, so as to move traffic away from the congested paths

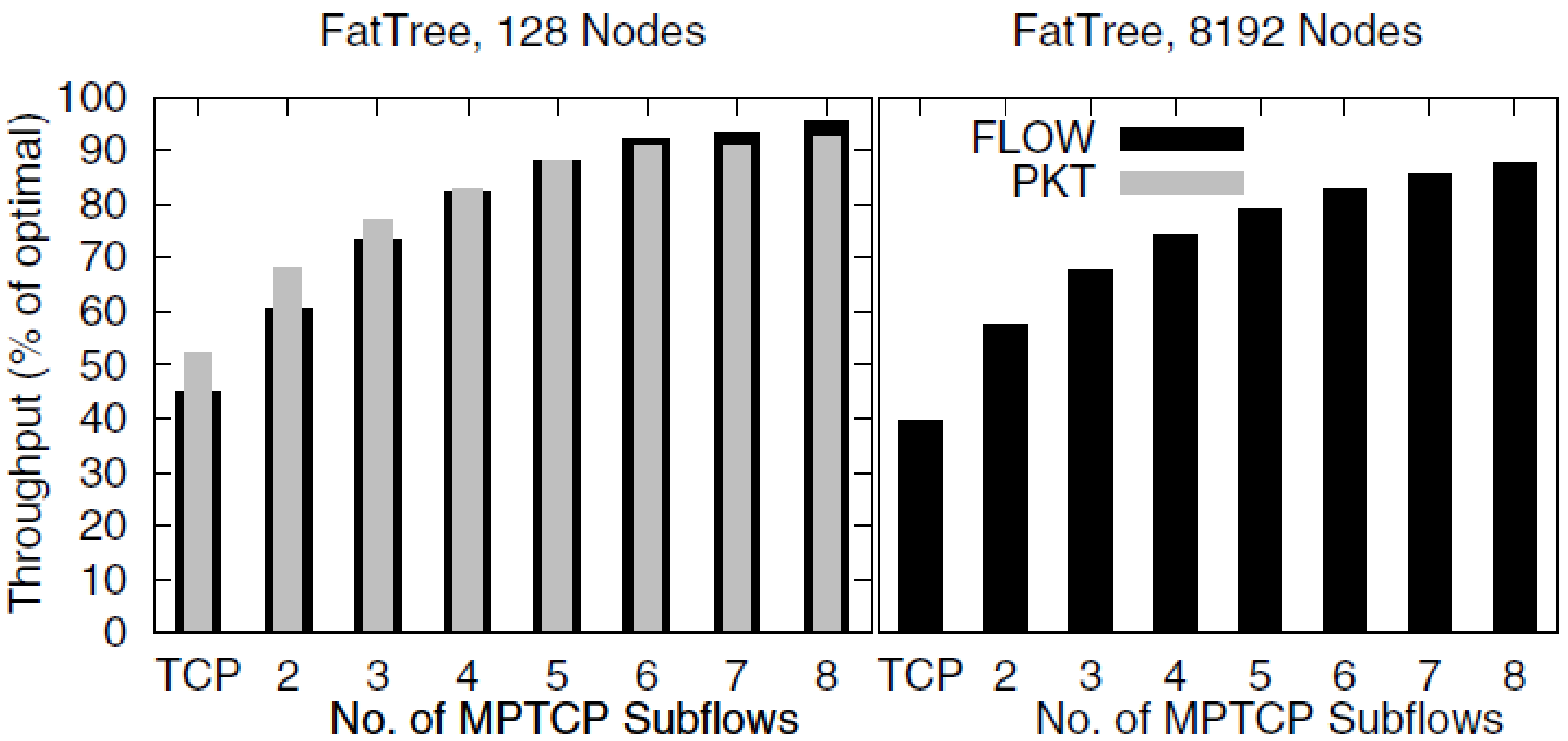


MPTCP in Fat Tree Network



Vast majority of flows using MPTCP achieve 80-100% throughput

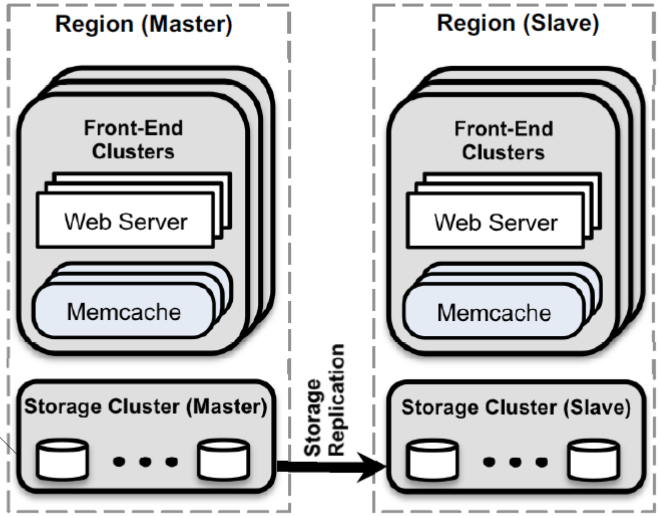
How many subflows are needed?

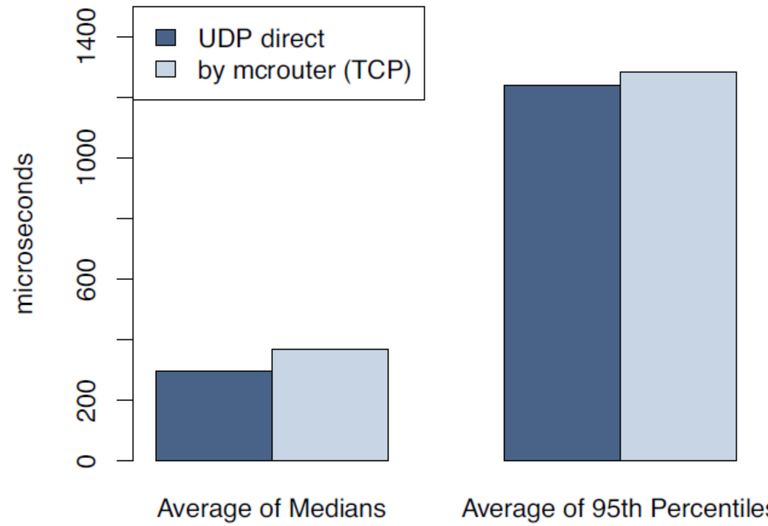


What about UDP?

Scaling Memcached at Facebook

HDFS,
MySQL,
etc.





- UPD has lower latency than TCP
- UDP creates less state (uses less memory) per client than TCP

Next week

- Flow control
- Infiniband
- Converged enhanced Ethernet

- Reading: