Advanced Computer Networks

CS740

# Transport in Data Center Networks (III)

https://pages.cs.wisc.edu/~mgliu/CS740/F25/index.html

**Ming Liu**
**mgliu@cs.wisc.edu**

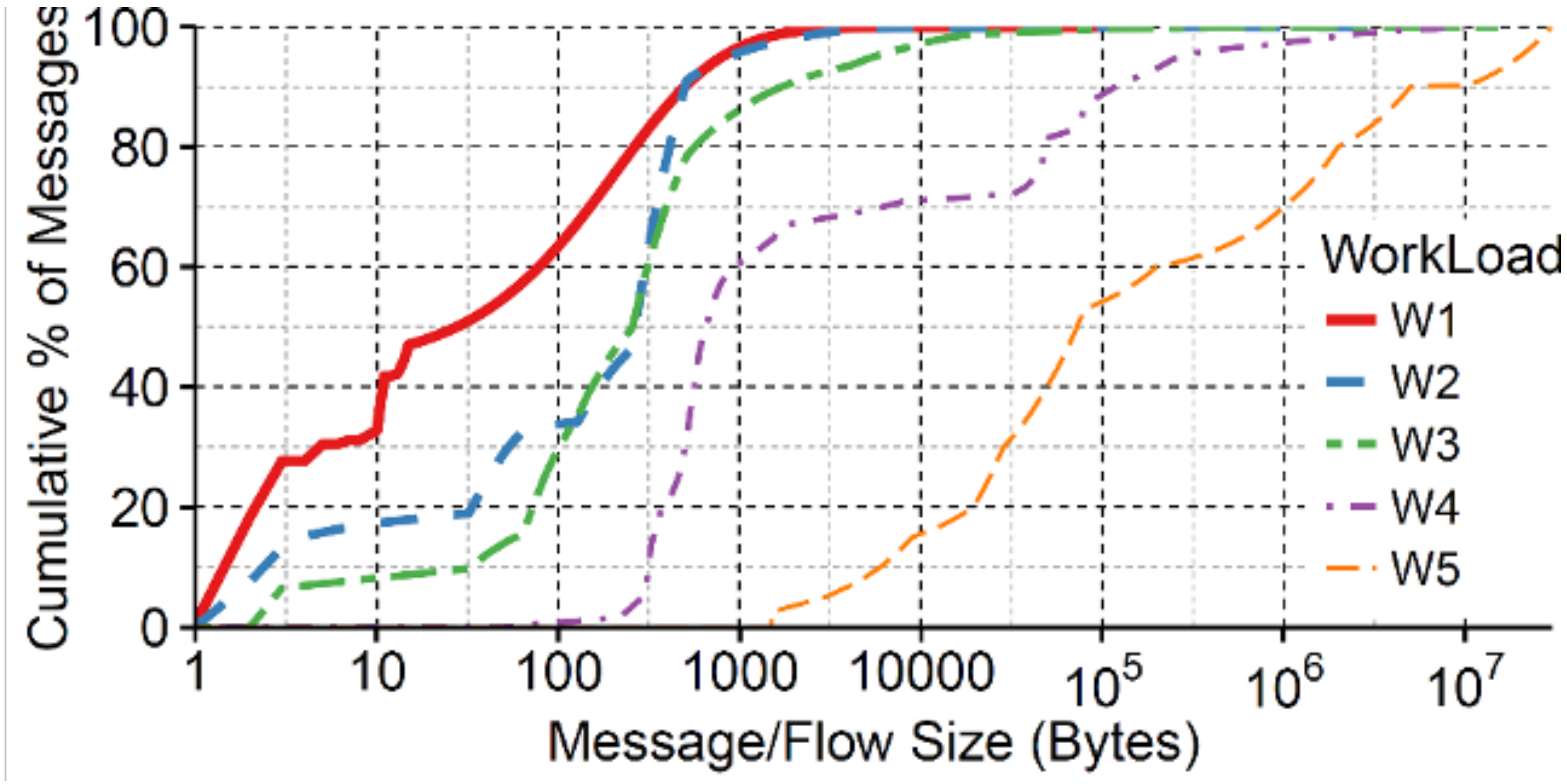
# Outline

- Last lecture
  - Transport in Data Center Networks (II)
- Today
  - Transport in Data Center Networks (III)
- Announcements
  - Midterm report due today 11:59 PM
  - Lab2 due 11/05/2025 11:59 PM

**Problem: How can we achieve low latency of tiny messages under high networking load?**

# Short Messages Dominate Workloads

- W1, W2: 95% of messages shorter than 1000 bytes



# Some Facts on (Tail) Latency

- One-way latency of a tiny 200 bytes message: **~5us**

# Some Facts on (Tail) Latency

- One-way latency of a tiny 200 bytes message: **~5us**

- 8 packets of queueing add **10us** of latency at 10Gbps

# Some Facts on (Tail) Latency

- One-way latency of a tiny 200 bytes message: **~5us**

- 8 packets of queueing add **10us** of latency at 10Gbps

3X!

# Some Facts on (Tail) Latency

- One-way latency of a tiny 200 bytes message: **~5us**

- 8 packets of queueing add **10us** of latency at 10Gbps

- A 100KB message takes **~80us** to transmit
  - Less impact

# Some Facts on (Tail) Latency

- One-way latency of a tiny 200 bytes message: **~5us**
- 8 packets of queueing add **10us** of latency at 10Gbps
- A 100KB message takes **~80us** to transmit
  - Less impact

**Near-hardware tail latency is hard!**

# Observation: get rid of any queueing effect

# Prior Solutions

# Prior Solutions

- DCTCP
  - Keep the queue short
  - Less than the threshold (K)

# Prior Solutions

- DCTCP
  - Keep the queue short
  - Less than the threshold (K)

- NDP
  - Cut off the payload when the queue is beyond a threshold
  - Rely on the receiver PULL packet for rate control

# Prior Solutions

- DCTCP
  - Keep the queue short
  - Less than the threshold (K)

- NDP
  - Cut off the payload when the queue is beyond a threshold
  - Rely on the receiver PULL packet for rate control

**Shallow queue != Zero queue**

# Can we remove the queue?

## ~~Can we remove the queue?~~

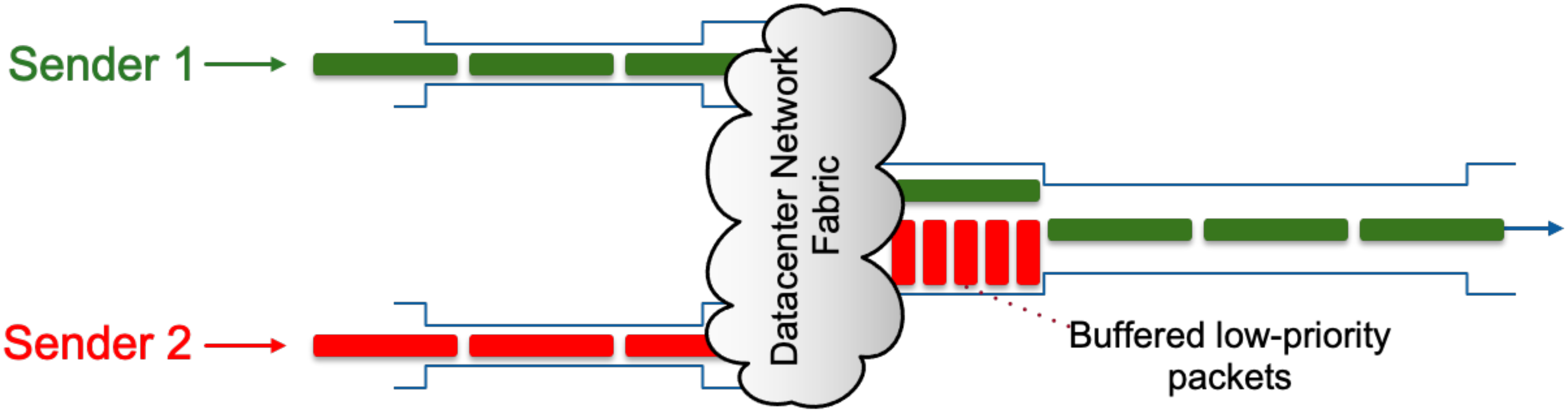
Transit Burst

High BW Utilization

# How about bypassing the queue?

# Networking Priorities

- Use hardware-provided priority queues to reduce queueing delay



**The key idea of Homa is using priorities effectively to bypass buffering.**

# How does Homa work?

# #1: Connection Setup @Sender Endhost

# #1: Connection Setup @Sender Endhost

- TCP: three-way handshake
  - Pessimistic and assume there is minimal spare network capacity

# #1: Connection Setup @Sender Endhost

- TCP: three-way handshake
  - Pessimistic and assume there is minimal spare network capacity

- NDP: no connection setup
  - Optimistic and assume there will be enough capacity

# #1: Connection Setup @Sender Endhost

- TCP: three-way handshake
  - Pessimistic and assume there is minimal spare network capacity

- NDP: no connection setup
  - Optimistic and assume there will be enough capacity

- **Homa: no connection setup**
  - **Similar assumptions as NDP**
  - **Connection-less message-based protocol**

# #2: Data Transmission in the 1st RTT @Sender Endhost

# #2: Data Transmission in the 1st RTT @Sender Endhost

- TCP: slow start
  - Initially with a small congestion window

# #2: Data Transmission in the 1st RTT @Sender Endhost

- TCP: slow start
  - Initially with a small congestion window

- NDP: full bandwidth-delay product (BDP)
  - Network utilization is relatively low

# #2: Data Transmission in the 1st RTT @Sender Endhost

- TCP: slow start
  - Initially with a small congestion window

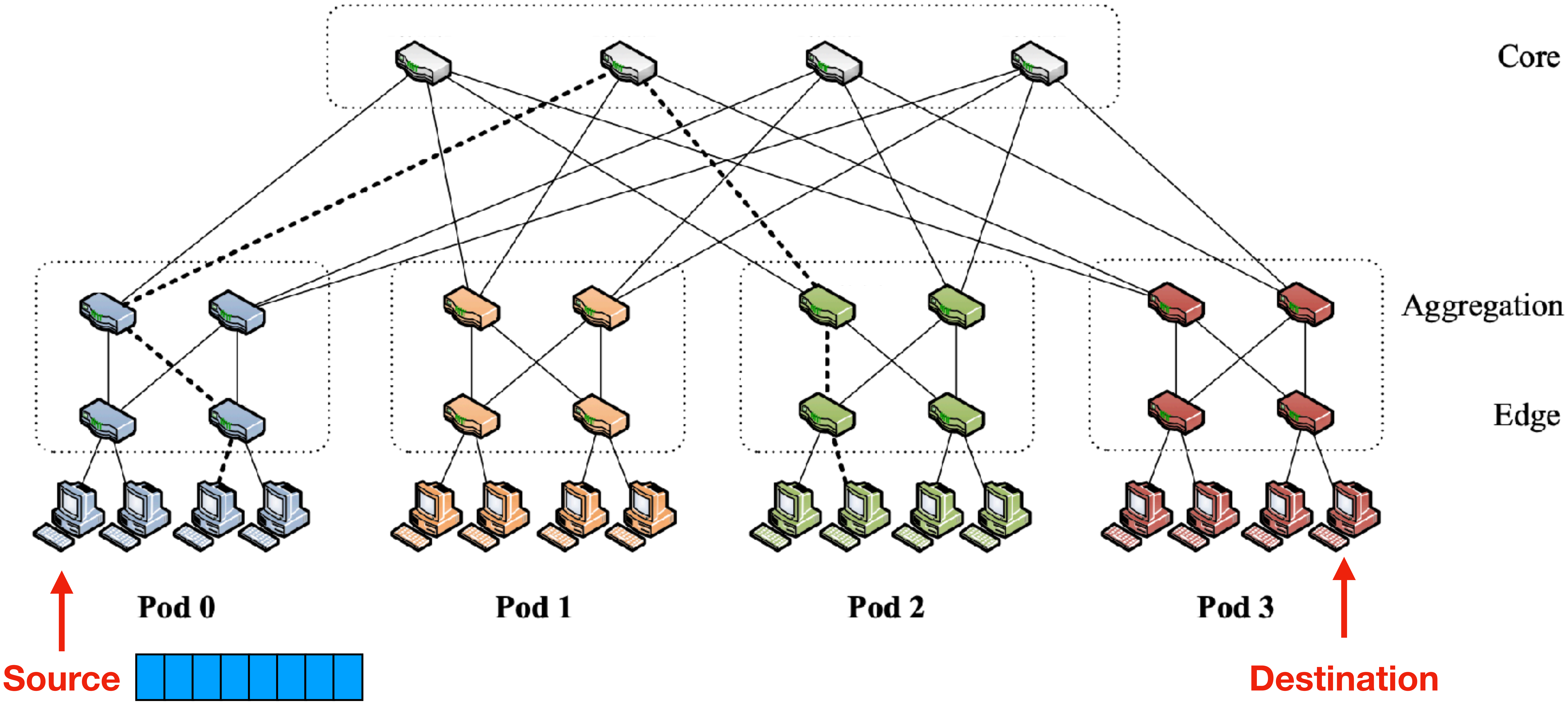
- NDP: full bandwidth-delay product (BDP)
  - Network utilization is relatively low

- **Homa: full bandwidth-delay product (BDP)**
  - **Unscheduled packets**

# #3: Routing @Switch

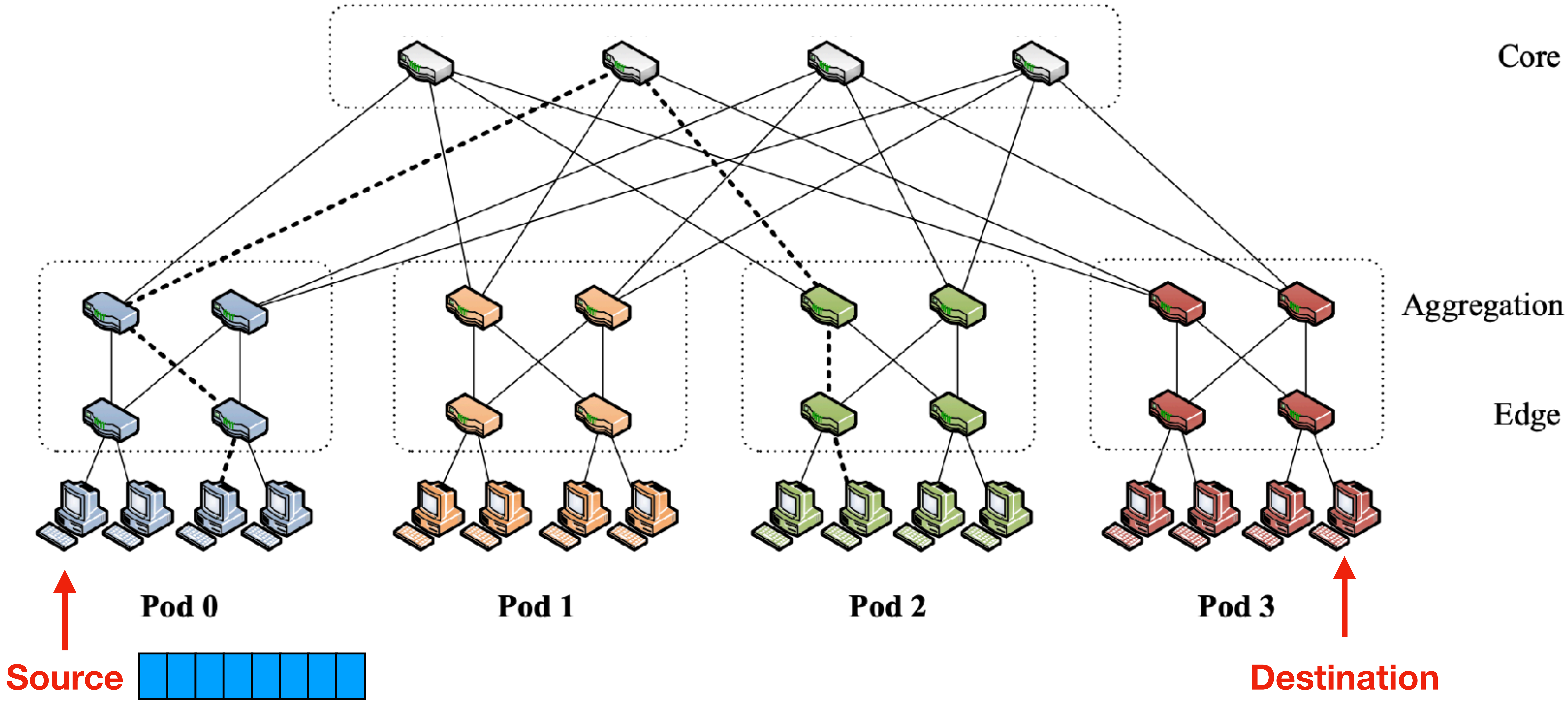
# #3: Routing @Switch

- TCP: per-flow routing



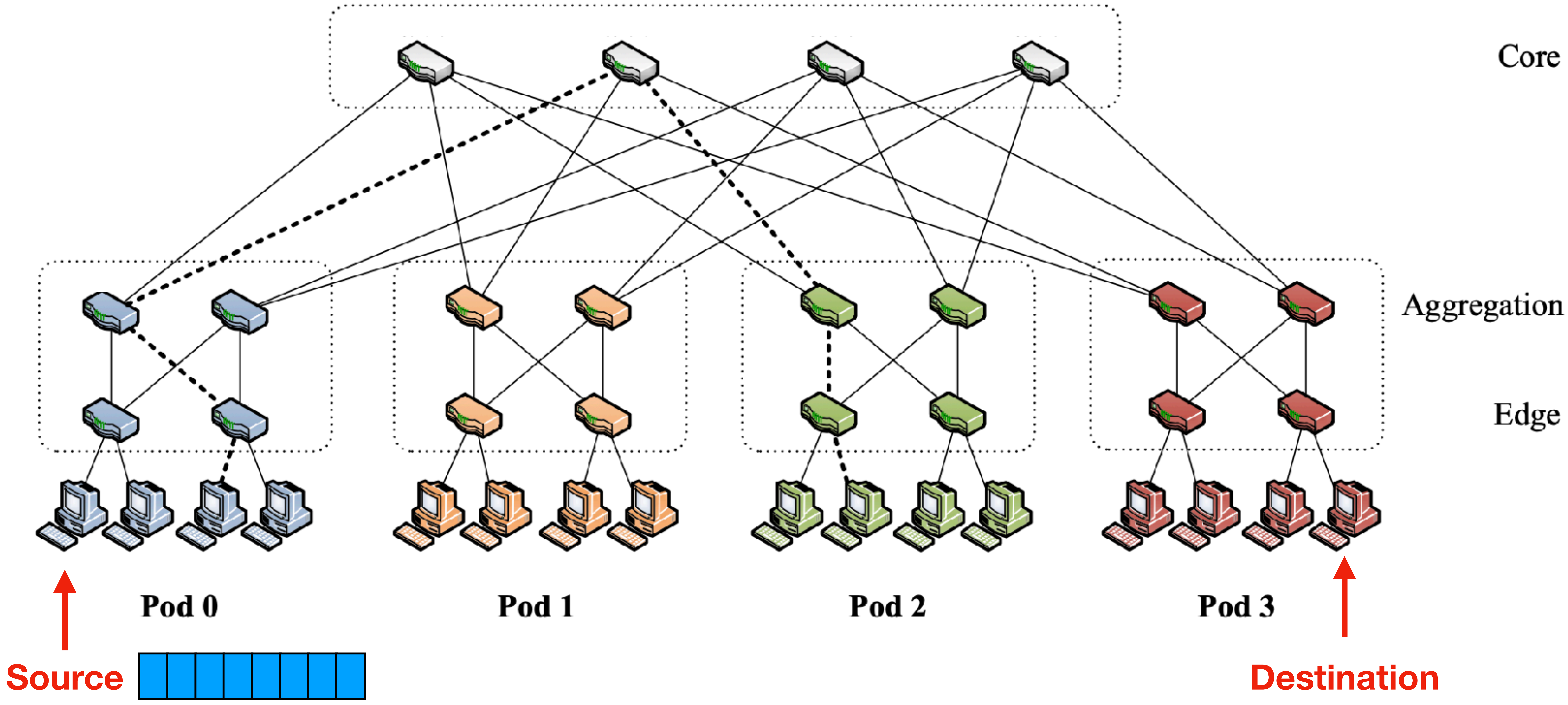
# #3: Routing @Switch

- NDP: per-packet routing



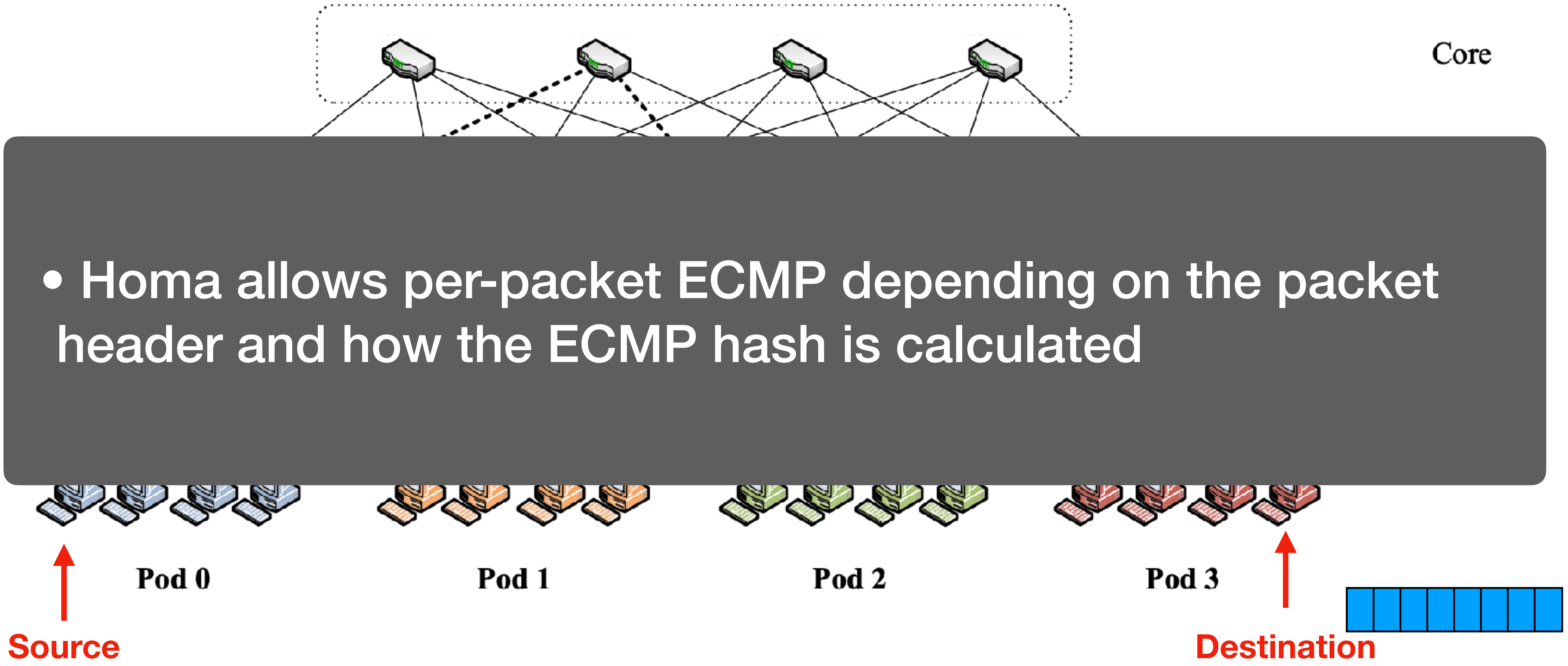
# #3: Routing @Switch

- Homa: per-flow routing



# #3: Routing @Switch

- Homa: per-flow routing




- Homa allows per-packet ECMP depending on the packet header and how the ECMP hash is calculated

# #4: Transport @Receiver Endhost

# #4: Transport @Receiver Endhost

- TCP: ACK+flow control
  - Update the sender the availability of the receiving buffer

# #4: Transport @Receiver Endhost

- TCP: ACK+flow control
  - Update the sender the availability of the receiving buffer

- NDP: Receiver-driven
  - NACK for reliability
  - PULL packets for packing

# #4: Transport @Receiver Endhost

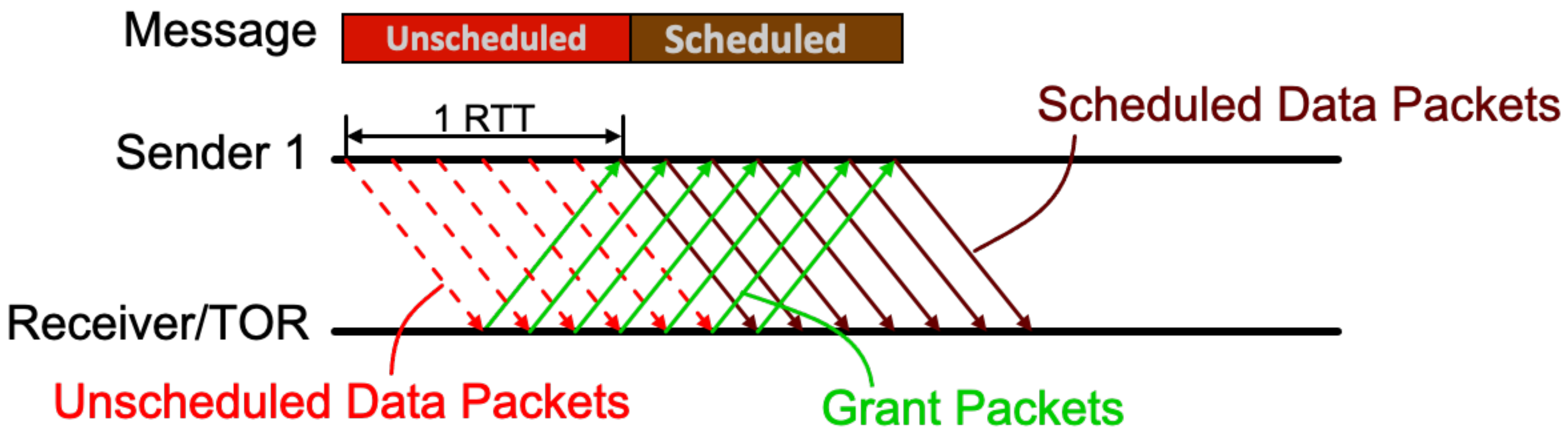
- TCP: ACK+flow control
  - Update the sender the availability of the receiving buffer

- NDP: Receiver-driven
  - NACK for reliability
  - PULL packets for packing

- **Homa: Receiver-driven**
  - **Network priority kicks in**

# Homa Receiver-driven Transport

- The receiver determines how to schedule the remaining flow
  - Unscheduled -> Scheduled
  - Priority: label each packet based on the urgency
  - Grant: provide transmission permission to the sender, one per packet



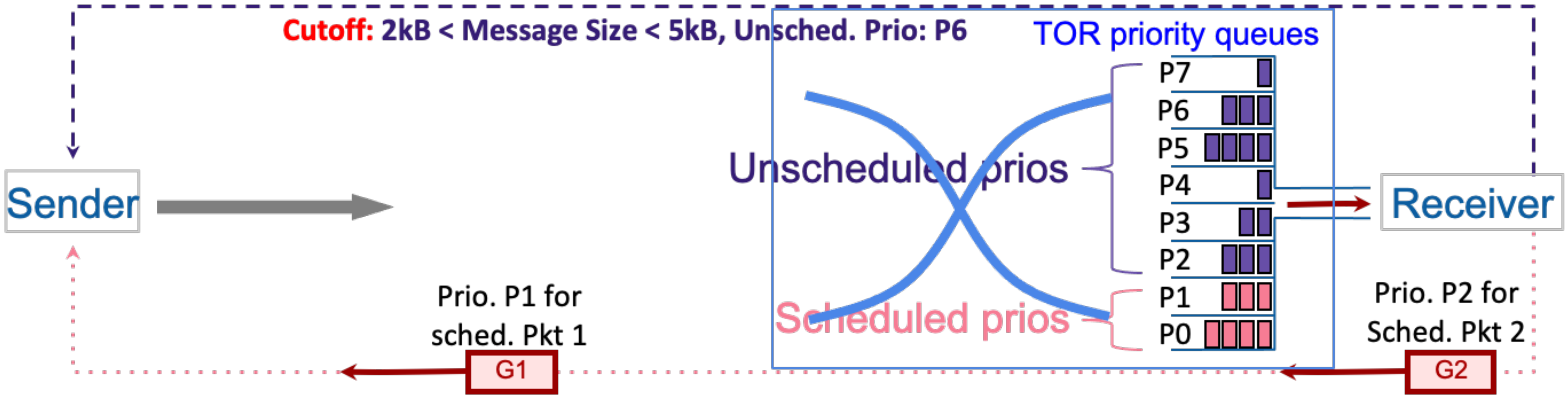
# Homa Receiver-driven Transport

- The receiver determines how to schedule the remaining flow
  - Unscheduled -> Scheduled
  - Priority: label each packet based on the urgency
  - Grant: provide transmission permission to the sender, one per packet

- **How does Homa assign priorities?**
- **How many grants does a receiver allocate?**

# Dynamic Priority Assignment

- Principles
  - Unscheduled packets: pre-assigned with higher priorities
  - Scheduled packets: adaptive priority specified in each grant



# Dynamic Priority Assignment

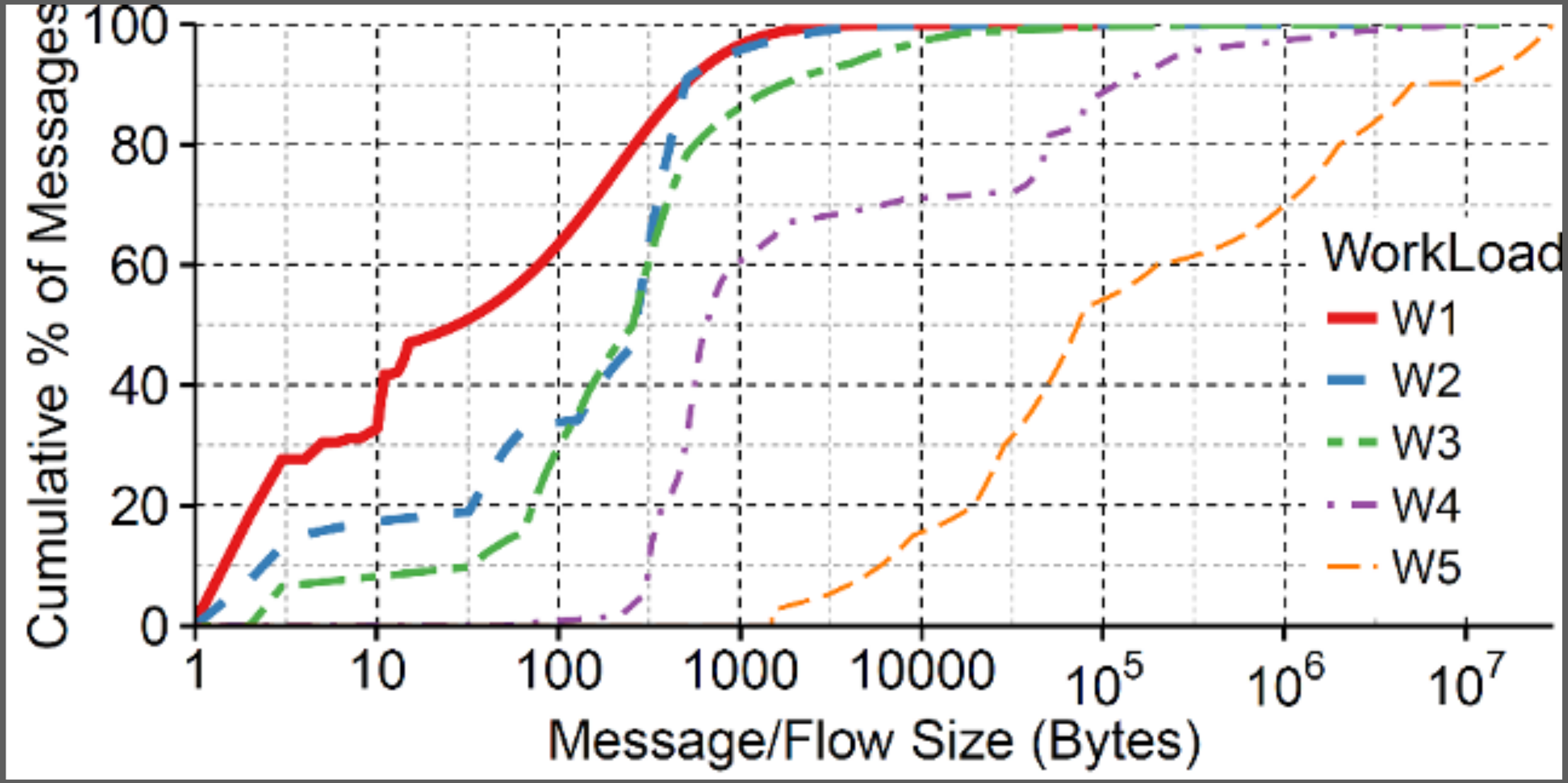
- Principles

Why unscheduled packets receive higher priority?

# Dynamic Priority Assignment

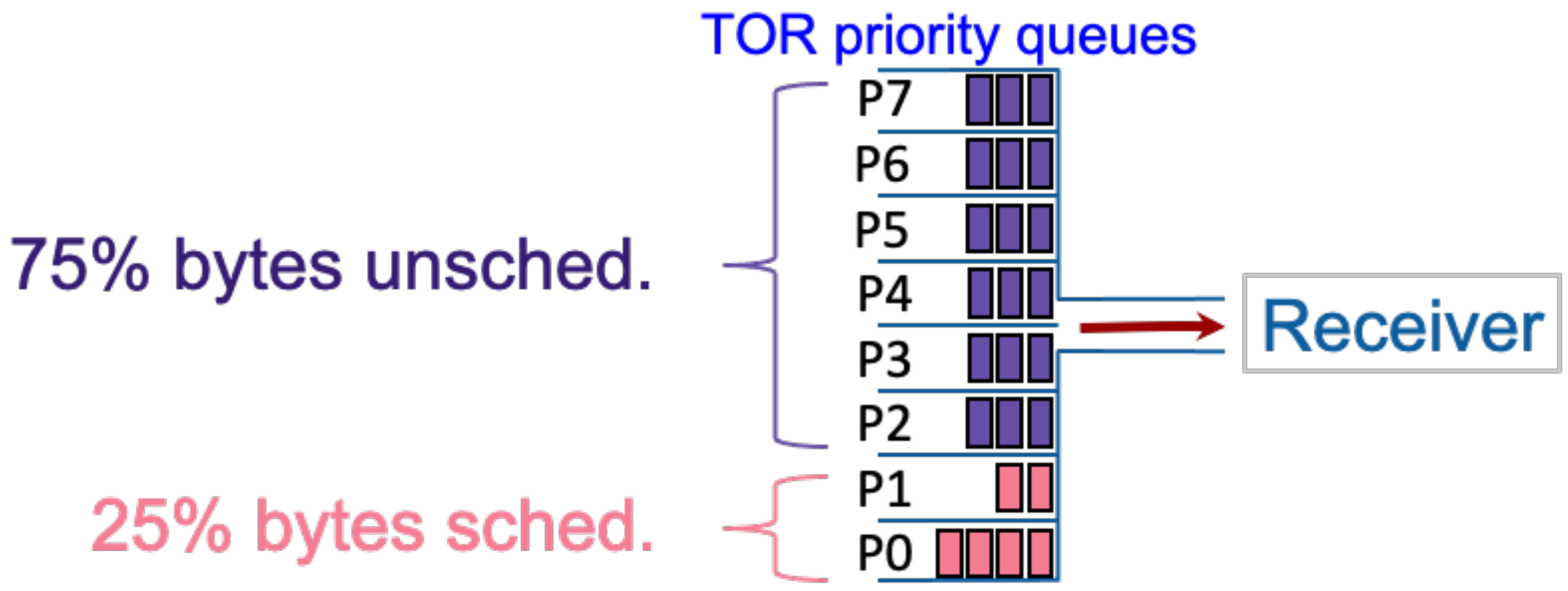
- Principles

**Why unscheduled packets receive higher priority?**



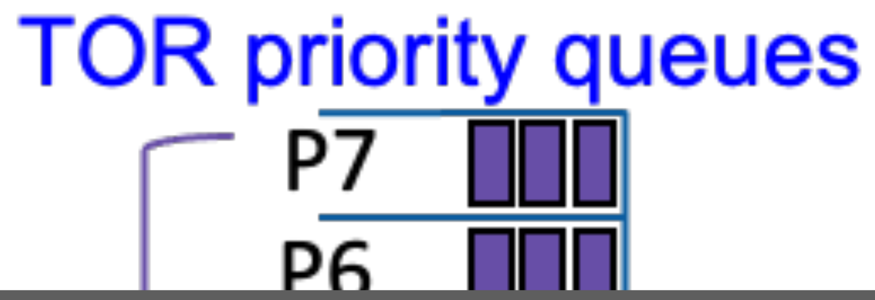
# Priority Assignment for **Unscheduled Packets**

- Pick cut-offs based on CDF of message sizes
  - Short messages, higher unscheduled priority
  - Equal bytes per priority level



# Priority Assignment for **Unscheduled Packets**

- Pick cut-offs based on CDF of message sizes
  - Short messages, higher unscheduled priority
  - Equal bytes per priority level



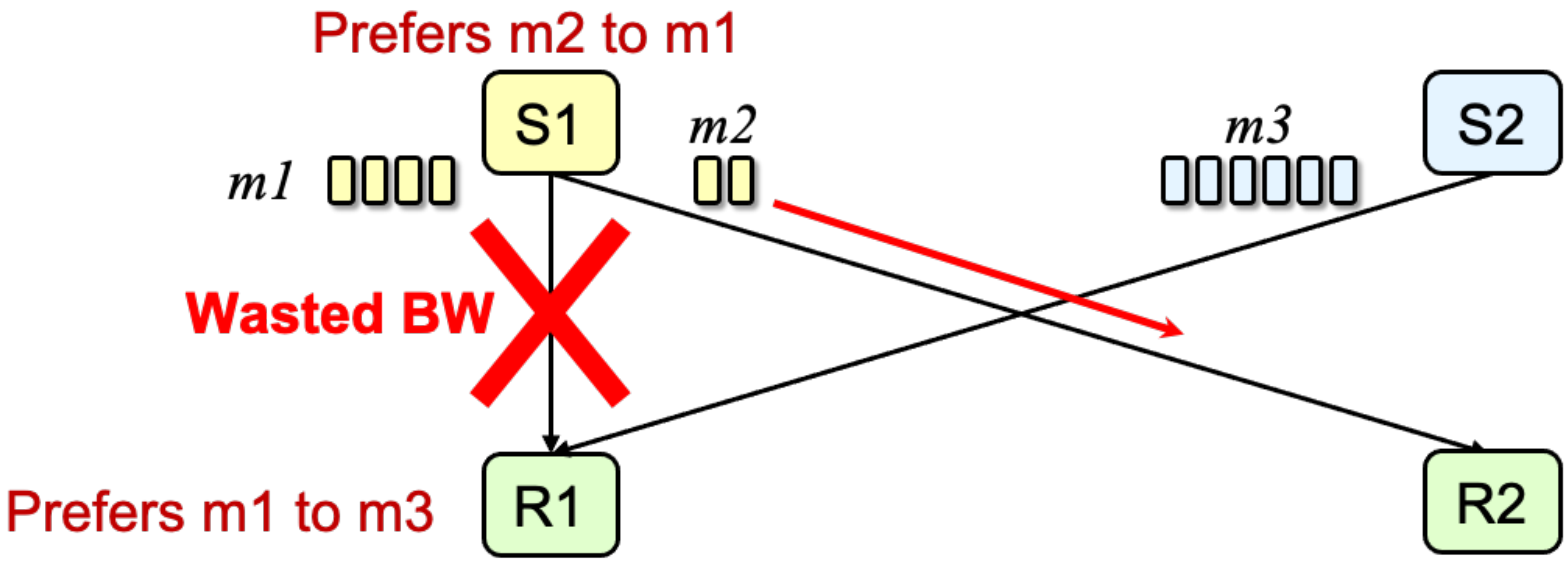

Fraction of priorities for unscheduled packets =
Fraction of incoming bytes unscheduled

# Priority Assignment for **Scheduled Packets**

- Allocate adaptively based on incoming messages
  - Start with the lowest priority
  - Use higher priority for pad preemption

# An issue: Priority Mechanisms Hurt Utilization

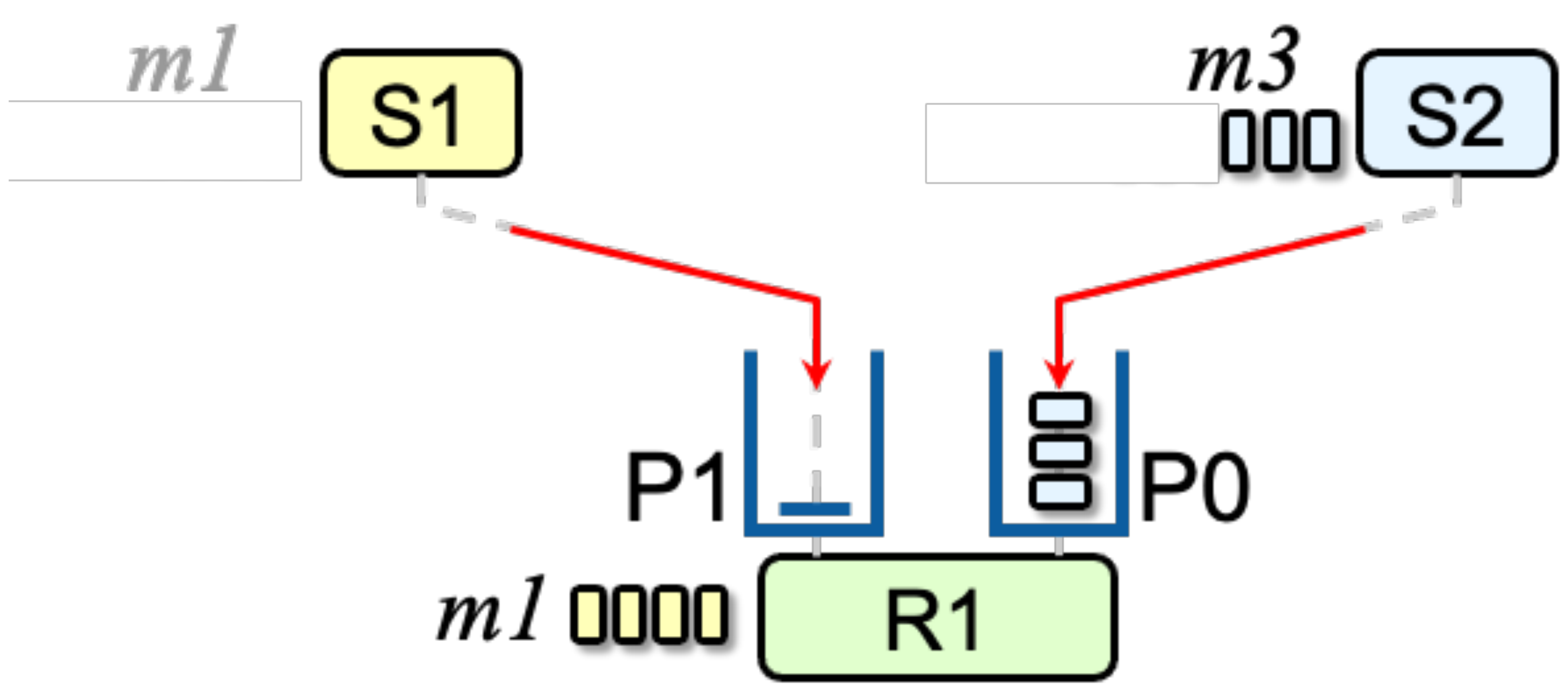
- Senders may not respond promptly to grant



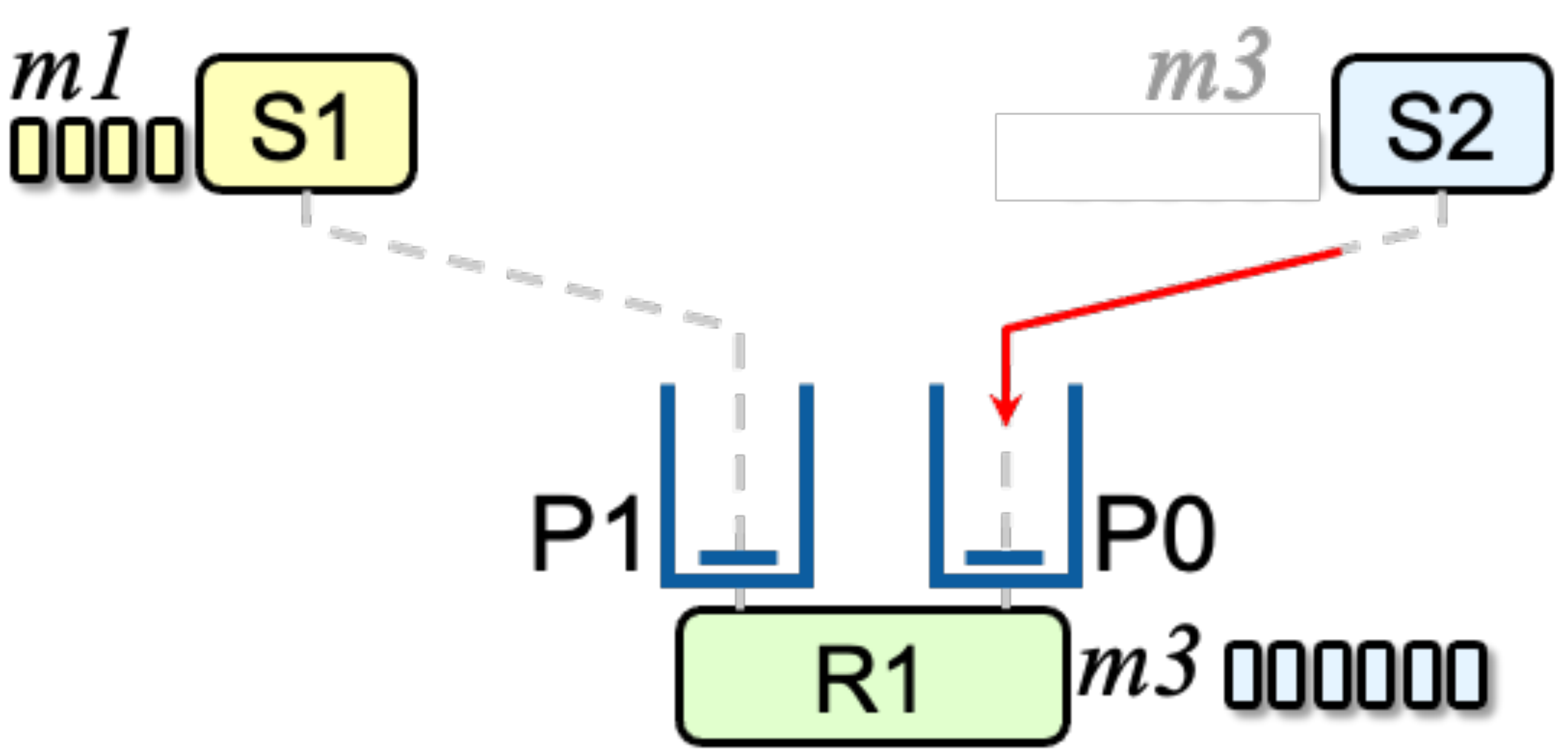
# Controlled Overcommitment

- Use priorities to favor short message
- Use buffering to achieve high bandwidth usage

Both senders respond




Sender 1 doesn't respond



# How many grants are issued?

# Use Grants for Rate Control

- Tell the sender to send N remaining bytes of data

- N = Min (BDP, Remaining bytes #, Offset)
    - Offset, determined based on receiving buffer availability and fairness

# #5: Data Transmission in the 2nd+ RTT @Sender Endhost

# #5: Data Transmission in the 2nd+ RTT @Sender Endhost

- TCP: congestion control kicks in
  - Adjust the congestion window based on the signal

# #5: Data Transmission in the 2nd+ RTT @Sender Endhost

- TCP: congestion control kicks in
  - Adjust the congestion window based on the signal

- NDP: receiver-driven
  - PULL packets are used for pacing
  - Trimmed packets are used for handling congestion collapse

# #5: Data Transmission in the 2nd+ RTT @Sender Endhost

- TCP: congestion control kicks in
  - Adjust the congestion window based on the signal

- NDP: receiver-driven
  - PULL packets are used for pacing
  - Trimmed packets are used for handling congestion collapse

- **Homa: receiver-driven**
  - **Grant packets are used for pacing**
  - **Priority is used for minimizing the impact of large flows**

# #6: Connection (Flow) Teardown @Sender

# #6: Connection (Flow) Teardown @Sender

- TCP: connection state machine
  - Different teardown modes depending on the sender/receiver status

# #6: Connection (Flow) Teardown @Sender

- TCP: connection state machine
  - Different teardown modes depending on the sender/receiver status

- NDP: N/A
  - The sender marks the last packet
  - PULL —> Push

# #6: Connection (Flow) Teardown @Sender

- TCP: connection state machine
  - Different teardown modes depending on the sender/receiver status

- NDP: N/A
  - The sender marks the last packet
  - PULL —> Push

- **Homa: N/A**
  - **The sender marks the last packet**
  - **Grants can still be received**

# What happens if the switch queue is full?

# #7: Queueing Handling @Switch

# #7: Queueing Handling @Switch

- TCP: drop => congestion signal
  - RED (random early drop)
  - DCTCP employs ECN

# #7: Queueing Handling @Switch

- TCP: drop => congestion signal
  - RED (random early drop)
  - DCTCP employs ECN
- NDP: packet trimming
  - Headers of trimmed packets are used for traffic control
  - Co-design the switch behavior with the transport protocol

# #7: Queueing Handling @Switch

- TCP: drop => congestion signal
  - RED (random early drop)
  - DCTCP employs ECN
- NDP: packet trimming
  - Headers of trimmed packets are used for traffic control
  - Co-design the switch behavior with the transport protocol
- **Homa: drop**
  - **The switch performs priority packet scheduling**

# What happens if packets are delivered out-of-order?

# #8: Reorder Handling @Sender/Reciver Endhost

# #8: Reorder Handling @Sender/Reciver Endhost

- TCP: a potential indicator of network congestion
  - Three duplicated ACKs
  - Selective retransmission

# #8: Reorder Handling @Sender/Reciver Endhost

- TCP: a potential indicator of network congestion
  - Three duplicated ACKs
  - Selective retransmission
- NDP: common behavior due to per-packet routing
  - Rely on PULL packets
  - Maintain a separate pull sequence space for each connection

# #8: Reorder Handling @Sender/Reciver Endhost

- TCP: a potential indicator of network congestion
  - Three duplicated ACKs
  - Selective retransmission

- NDP: common behavior due to per-packet routing
  - Rely on PULL packets
  - Maintain a separate pull sequence space for each connection

- **Homa: common behavior**
  - **Use priority to escalate the scheduled packet (re)transmission**

# What happens if links or switches fail?

# #9: Failure Handling @Sender/Receiver Endhost

# #9: Failure Handling @Sender/Receiver Endhost

- TCP: timeout engineering
  - Senders reduce the congestion window and retransmit unacked packets

# #9: Failure Handling @Sender/Receiver Endhost

- TCP: timeout engineering
  - Senders reduce the congestion window and retransmit unacked packets
- NDP: timeout + proactive retransmission
  - Senders keep a scoreboard for all paths
  - A small timeout based on the assumption that the network is regular

# #9: Failure Handling @Sender/Receiver Endhost

- TCP: timeout engineering
  - Senders reduce the congestion window and retransmit unacked packets

- NDP: timeout + proactive retransmission
  - Senders keep a scoreboard for all paths
  - A small timeout based on the assumption that the network is regular

- **Homa: timeout engineering**
  - **Little discussion**

# Is Homa fair?

# #10: Fairness Guarantee @Sender/Reciver Endhost

# #10: Fairness Guarantee @Sender/Reciver Endhost

- TCP: congestion control
  - Collaborate with the active queue management (AQM)

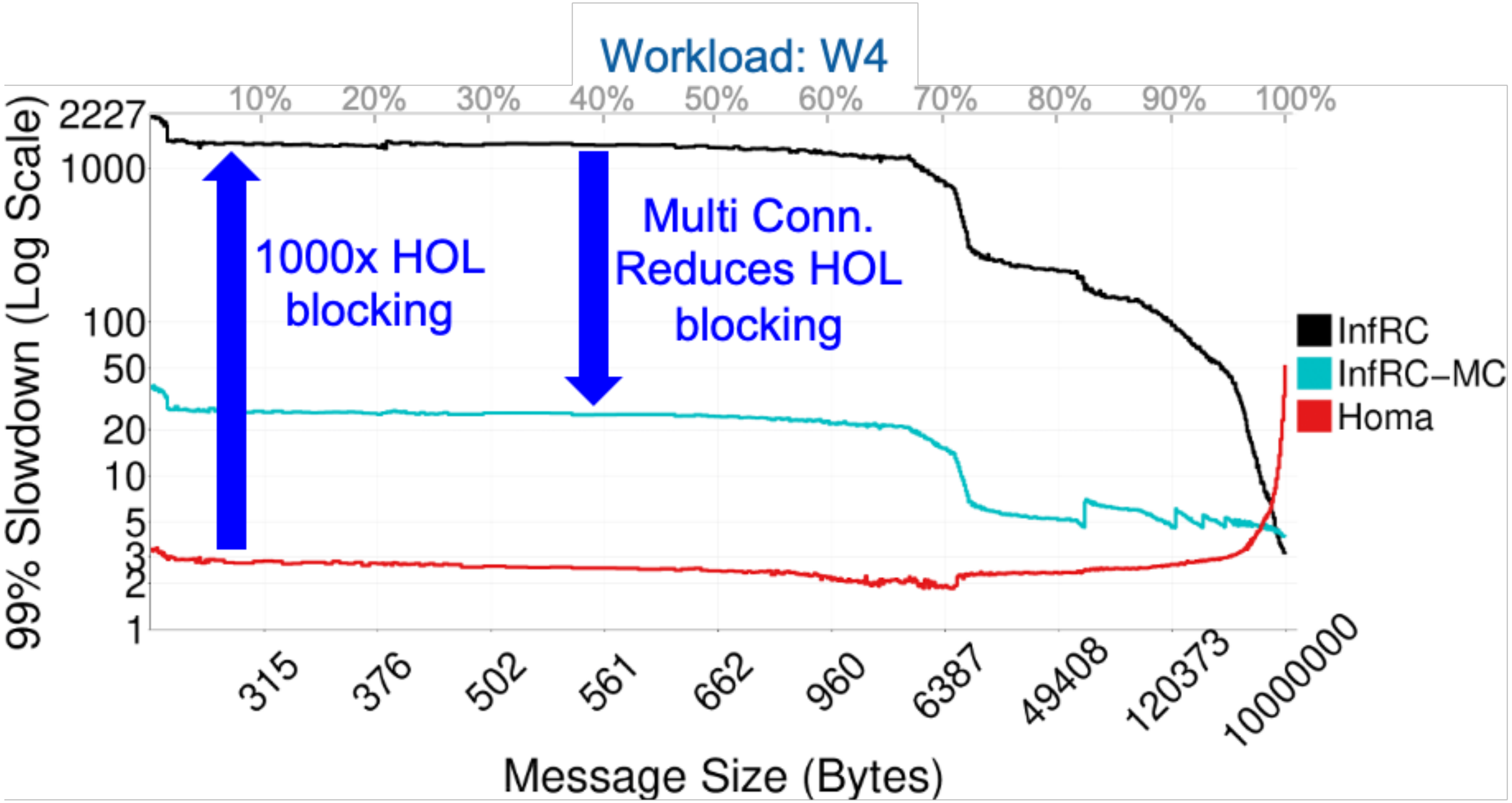
# #10: Fairness Guarantee @Sender/Reciver Endhost

- TCP: congestion control
  - Collaborate with the active queue management (AQM)
- NDP: fairness is inherently supported
  - The initial window is the same
  - The follow-up windows are controlled by the receiver, equally partitioned

# #10: Fairness Guarantee @Sender/Reciver Endhost

- TCP: congestion control
  - Collaborate with the active queue management (AQM)
- NDP: fairness is inherently supported
  - The initial window is the same
  - The follow-up windows are controlled by the receiver, equally partitioned
- **Homa**: fairness depends on priority
  - Small flows send unscheduled packets based on the message CDF
  - Large flows send scheduled packets based on priority adaptivity

# Homa Evaluation

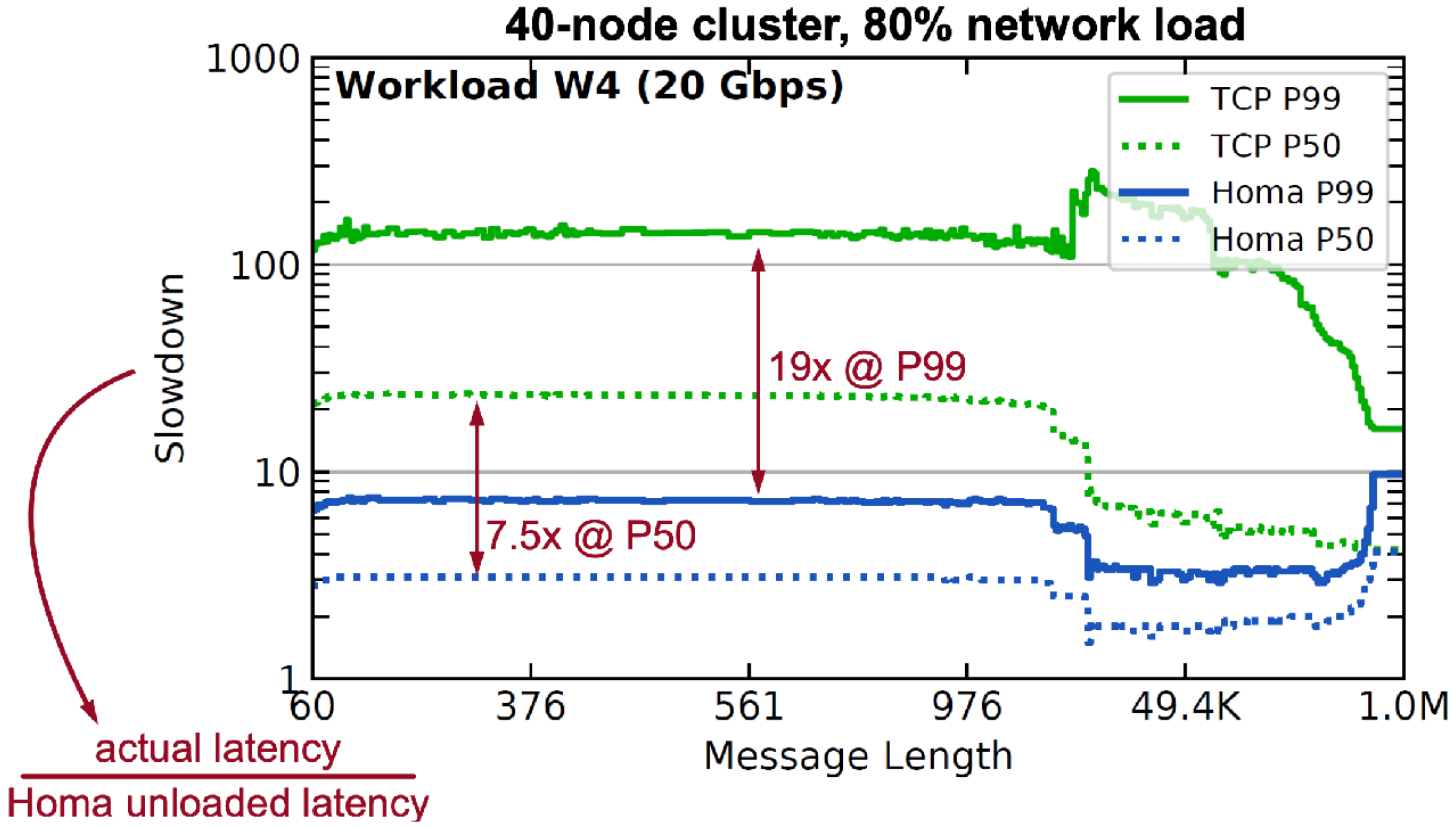


# Homa/Linux (ATC'21)

- Target: RPC-based applications in data centers
  - Message-oriented
  - Connection-less

```
int homa_send(int sockfd, const void *request, size_t reqlen,
        const struct sockaddr *dest_addr, socklen_t addrlen,
        uint64_t *id);
int homa_reply(int sockfd, const void *response, size_t resplen,
        const struct sockaddr *dest_addr, socklen_t addrlen,
        uint64_t id);
int homa_recv(int sockfd, void *buf, size_t len, int flags,
        struct sockaddr *src_addr, socklen_t addrlen,
        uint64_t *id);
```

# Homa/Linux >> TCP/Linux



# Still Have Some Overheads

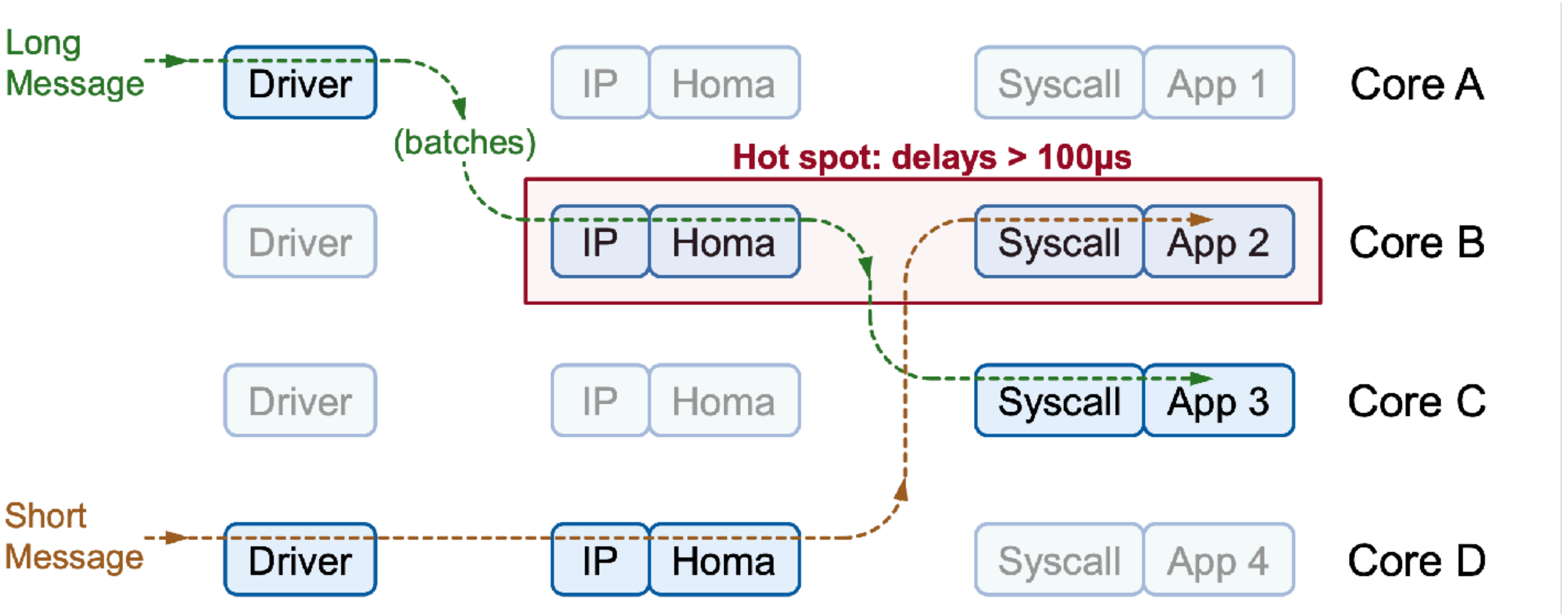
- Homa/Linux is 5—10x worse than the hardware raw capability
  - Homa/Linux: 100us
  - Homa/RamCloud: 14us

# Still Have Some Overheads

- Homa/Linux is 5—10x worse than the hardware raw capability
  - Homa/Linux: 100us
  - Homa/RamCloud: 14us

**Load balancing is hard! — the path between NIC to cores**

# Load Balancing Causes Hot Spots



# Homa/Linux v.s. Snap (SOSP'19)

- Google's user-space nstack implementation

| | Homa | Snap |
|---|---|---|
| Base latency (polling) | 15.1 µs | 9 µs |
| Cores to drive 80 Gbps bidirectional | 17 | 7–14 |

# Homa/Linux v.s. Snap (SOSP'19)

- Google's user-space nstack implementation

| | Homa | Snap |
|---|---|---|
| Base latency (polling) | 15.1 µs | 9 µs |
| Cores to drive 80 Gbps bidirectional | 17 | 7–14 |

**But still stuffers from load-balancing issues. For example, throughput per core drops by 3.5x - 7x.**

# Homa v.s. TCP

- **Connection oriented**
  - High time/space overheads (datacenter apps have 1000's of connections)
- **Stream oriented**
  - Awkward for RPCs (transport doesn't know message boundaries)
  - Head-of-line blocking
- **Fair sharing of bandwidth**
  - Increases latency, especially for short messages
- **Sender-driven congestion control**
  - Requires buffer occupancy to detect congestion
  - Buffer occupancy → high latency
- **Requires in-order packet delivery**
  - Cripples load balancing

https://arxiv.org/abs/2210.00714

# Summary

- Today
  - Homa

- Next topic: Endhost Networking Stack
  - Linux NStack (Sigcomm'21)
  - SNAP (SOSP'19)