Advanced Computer Networks

Transport in Data Center Networks (II)

CS740

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

Ming Liu
mgliu@cs.wisc.edu

Outline

- Last lecture
  - Transport in Data Center Networks (I)

- Today
  - Transport in Data Center Networks (II)

- Announcements
  - Lab2 due 11/05/2025 11:59 PM
  - Midterm report due 11/04/2025 11:59 PM

The Dilemma: Data center networking bandwidths keep increasing, but applications can hardly benefit from them!

Large-Scale RDMA Deployments

RDMA over Commodity Ethernet at Scale

Chuanxiong Guo, Haitao Wu, Zhong Deng, Gaurav Soni,
Jianxi Ye, Jitendra Padhye, Marina Lipshteyn
Microsoft
{chguo, hwu, zdeng, gasoni, jiye, padhye, malipsht}@microsoft.com

ABSTRACT

Over the past one and half years, we have been using RDMA over commodity Ethernet (RoCEv2) to support some of Microsoft's highly-reliable, latency-sensitive services. This paper describes the challenges we encountered during the process and the solutions we devised to address them. In order to scale RoCEv2 beyond VLAN, we have designed a DSCP-based priority flow-control (PFC) mechanism to ensure large-scale deployment. We have addressed the safety challenges brought by PFC-induced deadlock (yes, it happened!), RDMA transport livelock, and the NIC PFC pause frame storm problem. We have also built the monitoring and management systems to make sure RDMA works as expected. Our experiences show that the safety and scalability issues of running RoCEv2 at scale can all be addressed, and RDMA can replace TCP for intra data center communications and achieve low latency, low CPU overhead, and high throughput.

CCS Concepts

•**Networks** → **Network protocol design; Network experimentation; Data center networks;**

Keywords

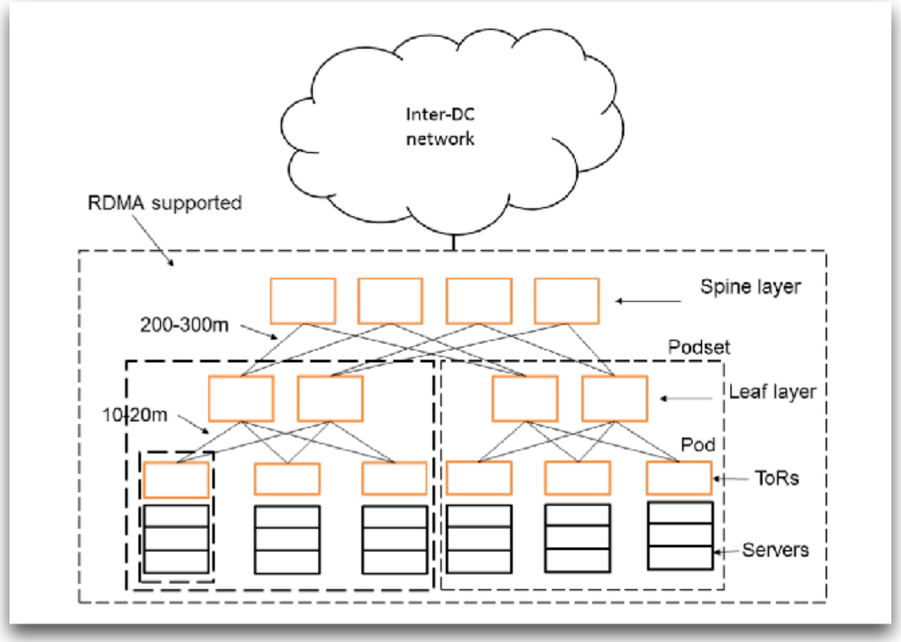
RDMA; RoCEv2; PFC; PFC propagation; Deadlock

Ethernet switches and network interface cards (NICs). A state-of-the-art DCN must support several Gb/s or higher throughput between any two servers in a DC.

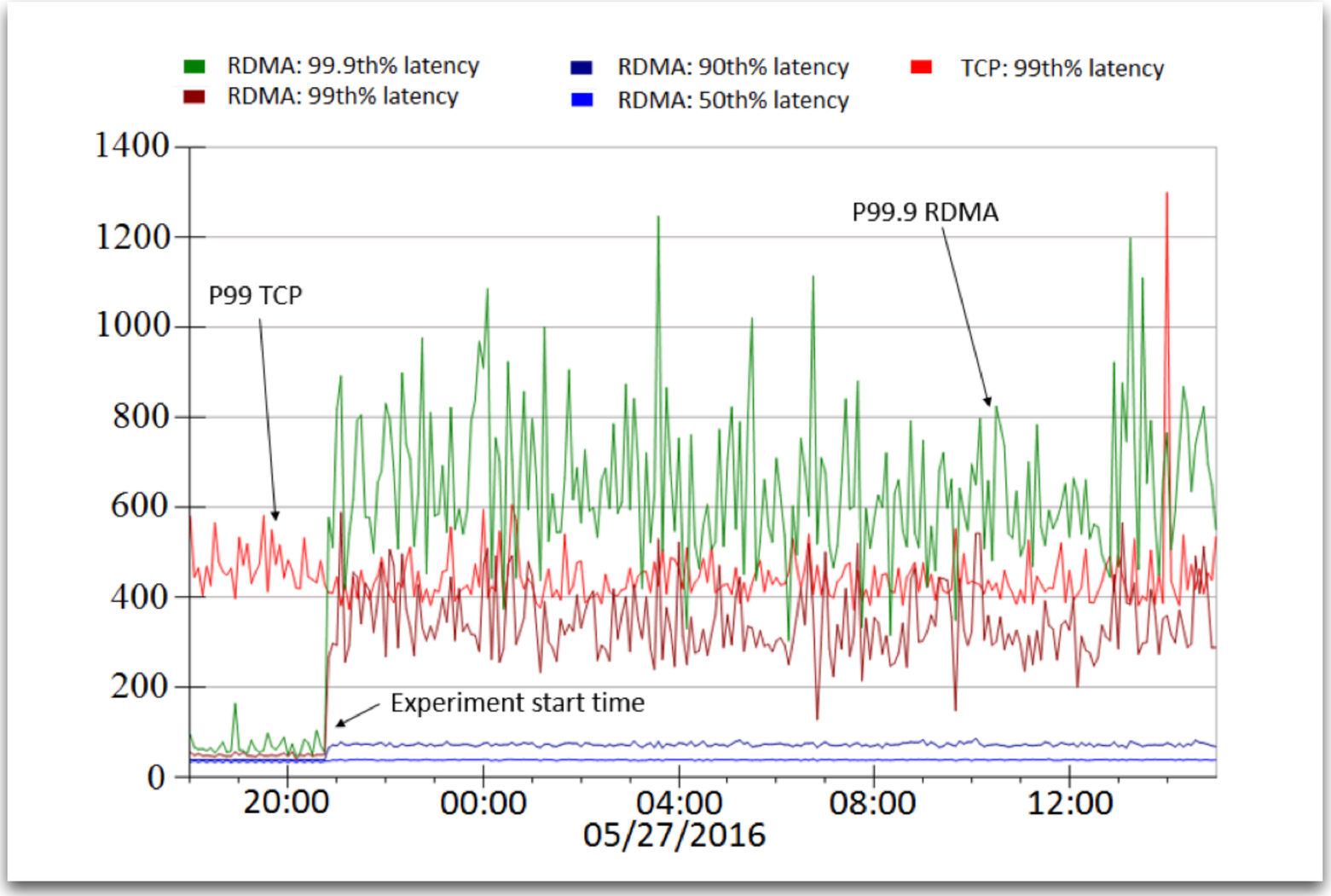
TCP/IP is still the dominant transport/network stack in today's data center networks. However, it is increasingly clear that the traditional TCP/IP stack cannot meet the demands of the new generation of DC workloads [4, 9, 16, 40], for two reasons.

First, the CPU overhead of handling packets in the OS kernel remains high, despite enabling numerous hardware and software optimizations such as checksum offloading, large segment offload (LSO), receive side scaling (RSS) and interrupt moderation. Measurements in our data centers show that sending at 40Gb/s using 8 TCP connections chews up 6% aggregate CPU time on a 32 core Intel Xeon E5-2690 Windows 2012R2 server. Receiving at 40Gb/s using 8 connections requires 12% aggregate CPU time. This high CPU overhead is unacceptable in modern data centers.

Second, many modern DC applications like Search are highly latency sensitive [7, 15, 41]. TCP, however, cannot provide the needed low latency even when the average traffic load is moderate, for two reasons. First, the kernel software introduces latency that can be as high as tens of milliseconds [21]. Second, packet drops due to congestion, while rare, are not entirely absent in our data centers. This occurs because data center traffic is inherently bursty. TCP must recover from the



Large-Scale RDMA Deployments, **but**



What do applications fundamentally need?

Application Demands

- #1: Location Independence
  - It shouldn’t matter where the distributed application is running

Application Demands

- #1: Location Independence
  - It shouldn’t matter where the distributed application is running
- #2: Low Latency
  - Predictable very low latency request/response behavior

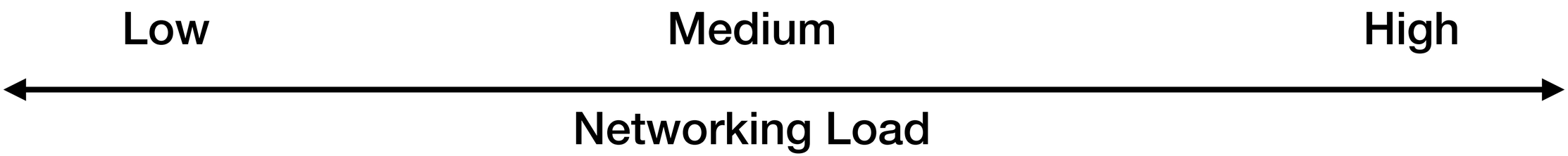
Application Demands

- #1: Location Independence
  - It shouldn’t matter where the distributed application is running
- #2: Low Latency
  - Predictable very low latency request/response behavior
- #3: Incast
  - Partition/Aggregation-style execution model

Application Demands

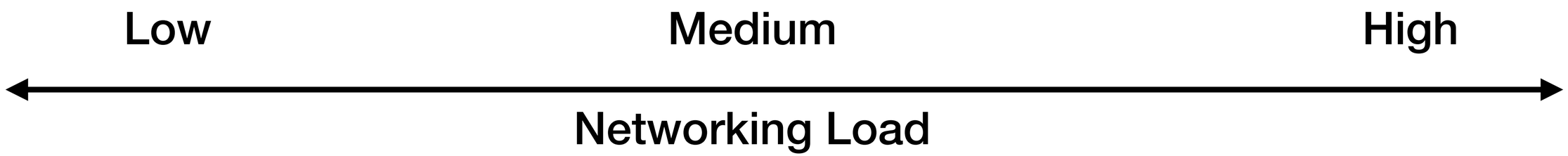
- #1: Location Independence
  - It shouldn’t matter where the distributed application is running
- #2: Low Latency
  - Predictable very low latency request/response behavior
- #3: Incast
  - Partition/Aggregation-style execution model
- #4: Priority
  - The receiver prioritizes arriving traffic from stragglers

Design Space Exploration



Design Space Exploration

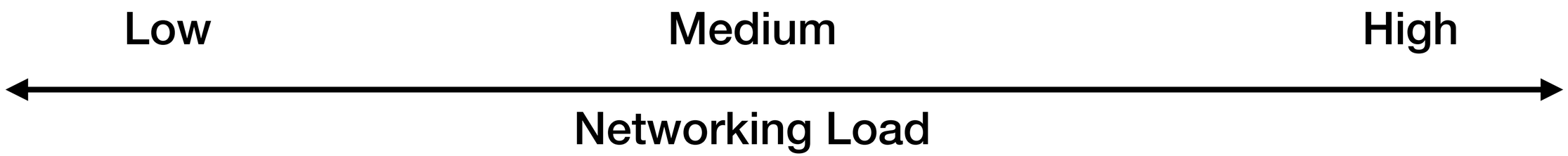
- Quickly max out the transmission speed



Design Space Exploration

- Quickly max out the transmission speed

- Zero-RTT connection setup
- Fast start

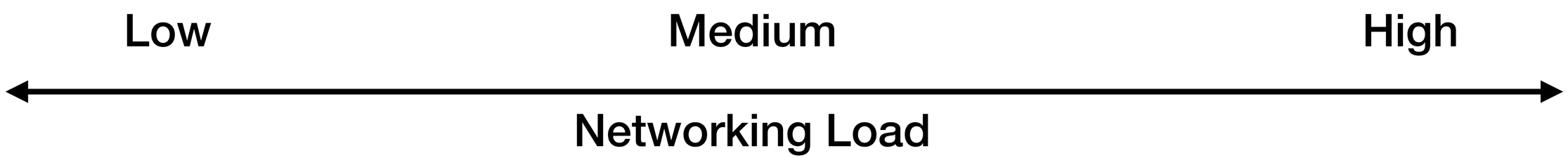


Design Space Exploration

- Quickly max out the transmission speed

- Fully explore the link availability

- Zero-RTT connection setup
- Fast start



Design Space Exploration

- Quickly max out the transmission speed

- Fully explore the link availability

- Zero-RTT connection setup
- Fast start

- Per-packet ECMP
- Reorder-tolerant handshake

Design Space Exploration

- Quickly max out the transmission speed

- Fully explore the link availability

- Timely throttling

- Zero-RTT connection setup
- Fast start

- Per-packet ECMP
- Reorder-tolerant handshake

Design Space Exploration

- Quickly max out the transmission speed

- Fully explore the link availability

- Timely throttling

- Zero-RTT connection setup
- Fast start

- Per-packet ECMP
- Reorder-tolerant handshake

- Optimized for incast

Low **Medium** **High**

Networking Load

Design Space Exploration

How to detect the networking load?

Design Space Exploration

How to detect the networking load?

- **Packet loss**

Design Space Exploration

How to detect the networking load?

- **Packet loss**
- **ECN**

Design Space Exploration

How to detect the networking load?

- **Packet loss**
- **ECN**
- **Priority-based flow control (PFC)**

Design Space Exploration

How to detect the networking load?

- **Packet loss**
- **ECN**
- **Priority-based flow control (PFC)**
- **Cut Payload**

Design Space Exploration

How to detect the networking load?

- Packet loss
- ECN
- Priority-based flow control (PFC)
- *Cut Payload*

How does NDP work?

#1: Connection Setup @Endhost

#1: Connection Setup @Endhost

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

#1: Connection Setup @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

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

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

- TCP: slow start
  - Initially with a small congestion window
  - Double it each RTT until filling the pipe

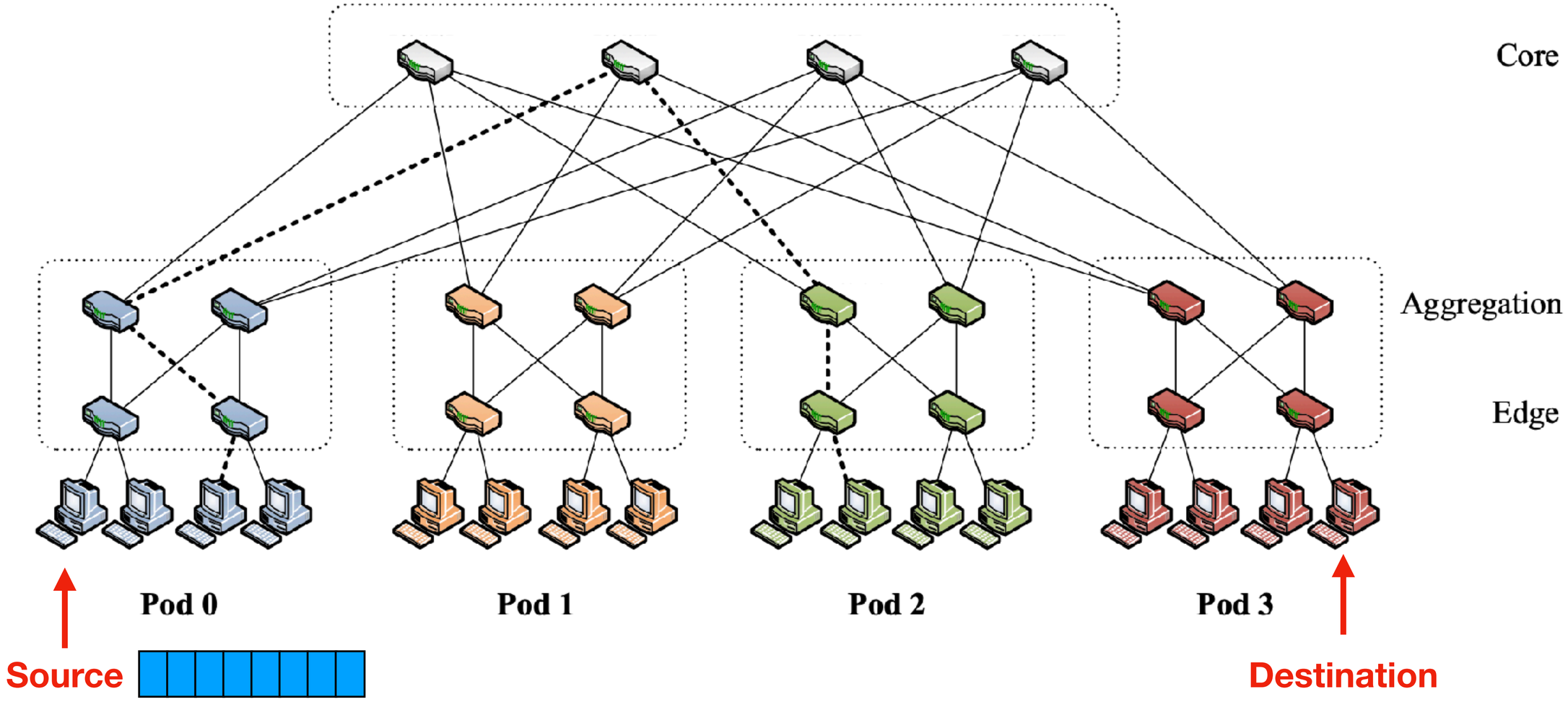
#2: Data Transmission in the 1st RTT @Endhost

- TCP: slow start
  - Initially with a small congestion window
  - Double it each RTT until filling the pipe

- NDP: full bandwidth-delay product (BDP)
  - Network utilization is relatively low
  - Link speeds and baseline RTTs are much more homogeneous
  - Network conditions can be known in advance

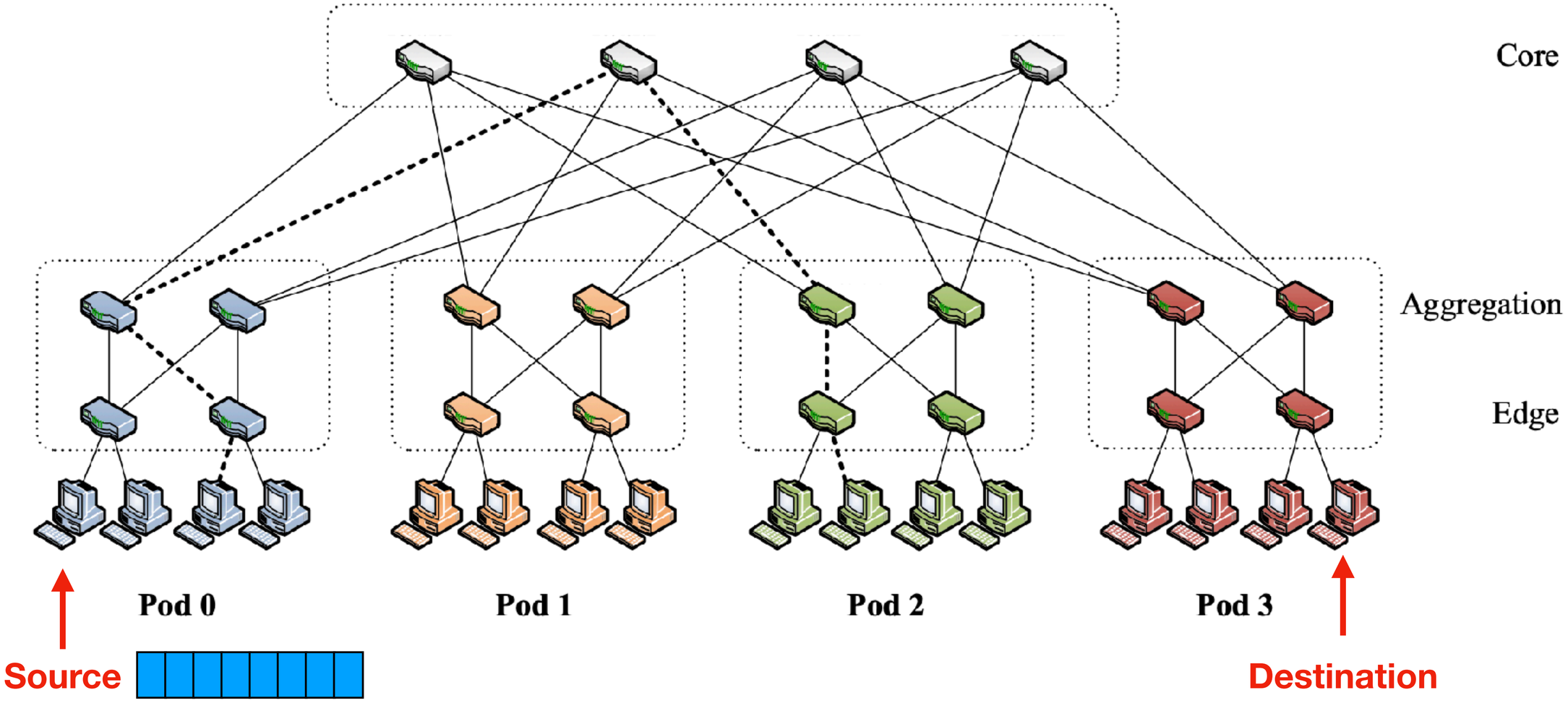
#3: Routing @Switch

- Default approach: per-flow routing



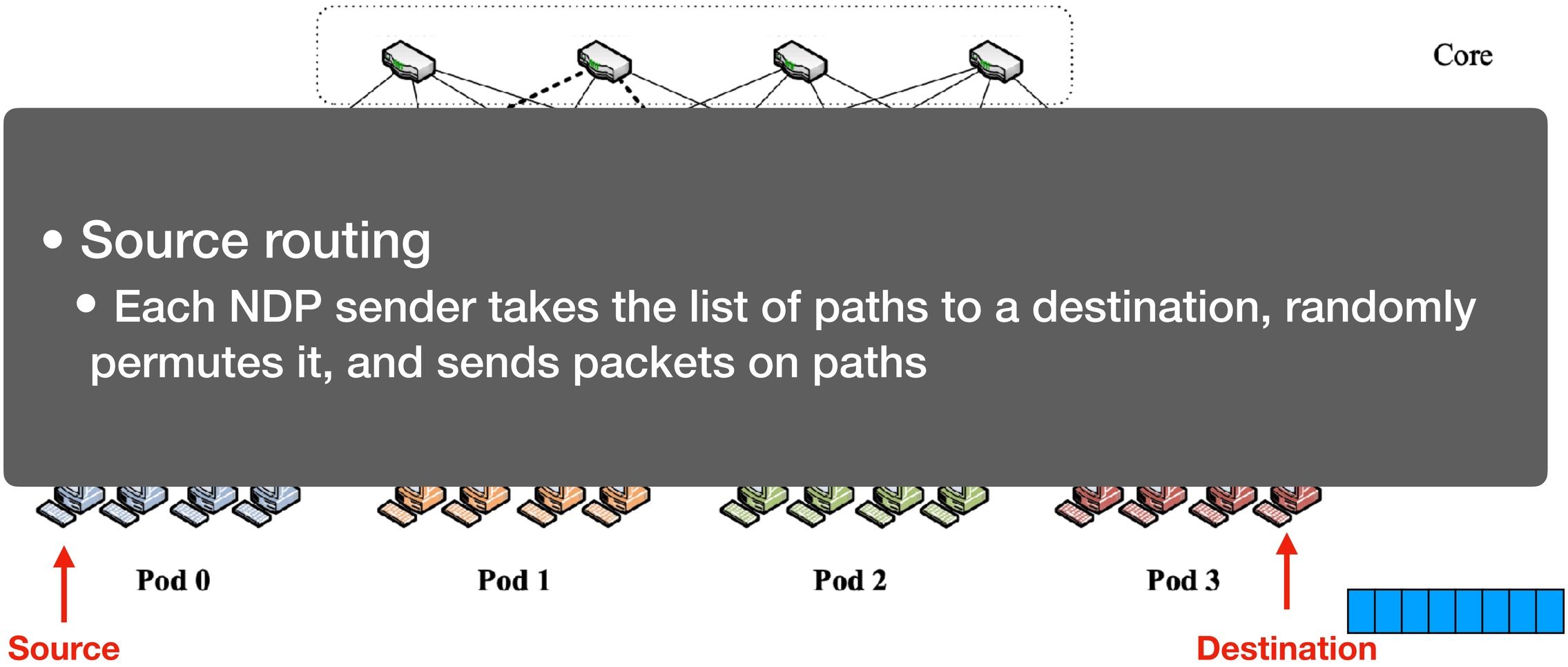
#3: Routing @Switch

- NDP: per-packet routing



#3: Routing @Switch

- NDP: per-packet routing



- Source routing
  - Each NDP sender takes the list of paths to a destination, randomly permutes it, and sends packets on paths

What happens if the switch queue is full?

#3: Queueing Handling @Switch

#3: Queueing Handling @Switch

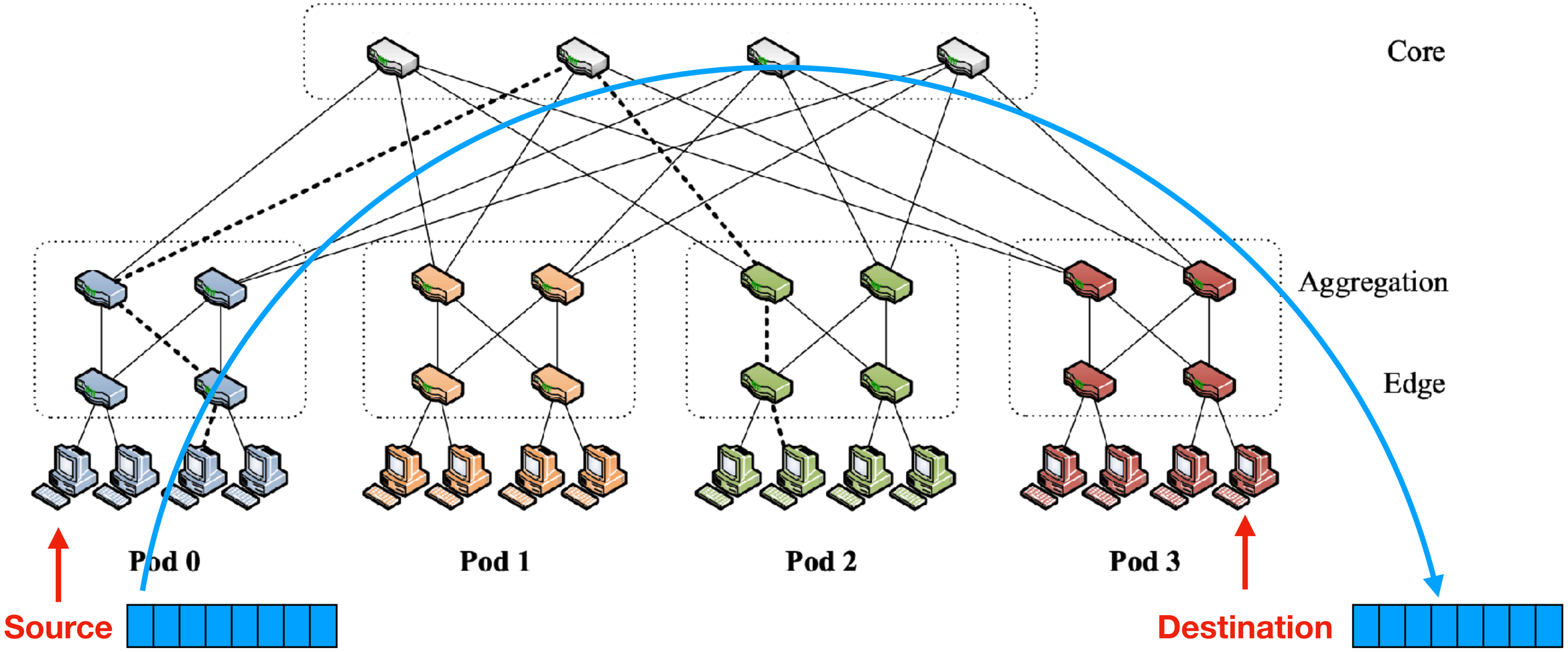
- TCP: drop => congestion signal
  - RED (random early drop)
  - DCTCP employs ECN

#3: Queueing Handling @Switch

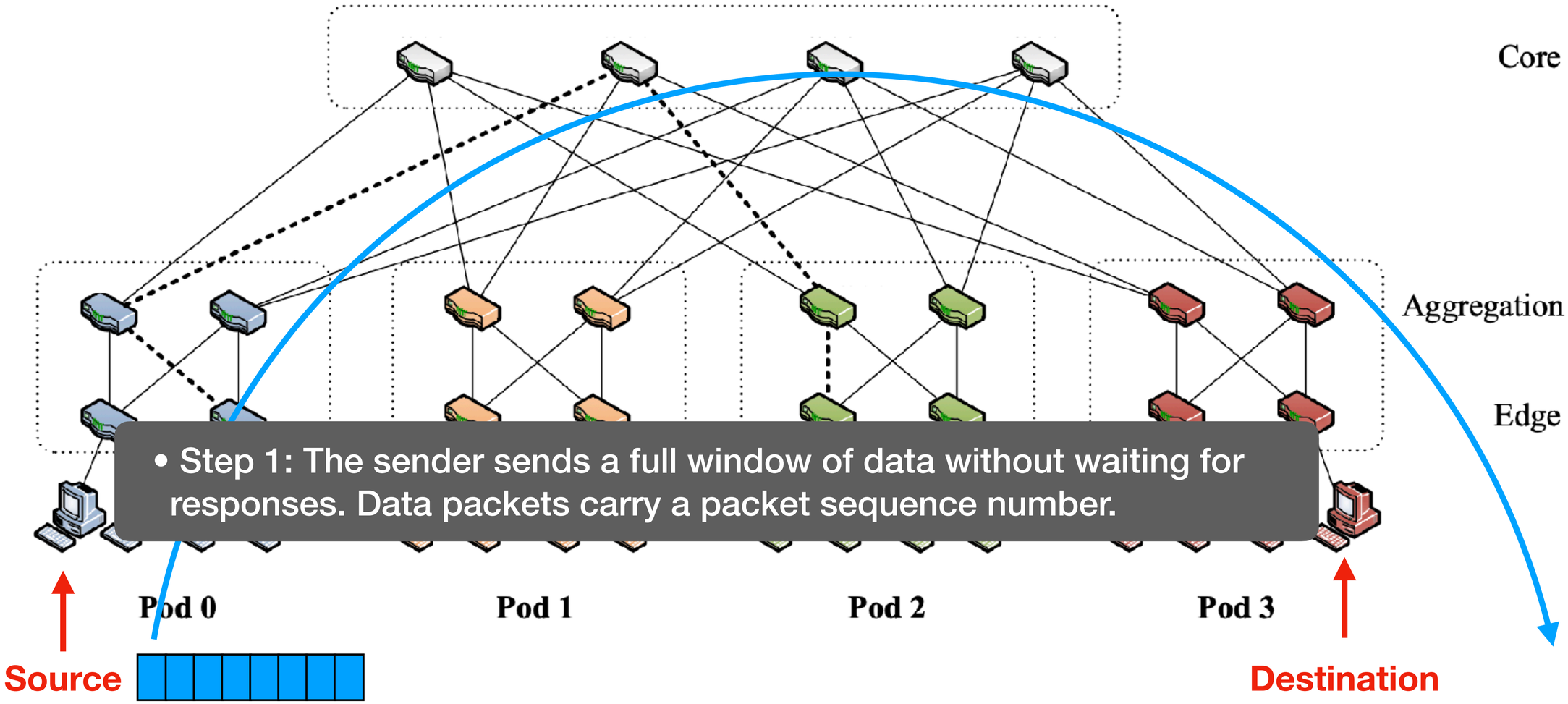
- TCP: drop => congestion signal
  - RED (random early drop)
  - DCTCP employs ECN

- NDP: packet trimming
  - Headers of trimmed packets are used for traffic control
  - Co-design the switch behavior with the transport protocol

#4: NDP Transport @Switch+Endhost

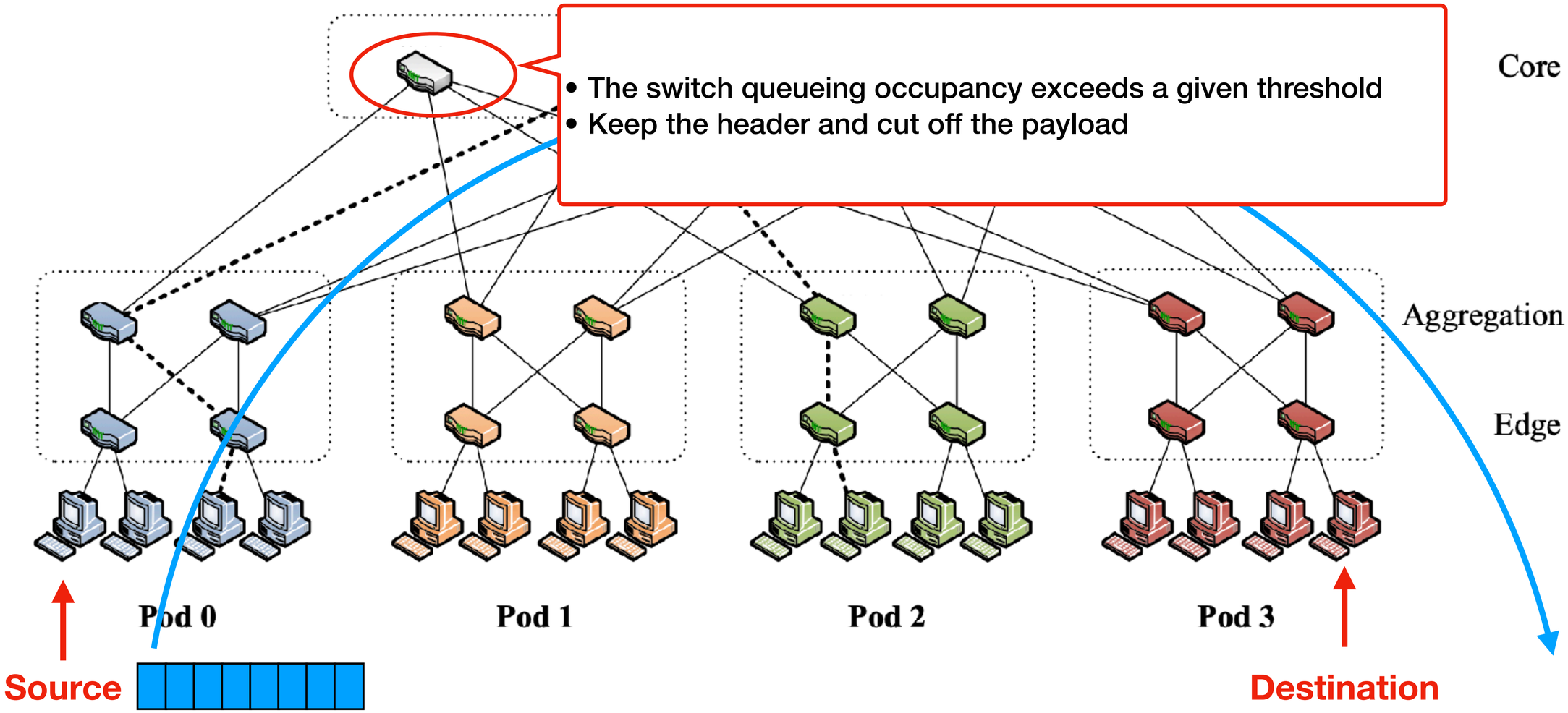


#4: NDP Transport @Switch+Endhost

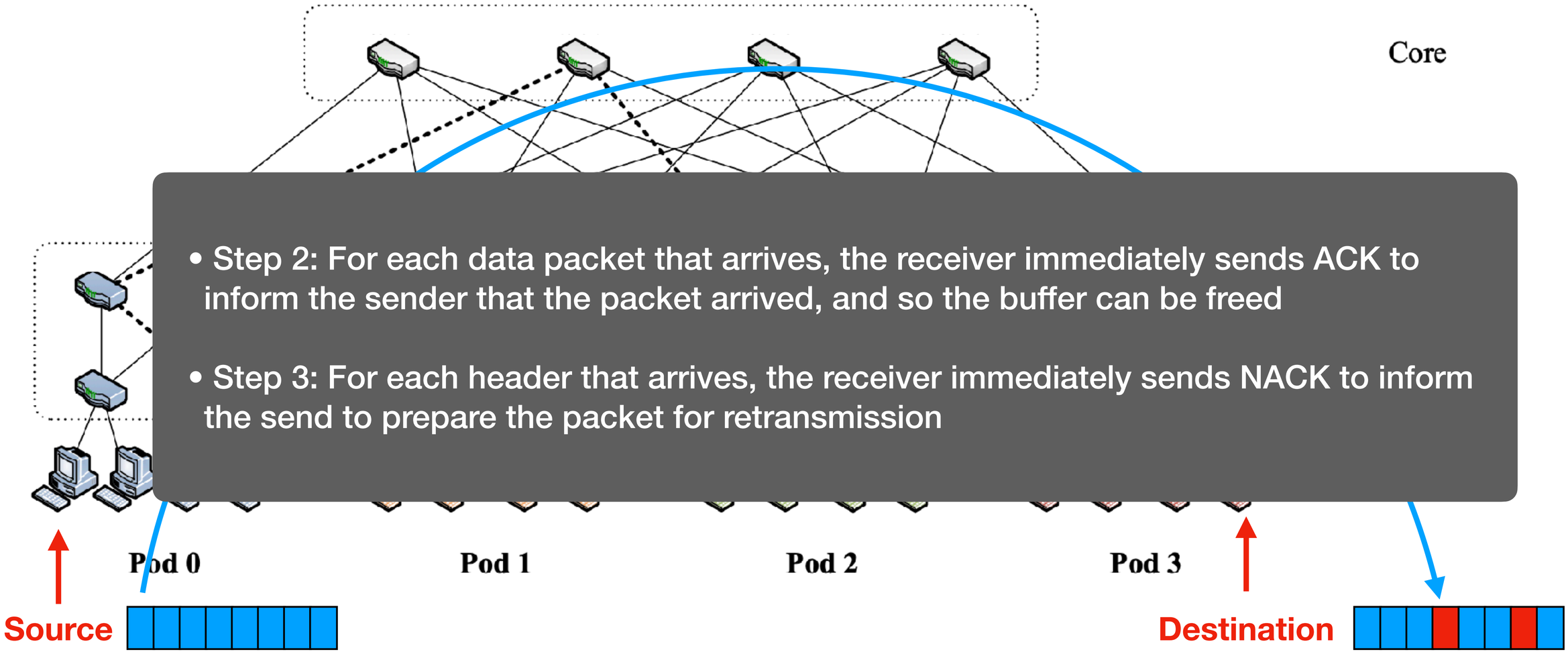


- Step 1: The sender sends a full window of data without waiting for responses. Data packets carry a packet sequence number.

#4: NDP Transport @Switch+Endhost

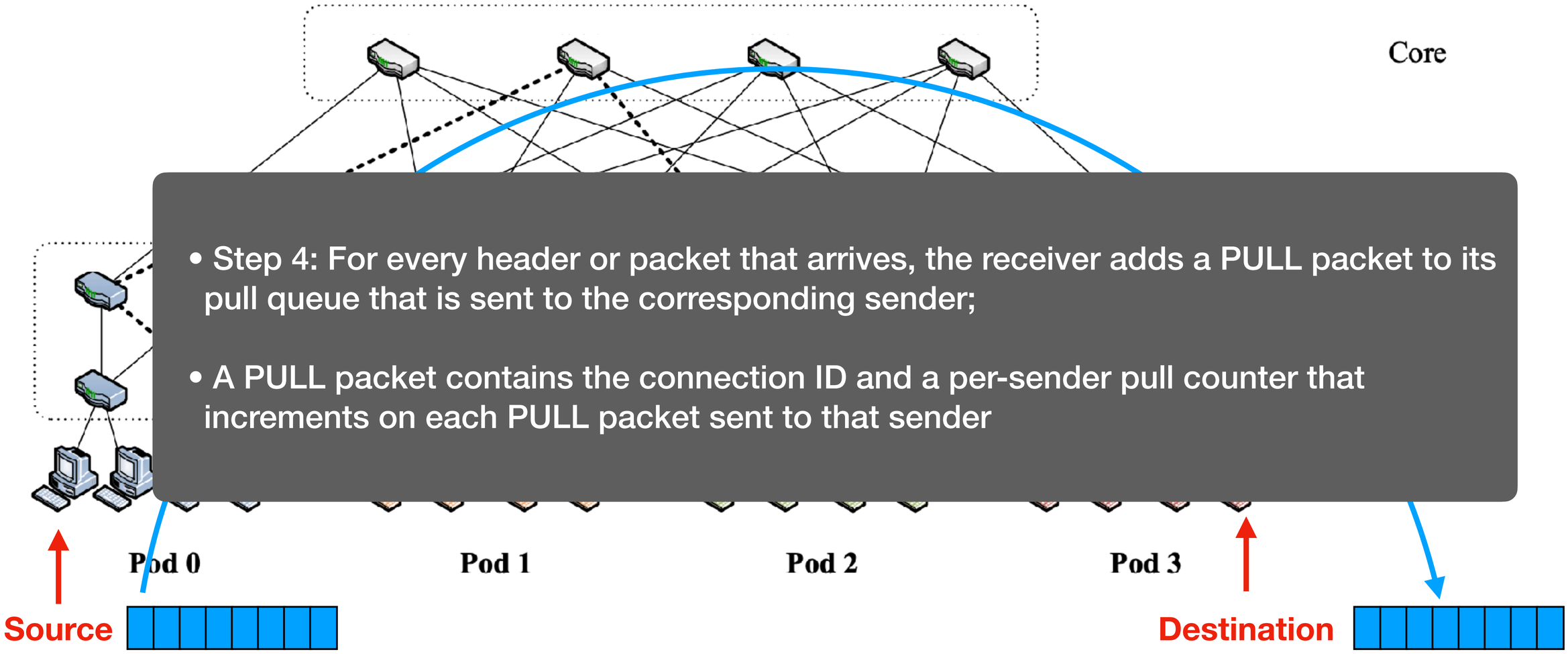


#4: NDP Transport @Switch+Endhost



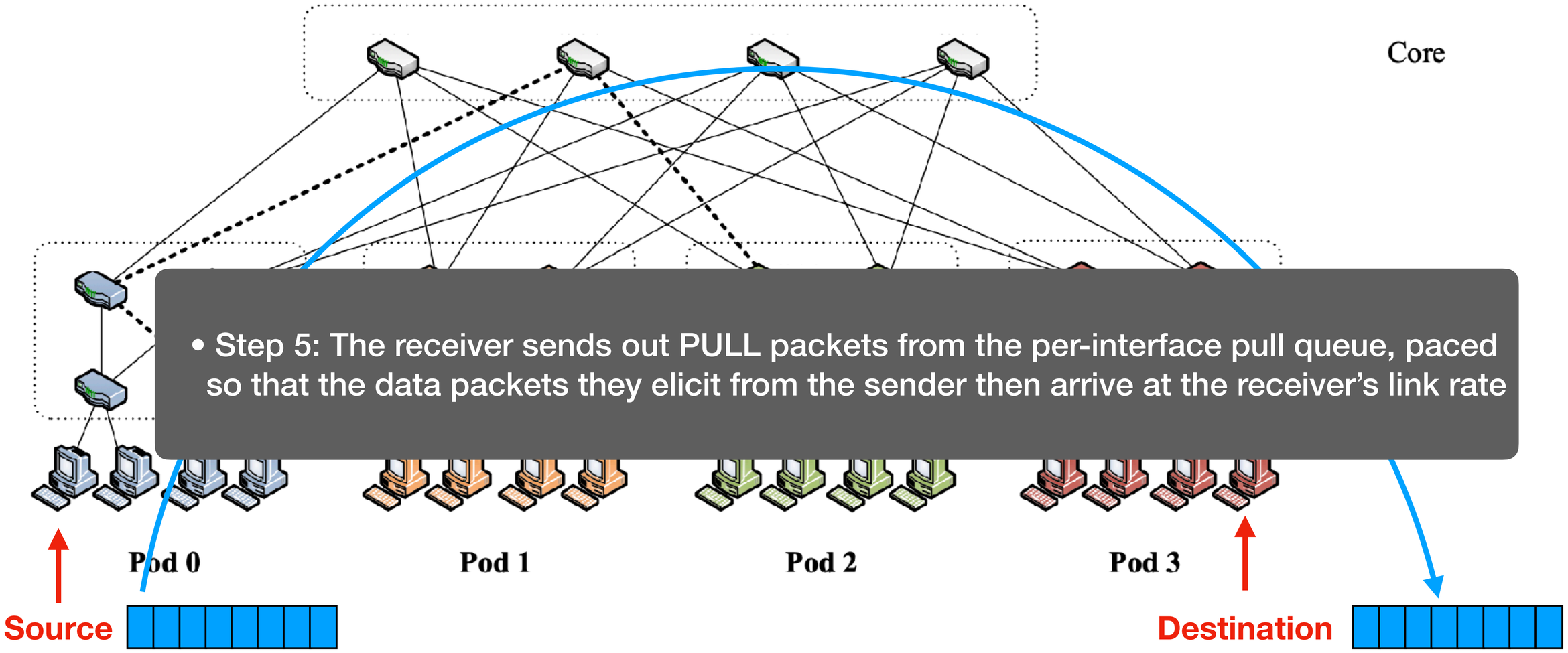
- Step 2: For each data packet that arrives, the receiver immediately sends ACK to inform the sender that the packet arrived, and so the buffer can be freed
- Step 3: For each header that arrives, the receiver immediately sends NACK to inform the send to prepare the packet for retransmission

#4: NDP Transport @Switch+Endhost



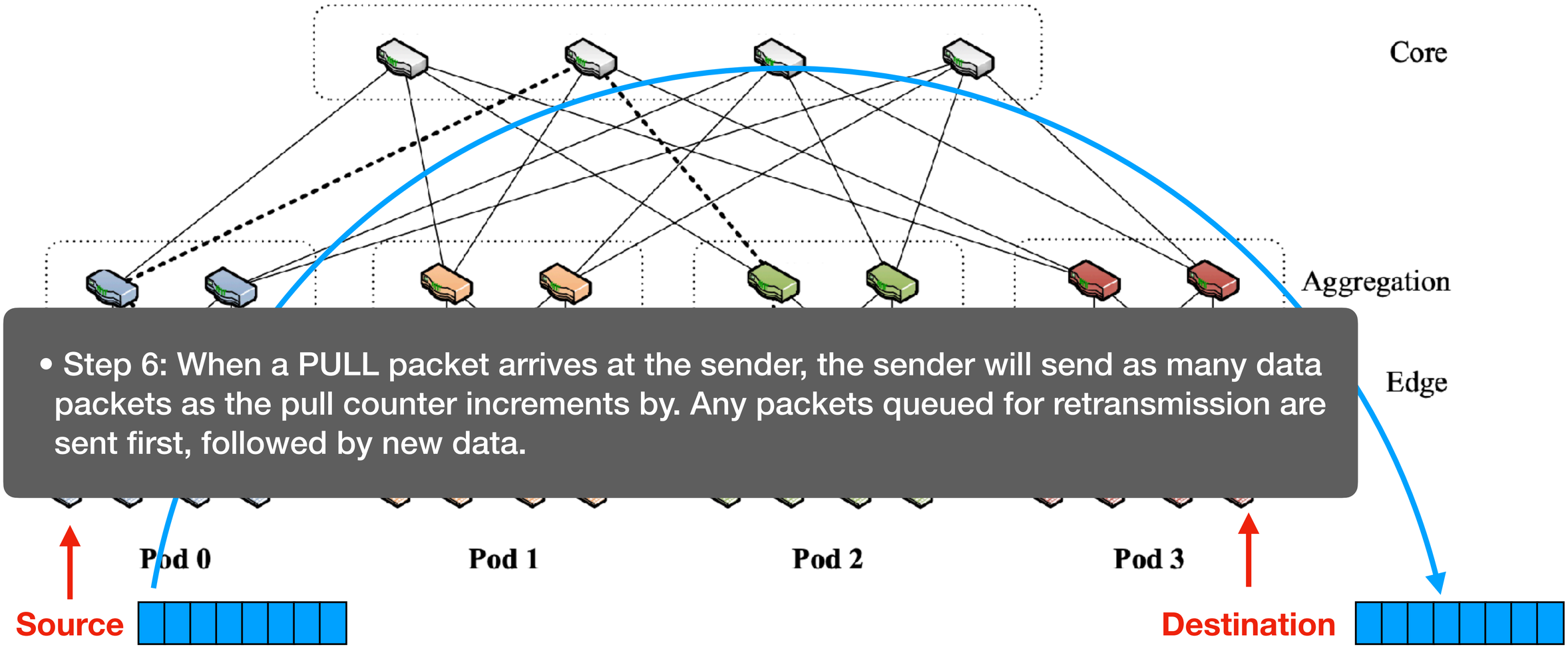
- Step 4: For every header or packet that arrives, the receiver adds a PULL packet to its pull queue that is sent to the corresponding sender;
- A PULL packet contains the connection ID and a per-sender pull counter that increments on each PULL packet sent to that sender

#4: NDP Transport @Switch+Endhost



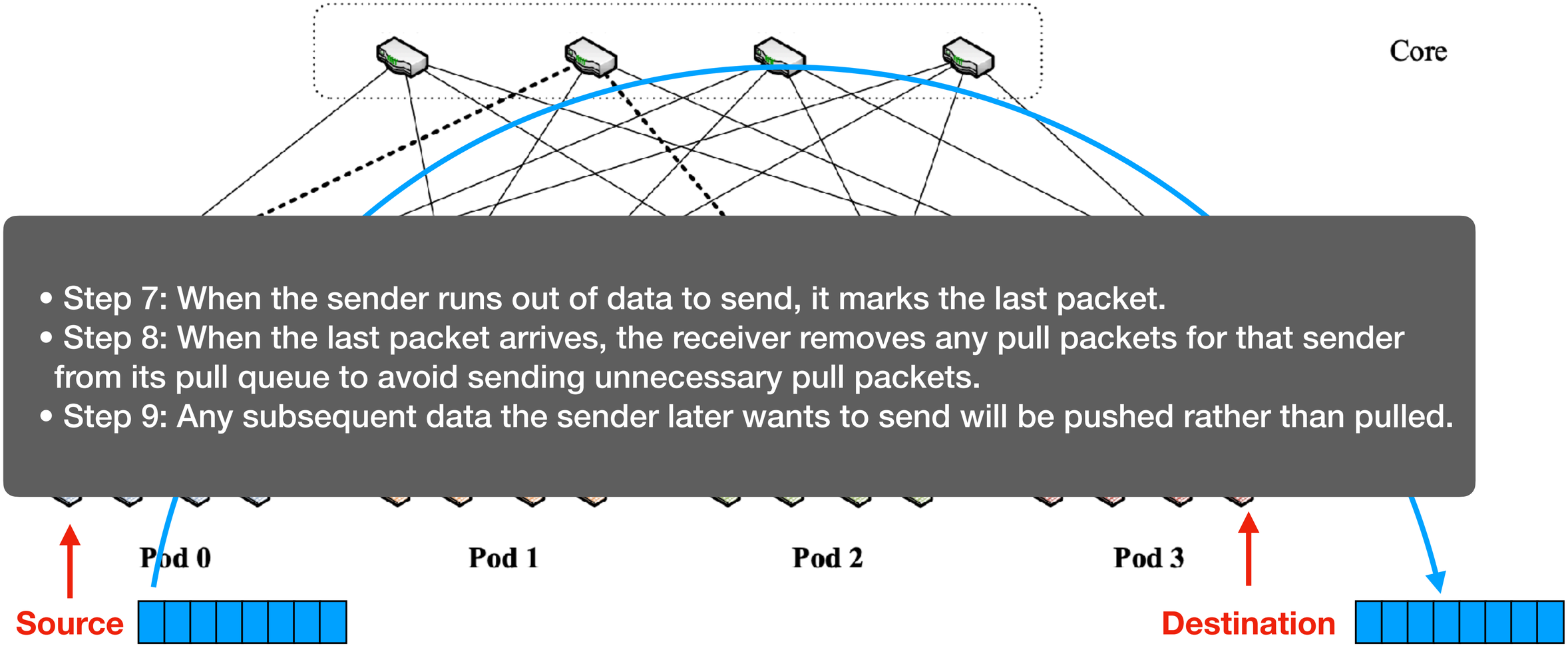
- Step 5: The receiver sends out PULL packets from the per-interface pull queue, paced so that the data packets they elicit from the sender then arrive at the receiver's link rate

#4: NDP Transport @Switch+Endhost



- Step 6: When a PULL packet arrives at the sender, the sender will send as many data packets as the pull counter increments by. Any packets queued for retransmission are sent first, followed by new data.

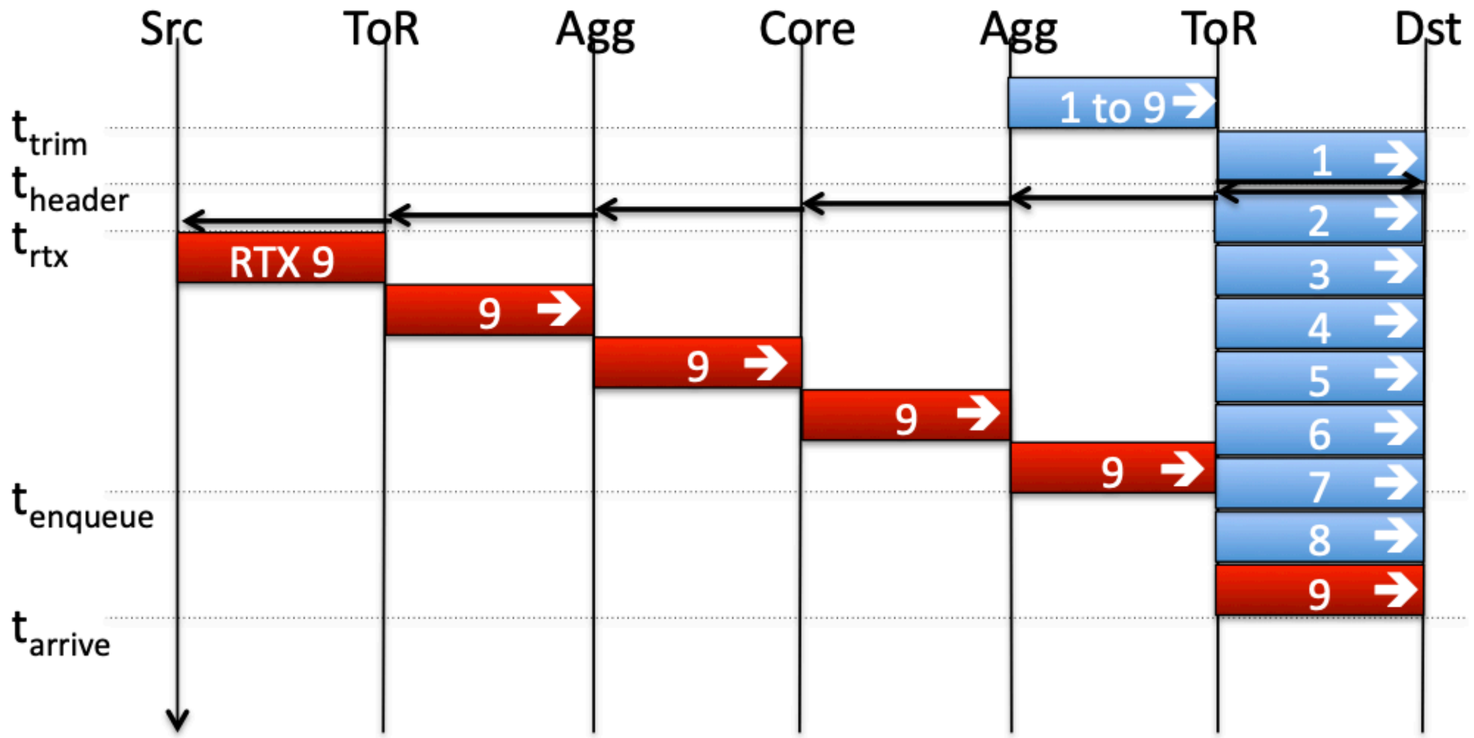
#4: NDP Transport @Switch+Endhost



- Step 7: When the sender runs out of data to send, it marks the last packet.
- Step 8: When the last packet arrives, the receiver removes any pull packets for that sender from its pull queue to avoid sending unnecessary pull packets.
- Step 9: Any subsequent data the sender later wants to send will be pushed rather than pulled.

Prioritized Trimmed Packets

- Low-delay retransmission is ensured



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

#5: Reorder Handling @Endhost

#5: Reorder Handling @Endhost

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

#5: Reorder Handling @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

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

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

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

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

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

- NDP: N/A
  - PULL packets are used for pacing
  - Trimmed packets are used for handling congestion collapse

What happens if links or switches fail?

#7: Failure Handling @Endhost

#7: Failure Handling @Endhost

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

#7: Failure Handling @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
  - Return-to-sender optimization kicks in when incast happens

Is NDP fair?

#8: Fairness Guarantee @Endhost

#8: Fairness Guarantee @Endhost

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

#8: Fairness Guarantee @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

How does NDP achieve the following?

- Quickly max out the transmission speed
- Zero-RTT connection setup
- Fast start

- Fully explore the link availability
- Per-packet ECMP
- Reorder-tolerant handshake

- Timely throttling
- Optimized for incast

Summary

- Today
  - NDP
- Next
  - Homa (Sigcomm’18)