

Gleam: An RDMA-accelerated Multicast Protocol for Datacenter Networks

Wenxue Li^{1*} Junyi Zhang^{2*} Gaoxiong Zeng² Yufei Liu² Zilong Wang¹
Chaoliang Zeng¹ Pengpeng Zhou² Qiaoling Wang² Kai Chen¹

¹Hong Kong University of Science and Technology ²Huawei

Abstract

RDMA has been widely adopted for high-speed datacenter networks. However, native RDMA merely supports one-to-one reliable connection, which mismatches various applications with group communication patterns (*e.g.*, one-to-many). While there are some multicast enhancements to address it, they all fail to simultaneously achieve optimal multicast forwarding and fully unleash the distinguished RDMA capabilities.

In this paper, we present Gleam, an RDMA-accelerated multicast protocol that simultaneously supports optimal multicast forwarding, efficient utilization of the prominent RDMA capabilities, and compatibility with the commodity RNICs. At its core, Gleam re-purposes the existing RDMA RC logic with careful switch coordination as an efficient multicast transport. Gleam performs the one-to-many connection maintenance and many-to-one feedback aggregation, based on an extended multicast forwarding table structure, to achieve integration between standard RC logic and in-fabric multicast. We implement a fully functional Gleam prototype. With extensive testbed experiments and simulations, we demonstrate Gleam’s significant improvement in accelerating multicast communication of realistic applications. For instance, Gleam achieves $2.9\times$ lower communication time of an HPC benchmark application and $2.7\times$ higher data replication throughput.

1 Introduction

Datacenter applications impose increasingly stringent requirements for network communication, such as persistently high throughput, ultra-low latency (μs scale), and low CPU overhead (to cut OpEx). To meet it, Remote Direct Memory Access (RDMA) is emerging as the de-facto networking technology going beyond 40Gbps links. Many tech giants have adopted RDMA into their production datacenters [30, 32, 40, 47]. These datacenters host various network-intensive applications, such as deep learning [34, 36], cloud storage [30, 35], graph exploration [45], *etc.*, which benefit greatly from the underlying RDMA communication.

However, native RDMA transports only support one-to-one reliable connection (RC) [2], which mismatches various applications with group communication patterns [22, 30, 36] (*e.g.*,

one-to-many). Multicast is prevalent in datacenter applications (§2.2). Specifically, previous work shows that multicast is the top-2 communication pattern in a High-performance Computing (HPC) cluster [19]. In High-performance Linpack (HPL) [3], an HPC’s benchmark application, more than 90% communication traffic is the multicast pattern.

To meet applications’ requirements, a straightforward solution is to provide a multicast primitive. However, while there are some existing multicast solutions [1, 10, 12, 44] (§2.3), none of them simultaneously achieve two of the performance requirements: 1) optimal forwarding multicast traffic; 2) fully unleashing the distinguished RDMA capabilities.

On one hand, some primary distributed frameworks (*e.g.*, MPI [12] and NCCL [10]) choose to develop their private multicast protocols upon the RDMA one-to-one RC transport (*a.k.a.*, application-layer multicast). Thus they can efficiently utilize the prominent RDMA capabilities. However, application-layer multicast cannot achieve optimal traffic forwarding, resulting in inefficient bandwidth utilization and communication bottleneck [25, 44]. Although many multicast algorithms are proposed to mitigate the bottleneck (*e.g.*, Ring and Double-binary tree [4, 11, 31]), they inevitably incur longer communication distance and higher latency resulting from intermediate nodes’ data forwarding.

On the other hand, in-fabric multicast can achieve optimal multicast forwarding with efficient bandwidth utilization and minimized communication distance. However, the in-fabric multicast cannot benefit from advanced RDMA capabilities. In particular, IP-based multicast [20, 25, 44] only supports layer-3 routing without any transport-layer functionality, thus is incompatible with the RDMA RC transport. IB multicast, defined in IB standard [1], only supports UD transport. Therefore, applications that adopt IB multicast cannot utilize the advanced one-sided WRITE¹, the extended message size, and the hardware-supported reliability.

In addition to the performance requirements, the multicast solution should also meet the practical deployment requirements in production environments. The fundamental challenge comes from the commodity RDMA Network Interface Cards (RNICs). Commodity RNICs provide limited capability for flexible programming [30, 47]. For instance, the complete network stacks of [8] are statically built-in with specialized

*Co-first authors.

¹The RDMA READ doesn’t fit into the multicast context.

circuits.

In this work, we focus on simultaneously (i) supporting the optimal multicast forwarding, (ii) fully exploiting the prominent features of RDMA, and (iii) satisfying the deployment requirements. To this end, we propose Gleam, an RDMA-accelerated multicast solution for datacenter networks (§3). Firstly, Gleam inherits the classical in-fabric distribution manner [20, 25, 44] to achieve optimal multicast forwarding. Secondly, Gleam re-purposes the existing RDMA RC logic with careful switch coordination to process multicast traffic, unleashing RDMA’s superior competencies. Finally, as Gleam reuses the standard RC transport, it is compatible with the large-scale deployed commodity RNICs.

The key challenge behind Gleam is how to integrate the optimal-forwarded multicast traffic with the existing RC logic (§3.1). Specifically, there are two main issues. Firstly, the current connection-oriented logic [2] is targeted for one-to-one connection, which cannot directly support one-to-many data delivery. Secondly, the existing reliability logic [32, 47] is designed for single feedback (including ACK, NACK, CNP, etc.) stream from a single receiver; thus, multiple feedback streams in multicast can confuse it and degrade the overall performance.

To reuse the connection-oriented logic of RC (§3.3), Gleam elaborately extends the widely-adopted multicast forwarding table structure [20]. Then, based on the extended table, Gleam replaces the connection-related header fields in multicast data packets to match different QPs/connections² on different receivers. As a result, Gleam achieves a virtual one-to-many connection among multicast members, providing applications with the advanced capability of RDMA connected transport service.

To reuse the reliability logic of RC (§3.4), Gleam performs the many-to-one feedback aggregation in the fabric. In particular, Gleam aggregates ACK so that the sender receives a unicast-like ACK stream, which is compatible with the existing ACK-interpretation logic. Moreover, Gleam carefully filters NACK packets to enable the sender to correctly detect and retransmit the lost packet. Consequently, Gleam can reuse the standard reliability logic and provides hardware-based reliability.

We implement a fully functional Gleam switch (§4), connected to four commodity servers. Each server is equipped with an unmodified commodity RNIC. Gleam is evaluated through extensive testbed experiments and simulations (§5). The testbed experiments show that Gleam accelerates the multicast pattern by up to 2.2×, and improves the realistic application’s performance, such as 2.9× lower HPL communication time and 2.7× higher data replication throughput. Meanwhile, large-scale simulations demonstrate that Gleam can maintain a high performance in large-scale topologies and a satisfying goodput when packet loss occurs.

²We use QP and connection equivalently in this paper.

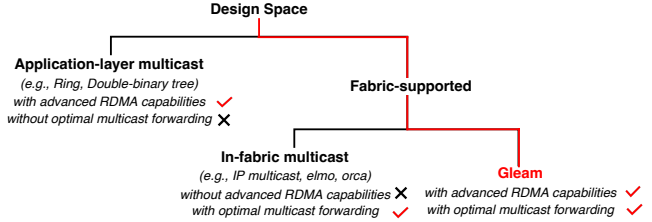


Figure 1: Design space for multicast solutions.

We summarize the design space for multicast solutions in Fig. 1. To the best of our knowledge, this is the first work to completely leverage RDMA capabilities to empower multicast communication. The key contributions of this work are summarized below:

- We observe that the reliable multicast delivery can be efficiently realized through fitting the multicast traffic into RDMA’s performant RC transport.
- We design Gleam, an RDMA-accelerated multicast protocol that simultaneously supports optimal multicast forwarding, advanced capabilities of RDMA, and compatibility with commodity RNICs. Gleam abstracts a virtual one-to-many connection and aggregates feedback in the middle fabric, which addresses the incompatibility between multicast traffic and existing RC logic.
- We implement a fully functional Gleam prototype. Extensive experiments demonstrate Gleam’s significant acceleration in multicast communication.

2 Background and Motivation

We first introduce some basic concepts of RDMA (§2.1); then present the prevalence of multicast pattern (§2.2); finally reveal the insufficiencies of existing solutions (§2.3).

2.1 RDMA Overview

RDMA is an emerging hardware-offloaded transport that implements the transport functionalities entirely in RNICs, including the basic packet encap/decap, reliability enhancements, congestion control, etc. As a consequence, RDMA provides high throughput, low latency, and efficient CPU utilization [47].

RDMA Transport Services. Commodity RNICs support three transport services: *reliable connection* (RC), *unreliable connection* (UC), and *unreliable datagram* (UD), each of which supports a different subset of RDMA operations. There are two types of operations: one-sided (WRITE and READ) and two-sided (SEND/RECEIVE). The one-sided operations don’t involve the remote CPU, while the two-sided operations require the remote CPU to participate.

Among the supported RDMA transport services, RC owns various advantages over UC and UD [27, 41]. Firstly, RC supports all RDMA operations, especially the one-sided WRITE,

while UD only supports SEND/RECEIVE. Additionally, RC provides hardware-supported reliability, which reduces the software overhead, while UC and UD lack this benefit. Finally, the message size limits are also different; the message of RC and UC can span multiple packets with a total size of up to 2GB, while the message of UD is limited to a single packet. We summarize the RDMA transport service characteristics in Table 1.

Queue Pair and Memory Region. RDMA applications communicate through *queue pairs* (QPs) and request a network communication by submitting a *work queue element* (WQE) to the associated QP via *verbs*. Each QP is identified by a QP number (QPN) and includes two queues: a *send queue* (SQ) and a *receive queue* (RQ). Each QP is usually associated with a *completion queue* (CQ), which is used for RNIC to post the *completion queue event* (CQE) that contains the completion status of the previous submitted WQE.

RDMA applications register the to-be-accessed memory area as *memory region* (MR) before communication starts. Each MR is associated with a *virtual address* (VA) and a pair of keys: *L_key* and *R_key*. The application utilizes the *L_key* (*R_key*) as authorization to access the local (remote) MR. As the one-sided operations don't involve the remote CPU, the related MR info is formatted into the header of the one-sided request's first packet.

2.2 Multicast Pattern

Datacenter applications are usually deployed in a distributed approach to gain more computation capability. The participants frequently communicate with others, representing group communication patterns. Multicast is a prevalent communication pattern that many applications exhibit, such as database [30, 39], telemetry and monitoring system [38], HPC [13] and machine learning system [34, 36]. We present examples in HPC and storage networks in detail below.

HPC. High-performance Linpack (HPL) [3] is a benchmark application used to rank the supercomputer's computing capacity [13]. HPL solves a linear system, $Ax = b$ (usually with a large order), by LU decomposition. The overall HPL procedure is divided into multiple epochs, each including three steps: *Panel Factorization*, *Panel Broadcast* and *Update*. The *Panel Broadcast* is a standard multicast transmission, and the *Update* includes a communication phase (called *Row Swap*) which can be implemented with multicast. As the volume of communicated traffic is large [26], an efficient multicast can significantly improve HPL's performance.

Storage network. Distributed storage networks offer high availability and durability using multiple replications [30]. Failures of storage devices are inevitable in large-scale clusters. Therefore, replicas are critical to prevent data loss. The replication delivery impacts application's QoS significantly, as it may last for minutes [39]. Thus, an efficient multicast ser-

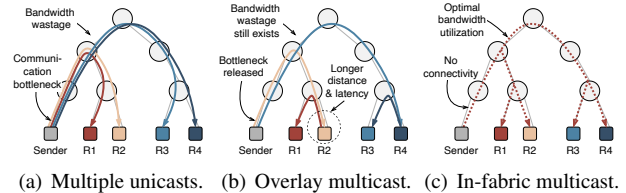


Figure 2: Existing multicast solutions.

	SEND/RECV	WRITE	READ	Message Size
RC	✓	✓	✓	2 GB
UC	✓	✓	✗	2 GB
UD	✓	✗	✗	4 KB

Table 1: RDMA operations and message size that supported in each RDMA transport.

vice can substantially improve the application's experience.

2.3 Insufficiencies of Existing Solutions

Existing multicast solutions cannot simultaneously achieve optimal multicast forwarding and efficient utilization of advanced RDMA capabilities.

Application-layer Multicast. Primary distributed computation frameworks, such as MPI [12] and NCCL [10], develop their private multicast protocol upon the RC transport service. Thus they can efficiently utilize the advanced features of RC. However, they cannot achieve optimal traffic forwarding resulting in inefficient communication performance. There are primarily two types of implementations: 1) the basic multiple sender-to-receiver unicasts (abbreviated as multiple unicasts in this paper), and 2) the overlay multicast.

As illustrated in Fig. 2(a), for multiple unicasts, the sender establishes multiple connections and transmits identical data to different receivers via multiple unicasts. This inevitably incurs severe bandwidth wastage and leads to a potential network bottleneck at the sender side. To address it, various overlay algorithms, e.g., Ring and Double-binary tree [4, 11, 31], are proposed to leverage a pipelined transmission strategy. Specifically, some intermediate receivers can relay data distribution after receiving data, thus spreading the transmission burden to all participants (Fig. 2(b)). However, the overlay multicast results in longer communication distance and higher latency because of intermediate nodes' forwarding. Specifically, in intermediate nodes, packets must go through RX stack, interrupt the CPU to make a forwarding decision, and finally go through TX stack to be sent out.

In-fabric Multicast. With in-fabric multicast, the sender only needs to send out one single copy of data; then, the fabric replicates and forwards the data to multiple receivers via a multicast distribution tree (Fig. 2(c)). However, existing solutions cannot unleash the full power of RDMA. In

particular, IP-based multicast [20, 25, 44] only supports layer-3 routing without any layer-4 functionality. Developing a specific layer-4 protocol in software upon existing IP-based multicast incurs additional CPU overhead and still forgo the efficient RDMA. IB multicast, defined in IB standard [1], only supports UD transport, inheriting all constraints of UD transport, as described in §2.1. Therefore, the insufficiencies of existing in-fabric multicast solutions make their large-scale deployment less attractive.

3 Gleam

We first describe the key ideas and design challenges of Gleam (§3.1). Then we overview the Gleam structure (§3.2), briefing its main components and how they work together. After that, we elaborate these design components one by one (§3.3-§3.5).

3.1 Key Ideas and Challenges

Gleam focuses on simultaneously achieving (i) the optimal multicast forwarding, (ii) the efficient utilization of the advanced capabilities of RDMA, and (iii) satisfying the deployment requirements. To this end, the key ideas behind Gleam are to (i) perform the optimal multicast forwarding in the in-fabric distribution manner [20, 25, 44]; and (ii) re-purpose the native RDMA RC logic with careful switch coordination for an efficient multicast transport.

However, there are critical challenges blocking the way: *how to achieve integration between the optimal multicast forwarding and the existing RDMA RC logic*. Specifically, there are two main compatibility issues.

The first is that the connection-oriented logic [2] of RC cannot find the associated QPs when receiving the traditional-forwarded multicast packets, illustrated in Fig. 3. During the traditional multicast forwarding, switches won't change the packet's layer-4 header. Switches either copy the entire packet [1, 20] or only modify the layer-3 (IP) header for layer-3 multicast routing [25, 44]. Thus all packets contain an identical layer-4 header, which only matches at most one connection. The non-matched RNIC will discard these packets as it cannot find the associated QP and Queue Pair Context (QPC) based on the non-matched layer-4 header.

Secondly, even if receivers can accept the packets and find the associated QPs, the second incompatibility impeding the leverage of RC is the existing reliability logic [32, 47]. The standard reliability logic is designed for single feedback (including ACK, NACK, CNP, etc.) stream from a single receiver. Thus, multiple feedback streams from multiple receivers can confuse RC and disturb its loss detection and retransmission routines.

Gleam is a fabric-supported multicast protocol that substantially differs from the traditional layer-3 in-fabric approaches. Gleam systematically integrates its design components to address these two incompatibilities.

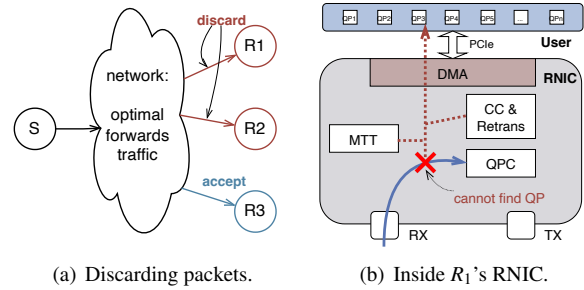


Figure 3: Traditional-forwarded traffic is incompatible with the existing connection-oriented logic of RDMA. R_1 and R_2 cannot find the associated QPs, so they discard packets. The dotted lines in (b) mean non-reachable.

3.2 Gleam Overview

Fig. 4 illustrates the architecture of Gleam. The working steps of Gleam are composed of three main phases: the control-plane multicast group registration (① & ②), the data-plane one-to-many data forwarding (③), and the data-plane many-to-one feedback aggregation (④).

For the control-plane multicast group registration, Gleam elaborately extends the legacy multicast forwarding table structure by integrating layer-4 states. We develop an out-of-band UDP-based protocol called *envelope* to register forwarding table to switches—the master node³ in the multicast group collects the layer-4 states of other nodes, fits these states into the *envelope* packet format, and transmits to switches and other nodes for building forwarding table and affirming the multicast membership, respectively (①). The involved nodes will answer ACKs to the master node to confirm its participation (②). Due to space limitations, we present the extended forwarding table structure in the main body, leaving the remaining control-plane details in Appendix A.

For the data-plane one-to-many data forwarding (§3.3), Gleam reserves the optimal multicast forwarding and achieves a virtual one-to-many connection. Let's take the multicast communication in Fig. 4 for an example. S only transmits data once via the existing RC connection. Then the switches in the multicast tree copy data and forward them to multiple receivers via the optimal paths. In addition, some specific switches replace the connection-related states in the packet header to match different QPs in different receivers (③), thus achieving a virtual one-to-many connection. For instance, L_1 , S_1 , C_2 , and S_3 copy and forward data packets to specific ports that are identified in the forwarding table. Additionally, L_2 , L_3 , and L_4 replace some connection-related states in packet header to match QPs in R_1 , R_2 , and R_3 .

After receiving data packets, receivers, R_1 , R_2 , and R_3 , generate normal ACK/NACK/CNP packets following the existing RC logic. Then the fabric aggregates ACK, filters NACK/CNP,

³For ease of understanding, readers can think master node as the multicast source. Actually, the master node can be any node in the multicast group.

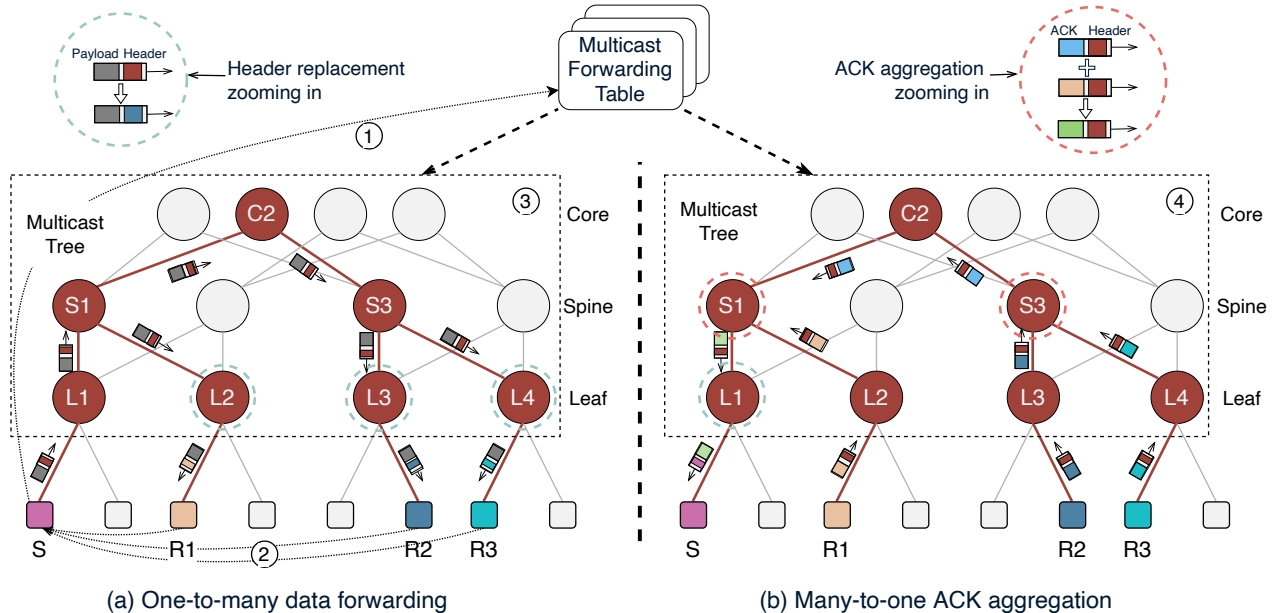


Figure 4: Gleam Architecture. We take a three-layer fat-tree topology as an example. The sender S , the three receivers, R_1 , R_2 , and R_3 , and the colored switches form a multicast group. The left and right sub-figures illustrate the one-to-many data forwarding and the many-to-one feedback aggregation (ACK as the example in the figure), respectively.

and forwards these feedbacks to the sender (④) (§3.4 & §3.5). We take ACK as an example. As shown in Fig. 4, L_2 , L_3 , L_4 , and C_2 only forward ACK packets to next-hop switches, as there is only one ACK stream as input. S_3 and S_1 perform ACK aggregation, as there are multiple ACK streams as input. L_1 changes the connection-related states in the ACK header to match the S 's QP before forwarding the aggregated ACK.

The aggregated ACK and NACK packets enable the sender to transmit the subsequent new data packets or retransmit the lost ones. The filtered CNPs are used to adjust the sender's sending rate. Every data packet will go through the above four steps until the multicast communication job is finished.

3.3 One-to-many Data Forwarding

For the data-plane one-to-many data forwarding, Gleam firstly reserves the optimal multicast forwarding, adopted by the previous in-fabric multicast solutions [20,25,44]. Thus the sender only needs to send one copy of data, and the fabric makes multiple copies and forwards them to multiple receivers via the optimal paths. Therefore, there is no bandwidth wastage, and the communication distance and latency are minimized.

Secondly, departing from the previous works that only support layer-3 routing, Gleam integrates layer-4 states into the fabric and further achieves a virtual one-to-many connection. Because of this, the single connected QP in the sender can simultaneously communicate with multiple QPs on multiple receivers. Therefore, Gleam can fully leverage the advanced features of connected transport service, *i.e.*, the one-sided WRITE operation and extended message size, as discussed in

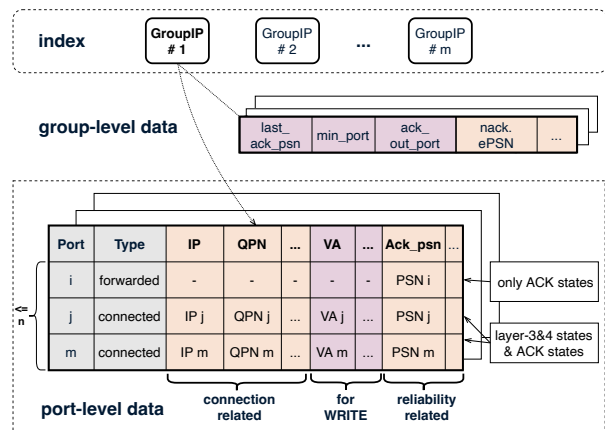


Figure 5: The extended multicast forwarding table in Gleam.

§2.1. Besides the more efficient bandwidth utilization, Gleam outperforms the application-layer multicast with shorter communication distance and lower forwarding latency, as there is no additional WQE and CQE processing, and the intermediate nodes are not involved in forwarding data.

Extended multicast forwarding table. Gleam elaborately extends the traditional multicast forwarding table structure by integrating layer-4 states. The extended table, illustrated in Fig. 5, is the foundation of Gleam. The indexed key of the table is the *multicast group IP address* (abbreviated as GroupIP). Many multicast groups can exist simultaneously, each with a unique GroupIP.

The GroupIP indexes two types of states: the group-level states and the port-level states. The group-level states contain statistics of the multicast group, which are used for the

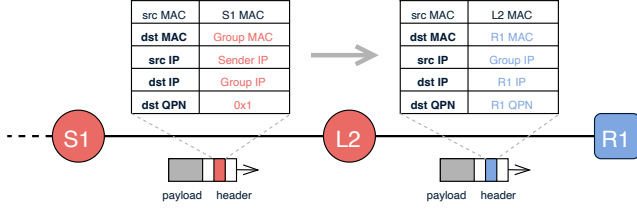


Figure 6: The packet header change in S_1 - L_2 - R_1 path in Fig. 4. The $dest_IP$ and $dest_QPN$ are changed from GroupIP and $0x1$ to R_1 's IP and QPN. Gleam changes the source IP from the sender's IP to GroupIP and manually replaces the destination MAC address.

many-to-one feedback aggregation. The port-level states are formatted into an array with at most n (# of switch ports) entries, only containing ports included in the multicast tree. The entry with $port$ as i represents states related to $port_i$. Each entry is assigned one of two types: *connected* and *forwarded*, where *connected* means that this port is directly connected to a receiver node, and *forwarded* means that the next hop is a switch. The *connected* entry contains the connected receiver's layer-3 and layer-4 states, as well as the ACK/NACK states, and the *forwarded* entry only contains ACK/NACK states. §3.4 introduces the usage of these ACK/NACK states.

The specific memory space used by one multicast table depends on the number of ports involved in the multicast group, which is at most n . We calculate that 1K multicast groups at most cost 0.92MB memory when each group contains the maximum entries. Moreover, the design goal of Gleam is not to compress the switch-maintained states but to provide a general multicast protocol with prominent RDMA features. We can use many approaches to extend Gleam to support more groups, and we'll talk about them in §6.

Establishing QPs. Each multicast member follows the common unicast-like steps to establish the RC QP but assigns a virtual destination to it. Specifically, the destination IP is set as a unique GroupIP, and the destination QPN can be assigned as any non-conflicting value (*e.g.*, $0x1$). Commodity RNICs provide the application with the programming interface to specify the destination IP and QPN without modifying the RNIC circuit [6]. After QPs establishment, multicast members exchange their QPs information and register the above-described forwarding table to switches, as described in Appendix A. Once the registration finishes, the sender can start sending multicast data packets.

One-to-many data forwarding. Switches involved in the multicast tree are responsible for forwarding data via the multicast tree and modifying packet headers to match different QPs. The switch follows Algorithm 1 to process data packets. Upon receiving a data packet (p), the switch uses the destination IP ($p.dest_IP$) in the packet header to index the associated multicast forwarding table (T). Then switch iterates all entries (one entry corresponds to one port) in T and actions as follows: (i) if the type is *forwarded*: creates a packet copy

Algorithm 1 Forwarding Packets and Replacing Headers.

```

1:  $p \leftarrow$  data packet
2:  $j \leftarrow$  port that  $p$  enters
3:  $T \leftarrow$  multicast forwarding table indexed by  $p.dest\_IP$ 
4: for  $Entry \in T$  do  $\triangleright$  loop over  $T$  and copy/forward  $p$ 
5:   if  $Entry.type = forwarded \ \& \ Entry.port \neq j$  then
6:      $\bar{p} \leftarrow$  a copy of  $p$ 
7:     send  $\bar{p}$  out from  $Entry.port$ 
8:   if  $Entry.type = connected \ \& \ Entry.port \neq j$  then
9:      $\bar{p} \leftarrow$  a copy of  $p$ 
10:     $\bar{p}.dest\_IP(QPN) = Entry.dest\_IP(QPN)$ 
11:     $\bar{p}(\dots) = Entry(\dots)$ 
12:    send  $\bar{p}$  out from  $Entry.port$ 

```

and forwards it through this port; (ii) if the type is *connected*: creates a packet copy, modifies its connection-related states, and forwards it through this port.

We take one path, S_1 to L_2 to R_1 , of the multicast tree in Fig. 4 as an example to show the header change, which is illustrated in Fig. 6. Firstly, the destination IP and QPN are modified to match R_1 's QP identification, as described in §3.1. Besides, Gleam changes the source IP from the sender's IP to GroupIP. As a result, when R_1 generates feedback, the feedback's destination IP will be the data packet's source IP, *i.e.*, GroupIP. Thus feedback packets can also index the associated forwarding table by their destination IP. Besides, Gleam switches replace the destination MAC address to avoid the receiver's MAC layer discarding the packet.

Support for one-to-many WRITE. Gleam maintains connectivity between the sender and multiple receivers, which is sufficient for SEND/RECEIVE. However, WRITE requires more support. WRITE allows a node to write a memory slot on a remote node. The to-be-written MR info (including the remote VA and R_key) is indicated in the first packet of the WRITE request. The WRITE responder's RNIC will check the MR info and execute the request only when they are correct. Otherwise, the packets will be discarded. To enable one-to-many WRITE, Gleam needs to modify the MR states in the WRITE request header for different receivers.

Besides maintaining MR info for different receivers, the MR info needs to be updated for every WRITE request because the MR changes with different WRITE requests. We force the host application to invoke an extra WRITE message which contains the MR states of different receivers, before submitting the actual WRITE request. Then the leaf switch recognizes this special message, updates MR info to the table, and replaces the MR states for the subsequent real WRITE request. This per-request updating scheme introduces minimal extra bandwidth overhead as long as the extra WRITE message is small compared to the total volume of transmitted data. We evaluate the performance of one-to-many WRITE in §5.2.2. Moreover, we discuss a possible way to avoid this

Algorithm 2 Many-to-one ACK/NACK Aggregation

```
1:  $p \leftarrow$  ACK/NACK packet
2:  $T \leftarrow$  forwarding table indexed by  $p.dest\_IP$ 
3:  $Entry \leftarrow$  entry in  $T$  with  $port\ i$  that  $p$  enters
4:  $min\_port \leftarrow$  port with minimum  $ack\_psn$  last time
5:  $last\_ack\_psn \leftarrow$  last aggregated ACK's PSN
6: if  $p$  is ACK then
7:   if  $p.psn \geq Entry.ack\_psn$  then  $\triangleright$  update  $Entry$ 
8:      $Entry.ack\_psn = p.psn$ 
9:      $Entry(...) = p(...)$ 
10:  if  $i = min\_port \ \& \ p.psn \geq last\_ack\_psn$  then
11:    GENERATENEWACK/NACK()
12: else  $\triangleright p$  is a NACK packet
13:   if  $p.psn - 1 \geq Entry.ack\_psn$  then
14:      $Entry.ack\_psn = p.psn - 1$   $\triangleright$  update  $Entry$ 
15:   if  $p.psn \leq T.nack.ePSN$  then  $\triangleright$  update  $T.nack$ 
16:      $T.nack.ePSN = p.psn$ 
17:      $T.nack(...) = p(...)$ 
18:   GENERATENEWACK/NACK()
```

extra overhead in Appendix C. This alternative approach requires the RNIC's modification.

3.4 Many-to-one Feedback Aggregation

With the extended one-to-many data forwarding, Gleam can optimally forward data replicas to multiple receivers, and different receivers' QPs can accept the packet. However, the second incompatibility impeding the leverage of RC is the existing reliability logic [32, 47]. The current reliability logic in commodity RNICs is designed to interpret a single feedback stream from a single receiver; thus, multiple feedback streams from multiple receivers can compromise reliability.

Feedback contains various types of packets, such as ACK, NACK, notification packets for congestion control (CC) (such as CNP [47]), *etc.* We talk about ACK and NACK here, and the processing for CC-related feedback is described in §3.5. Gleam performs the in-fabric many-to-one ACK aggregation and NACK filtering to deliver a unicast-like ACK/NACK stream to the sender. As a result, the sender can correctly interpret ACK and proceeds with data transmission. Besides, when the loss occurs, the sender can precisely detect and retransmit the lost packet.

The basic principles of ACK-aggregation/NACK-filtering are that (i) the multicast source should receive an ACK only when *all* receivers have received the corresponding packets; (ii) the multicast source should receive a NACK when *any* receiver loses a packet. Moreover, the aggregation needs to consider the processing rules of RDMA protocol, such as the go-back-N retransmission and ACK coalescing.

Gleam maintains ACK/NACK-related information in the extended multicast forwarding table, illustrated in Fig. 5.

Algorithm 3 ACK/NACK Generation

```
1: function GENERATENEWACK/NACK()
2:    $T \leftarrow$  forwarding table indexed by  $p.dest\_IP$ 
3:    $ack\_out\_port \leftarrow$  port that data packets enter
4:    $min\_psn \leftarrow \infty$ 
5:    $min\_port \leftarrow -1$ 
6:   for  $Entry \in T$  do  $\triangleright$  find the minimized  $ack\_psn$ 
7:     if  $min\_psn < Entry.ack\_psn$  then
8:        $min\_psn = Entry.ack\_psn$ 
9:        $min\_port = i$ 
10:  // generate aggregated ACK
11:  create ACK packet  $p$  with ( $psn = min\_psn$ )
12:  send  $p$  through  $ack\_out\_port$ 
13:  // check for generating NACK
14:  if  $min\_psn + 1 = T.nack.ePSN$  then
15:    create NACK packet  $p$ 
16:     $p.psn = T.nack.ePSN$ 
17:    send  $p$  through  $ack\_out\_port$ 
18:  update global  $last\_ack\_psn$  with  $min\_psn$ 
```

Upon receiving an ACK/NACK packet, switches find the associated forwarding table by ACK/NACK packets' destination IP, *i.e.*, GroupIP. We first present a working logic without packet loss and then describe how to handle NACK packets.

Handle ACK. The ACK-related states includes (i) the group-level data, including the PSN of the last aggregated ACK ($last_ack_psn$), the port from which the ACK should be sent (ack_out_port), *etc.*; (ii) the port-level data, including the largest acked PSN (ack_psn) of each port. Switches process ACK packets, update the related states in the forwarding table, and generate the aggregated ACK packet, following Algorithm 2.

Upon receiving an ACK packet, the switch firstly updates this port's ack_psn if the PSN of incoming ACK is larger than the old ack_psn . Then, if the trigger condition (Line 10) is satisfied, the GENERATENEWACK/NACK function in Algorithm 3 is called to generate an aggregated ACK. The aggregated ACK contains the minimum ack_psn recorded by the switch, which is found by iterating all multicast forwarding table entries. As a result, each aggregated ACK forwarded by the switch confirms that all downstream receivers have received the corresponding data packets.

The port that owns the minimum ack_psn is recorded as min_port . Each time an ACK with a larger PSN is received from min_port (Line 10), the ACK generation is triggered. Thus not every ACK packet will trigger the generation, and the number of ACKs received by the source is reduced.

Handle NACK. When the loss occurs, the receiver generates NACK packet to notify the multicast sender. The NACK packet (p) contains the receiver's expected PSN ($p.psn$). Each NACK will acknowledge all data packets with PSN smaller than the expected PSN. This rule must be carefully handled

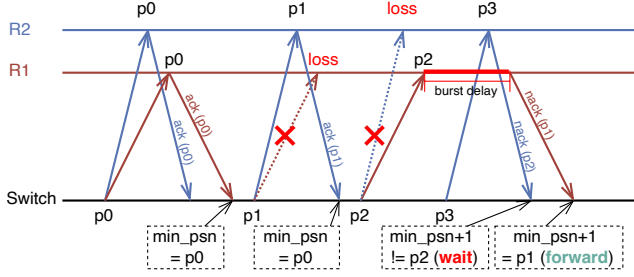


Figure 7: An example of NACK filtering.

in NACK generation.

We illustrate an example in Fig. 7. There are two receivers and one switch. The switch generates two copies of data and forwards them to two receivers. There are two lost packet, $p1_{R1}$ and $p2_{R2}$, and two associated NACK packets, $nack_{p1}$ and $nack_{p2}$. The switch should forward the $nack_{p1}$ because it contains the minimum expected PSN. If the $nack_{p2}$ is forwarded to the sender first, the loss of $p1_{R1}$ will be covered because the sender will assume that all the packets before the expected PSN of $nack_{p2}$ (*i.e.*, $p2$) have been received. Thus the sender won't retransmit $p1$ anymore, and the reliability is compromised.

Therefore, the NACK packet should be forwarded only when all receivers have acknowledged all packets with PSN smaller than its expected PSN. We implement this judgement in Line 14 of Algorithm 3. If the condition is not satisfied, the switch keeps waiting, during which new ACK/NACK packets keep coming. If the switch receives a new NACK and its PSN is not great than the recorded $T.nack.ePSN$, the $T.nack.ePSN$ is updated, and the NACK generation condition is rechecked, as shown in Line 15 of Algorithm 2.

3.5 Other Considerations

From single source to multiple sources. Multicast source switching is common in datacenter communications. For example, in the *PB* phase of HPL, different nodes play as the multicast source in different epochs. Gleam can support this communication requirement by switching multicast source inside the group without reestablishing QPs. In particular, when the last multicast source finishes transmission, the multicast members use the already established QPs for the subsequent multicast communication. The multicast forwarding table maintained by the switches stays almost untouched. In Appendix B, we provide a detailed description of multicast source switching.

Congestion control. As described above, we reuse the entire RC transport at the end-host for RNIC compatibility. This means that we also reuse the built-in congestion control (CC) mechanism to regulate the multicast sending rate. Rationally, a multicast congestion control algorithm should match its sending rate with the most congested path (*i.e.*, bottleneck)

of its data distribution tree [43, 46].

To this end, we enhance the switch feedback aggregation with a congestion signal filtering mechanism while remaining the end-host CC unchanged. Specifically, we maintain a congestion counter for each link at the switch, recording the congestion signal from the receivers or downstream switches. We then perform signal filtering to only pass through the congestion signal from the most congested link⁴ (forming a path when cascading each switch link end-to-end). Further, a periodic aging mechanism is added to update the congestion counter to match the frequently changing network dynamics.

4 Implementation

Gleam's implementation consists of (i) a fully-functional Gleam switch prototype which implements the overall in-fabric logic; and (ii) a set of software APIs exposed to applications. Our prototype is built upon a FPGA-assisted commodity switch.

FPGA-based prototype. We implement the group registration, data packet duplication, header modification, and feedback aggregation logics on an FPGA board. The board is equipped with a commodity FPGA chip [15] and four 100Gbps Ethernet interfaces. The FPGA resource utilization is shown in Table 2. We build our testbed with the FPGA board, a commodity Ethernet switch, and four servers, as illustrated in Fig. 8. Each server is equipped with a commodity RNIC. The FPGA board and four RNICs are connected to the commodity switch through 100Gbps Ethernet interfaces.

The commodity switch is configured by Access Control List (ACL) to route the servers' multicast traffic to the FPGA board. The FPGA board identifies the multicast data (ACK⁵) packets through the specific packet header by *Parser* and *Arbiter*. The data (ACK) packets will be duplicated (aggregated) by *Duplicator* (ACK Aggregator). The resulting packets will be pushed in *Queue System*, waiting for the *Multiplexer* to schedule in case for queue competition. Finally, the duplicated data (aggregated ACK) packets are sent back to the commodity switch. During processing, the *Multicast Forwarding Table* is accessed when needed.

Software APIs. We provide various communication libraries and middleboxes for Gleam multicast support. Take the commonly-used OpenMPI as an example, we modify the OpenMPI (v4.1.1) [12] and UCX (v2.3) [14] to adapt to Gleam's design, as shown in Fig. 8. Specifically, we add a new implementation of *MPI_Bcast* and modify UCX for multicast QPs creation and data transmission. When the new *MPI_Bcast* is called, the MPI process calls the UCX to establish QPs for multicast. Multicast members exchange their

⁴There are different implementations of congestion signals. Some adopt stand-alone CNP, while others reuse the ACK to carry the congestion bits. Our congestion signal filtering design works for them all.

⁵In Fig. 8, we use ACK to represent all types of feedback.

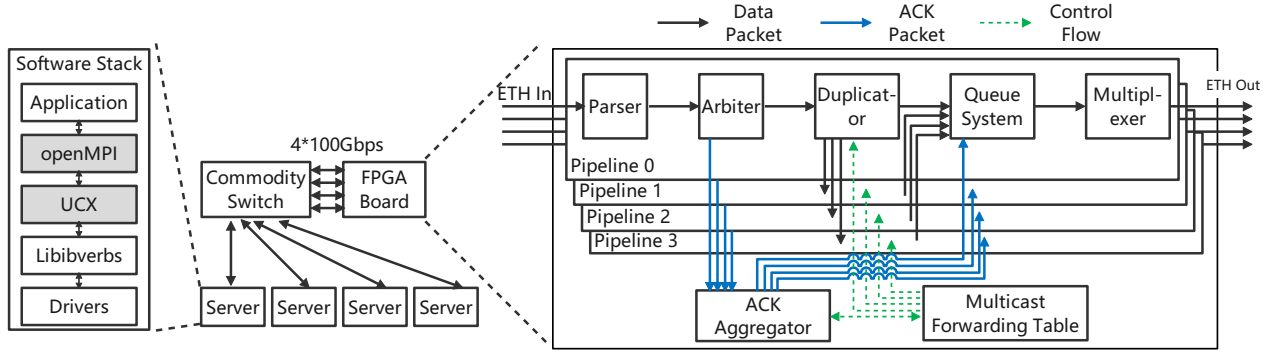


Figure 8: Gleam testbed consists of a commodity switch, an FPGA-based prototype, and four servers. The FPGA board implements four processing pipelines, each capable of line-rate traffic processing for a 100Gbps Ethernet interface.

Resource	LUT	Register	BRAM
Usage	53169	15391	188

Table 2: Resource usage of the Gleam in-fabric logic.

QPs information, and the handshake starts, as described in Appendix A. Once the multicast group is successfully established, the UCX finally calls the RDMA primitives defined in the well-known *libibverbs* [5] to transmit data. The software modifications at the end-host are transparent to the upper-layer applications and don’t require any RNIC or RDMA driver modification.

5 Evaluation

We evaluate Gleam’s performance through extensive testbed and simulation experiments. In testbed experiments, we first examine Gleam using the micro-benchmark (§5.1); then we deploy two realistic applications (§5.2) upon Gleam and show their performance improvement. The testbed’s topology and configuration are described in §4. In simulations (§5.3), we evaluate Gleam in an extended large-scale topology and explicitly estimate the goodput of Gleam over different packet loss rate. Our experiment results reveal that:

- Gleam accelerates *MPI_Bcast* by up to $2.2\times$ compared with OpenMPI.
- Gleam improves the performances of the realistic applications. Specifically, Gleam speeds up the HPL communication by up to $2.9\times$, improves the IOPS throughput of storage data replication by $2.7\times$, and reduces the single IO latency by up to 65%.
- Gleam achieves consistently high performance in large-scale topologies. Its communication speedup scales well with the multicast group size. Its performance gain is also resilient to packet losses.

5.1 Micro-benchmark

We first evaluate *MPI_Bcast*, one of the basic MPI primitives, through the modified OpenMPI [12] and UCX [14]. We measure the job completion time (JCT) as the main metric. In our testbed, we select one server as *MPI_Bcast* source and three servers as *MPI_Bcast* receivers, forming a small one-to-three multicast group. The original OpenMPI performance is then compared with that of Gleam.

We measure the JCT under various multicast message sizes, and calculate Gleam’s acceleration ratio, shown in Fig. 9. Gleam achieves lower JCTs across various message sizes compared with OpenMPI. With larger message size, Gleam achieves lower JCT. For instance, Gleam achieves $12\mu s$ reduction ($1.6\times$ acceleration ratio) with 64KB message and $124ms$ reduction ($2\times$ acceleration ratio) with 1GB message. For message size larger than 128KB, Gleam stably achieve about 50% less JCT.

The performance improvement of Gleam stems from its optimal bandwidth utilization. Gleam delivers data efficiently without bandwidth wastage. In contrast, OpenMPI inevitably wastes bandwidth because it transmits identical data multiple times. Consequently, with larger message size, Gleam saves more bandwidth and thus offers lower JCT. Through these micro-benchmark experiments, we validate the correctness of our Gleam prototype and demonstrate that Gleam can speed up multicast communication under simple settings.

5.2 Realistic Applications

We deploy two realistic applications with multicast patterns (§2.2): HPL (§5.2.1) and storage data replication (§5.2.2); and evaluate their performances with and without Gleam.

5.2.1 High-performance Linpack (HPL)

We integrate Gleam into HPL [3], and build a HPL cluster using our prototype testbed to evaluate the performance of HPL. Each HPL epoch contains three steps: *Panel Factorization*,

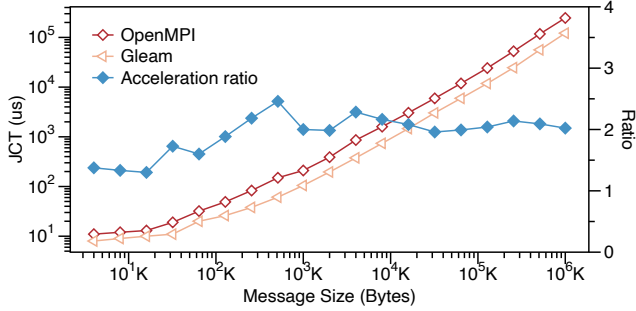


Figure 9: The JCTs of *MPI_Bcast*. The left vertical axis indicates the multicast JCTs. The right vertical axis shows the acceleration ratios of Glean over OpenMPI.

Panel Broadcast (PB) and *Update*. The *PB* is a standard multicast transmission, and the *Update* includes a communication phase, called *Row Swap (RS)*, which can be implemented with multicast. The volume of multicast traffic is around several GBytes at the first epoch and linearly decrease to nearly zero as the computing proceeds.

We test HPL’s *PB* and *RS* stages separately. We measure the total JCT (including computation and communication) of HPL when speeding up *PB* and *RS* solely, and only the communication time of *PB* and *RS*. Moreover, the *RS* communication volume depends on the real-time computing result; thus we evaluate *RS* with two data distributions, which are uniform and centralized. The original HPL implementation is selected as alternative solution compared with Glean, where the *PB* and *RS* are recommended to use the *increasing-ring* (abbreviated as *ring* in following HPL evaluations) and *long* algorithms, respectively [3]. Both algorithms are overlay multicast solutions implemented by multiple underlay RC unicasts.

As results illustrated in Fig. 10 and Fig. 11, Glean reduces JCT in all settings. Specifically, with communication time only, Glean reduces the JCT of *PB*, *RS (uniform)*, and *RS (centralized)* by 67%, 18%, and 46%, respectively. With both computation and communication time included, Glean reduces the JCT of *PB*, *RS (uniform)*, and *RS (centralized)* by up to 12%, 4.67%, and 9.55%, respectively.

Tolerance of Data Distribution Note that Glean achieves better improvement in non-random data distribution (46%) compared with uniform data distribution (18%). This is expected as the performance of Glean doesn’t depend on data distribution. However, the *long* algorithm is sensitive to data distribution. The *long* algorithm achieves best performance when data is uniformly distributed, but degrades if the data is non-random distributed or even centralized, as more communication is needed to make the data evenly distributed before data exchanging between neighbors.

5.2.2 Storage Data Replication

We also integrate Glean into a proprietary commodity distributed storage system. We utilize our testbed prototype to evaluate the performance of storage data replication. We select one node as client and three nodes as servers, thus forming a 3-copies writing setting. We compare the performance of Glean with 3-unicasts, and the one-copy writing is also measured as a baseline. The WRITE operation is used in all solutions. In 3-unicasts, client maintains three RC connections with three servers. In Glean, client only maintains one RC connection. We measure the writing throughput using IOPS (IO per second) as the main metric, and the single IO latency.

Throughput. We set IO size as 8KB and let the client keep writing data to three servers. We measure the average IOPS achieved by the client. As Fig. 12 shows, Glean can achieve nearly optimal writing IOPS, about 1.167M IOPS, which is comparable with the ideal one-copy writing’s 1.188M IOPS. On the contrary, the 3-unicasts only achieves 0.413M IOPS, only about 35% of Glean. With Glean, the application goodput can achieve 76.5Gbps (1.1M × 8KB), while 3-unicasts only can reach 26.24Gbps. This is because the client only need to send one copy of data for 3-replications writing with Glean, effectively mitigating the bandwidth bottleneck at the client’s output link.

IO Latency. In addition, we measure the single IO latency over different IO size, shown in Fig. 13. The single IO latency is defined as the period between the client submits the WRITE request and receives the CQE from the RNIC. The result shows that Glean achieves significantly lower latency than 3-unicasts, and delivers a comparable latency with the ideal one-copy. For instance, Glean reduces IO latency by about 40% and 60% in 64KB and 512KB, compared with 3-unicasts. Moreover, Glean accomplishes lower latency in larger IO size. As result shown, the gap between Glean and 3-unicasts is enlarged as IO size increasing.

The advantage of Glean comes from the reduction of end-host storage stacks involvement and the total transmitted traffic volume. 3-unicasts transmits the identical data and experiences the same storage stacks three times. As a result, the IO latency is increased. On the other hand, Glean enables the client to run the storage stack and transmit the data only once, which effectively reduces the IO latency.

5.3 Simulations

We provide complementary experiments using ns-3 [9]. We first build a large-scale topology and evaluate Glean over different multicast scales. We then simulate a lossy environment and measure Glean over different packet loss rates.

Simulation setting. We simulate a large-scale 3-layer fat-tree topology with 16384 servers and a 1:1 oversubscription ratio.

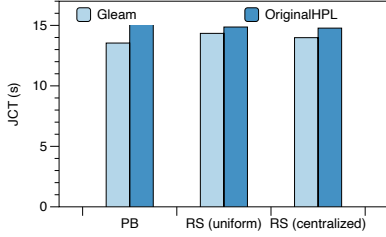


Figure 10: HPL JCTs (computation & communication).

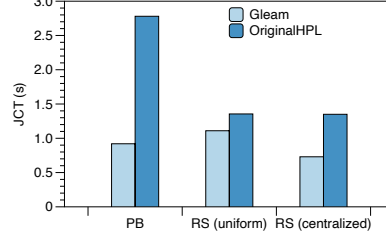


Figure 11: HPL JCTs (communication only).

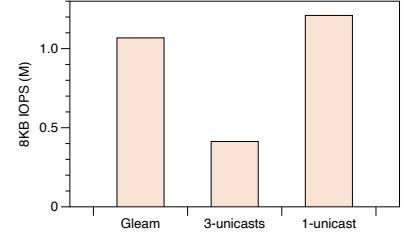


Figure 12: Writing throughput in IOPS.

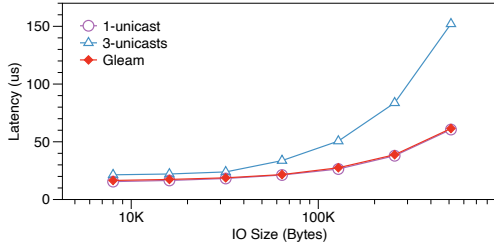


Figure 13: Single IO latency.

Each server is equipped with one 200Gbps RNIC, and each switch has 64 200Gbps ports.

Workload and Metrics. We again use the HPL pattern as the simulation workload. We provide various workload scales, each labeled as $N * N$, meaning that $N * N$ nodes form a logical $N * N$ matrix to run HPL. In each workload, each row node would perform one *PB*, and each column node would perform one *RS*. With Gleam, the *PB* procedure involves N multicast groups transmitting simultaneously, each consisting of N group members. The *RS* involves another different N multicast groups. The *ring* algorithm for *PB* and *long* algorithm for *RS* are simulated as comparisons with Gleam. We use the total JCT, *i.e.*, the total communication time of *PB* and *RS*, as the experiment metric.

The simulations are time-costly because of its large topology and the more than four million flows. To accelerate the simulation, we integrate MPI framework into the ns-3 simulator to enable the parallel simulation with multiple threads. The simulation run time is reduced by 70% with 12 threads.

Large-scale multicast. We measure the average JCT of HPL workload over different multicast scales. In particular, we select five scales: $8 * 8$, $16 * 16$, $32 * 32$, $64 * 64$, $128 * 128$. For each scale, we run multiple times and calculate the average JCT. The original HPL implementation (*ring* for *PB* and *long* for *RS*) is compared with Gleam. Results in Fig. 14 show that Gleam achieves lower JCT in all multicast scales with a reduction from 62% to 73%. For instance, Gleam reduces the JCT from 13.7ms to 5.17ms (62% reduction) over $8 * 8$ scale, and from 19.4ms to 5.17ms (73% reduction) over $128 * 128$ scale. The improvement increases with the growing multicast scale, consistent with the results in §5.1.

Note that Gleam’s JCT doesn’t scale up with increasing multicast scale. The reason is that, in Gleam, the larger multicast scale only brings more parallel-transmitted multicast groups, while the traffic volume transmitted by each group stays the same. Because of Gleam’s group-level load-balancing (Appendix A), more multicast groups don’t incur much congestion deterioration. In contrast, with the *ring* and *long* algorithms, the number of parallel unicast flows expands linearly with the growing multicast scale, which causes congestion and results in JCT degradation.

Loss tolerance. We compare the JCT of HPL workload under different packet loss rates, from 10^{-8} to 10^{-3} , which are emulated via randomly discarding packets in the middle switches. We choose two group sizes (64 and 512) to evaluate. As shown in Fig. 15, Gleam presents good loss tolerance as it can maintain lower JCT compared with original HPL implementation (*ring* and *long*) under all loss rates. In particular, Gleam reduces the JCT by 46%-52% with size 64 and 86%-94% with size 512, respectively.

To better show Gleam’s reaction to loss, we calculate the goodput under different packet loss rates, *i.e.*, the normalized throughput compared with setting without loss, shown in Fig. 16. Gleam’s goodput decreases largely than *ring* and *long* because the multicast sender is responsible for multiple receivers, *i.e.*, when any receiver loses a packet, the sender must retransmit it. In contrast, the sender in *ring* and *long* is only responsible for one receiver.

Even though Gleam is more sensitive to packet loss, Gleam imposes at most 10% goodput degradation with a loss rate not larger than 0.01%, which is a common packet loss rate in data center [47]. Even under an excessive 0.1% loss rate, Gleam can maintain 42% goodput and perform better (gains a 7× lower JCT) than the original HPL implementation.

6 Related Works

Internet and Datacenter multicast. Multicast has been widely applied in large-scale Internet applications, such as Internet broadcast [7], video conferencing [16], and multiplayer games [18], *etc.* Prior works for the Internet [17, 23, 24, 33, 42] mostly focus on the multicast routing, *i.e.*, to find promising

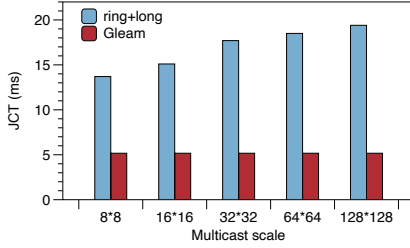


Figure 14: JCTs with different multicast scales.

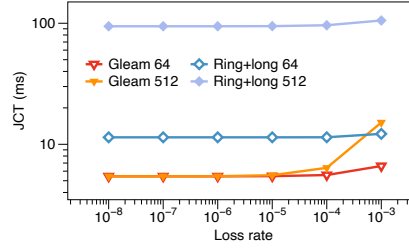


Figure 15: JCTs under various packet loss rates.

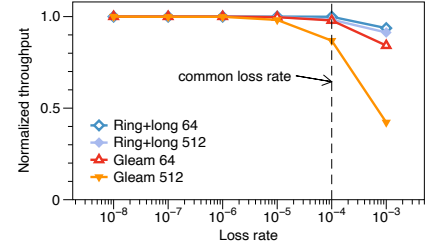


Figure 16: Normalized throughput (goodput) under various packet loss rates.

multicast paths, inside ISPs. For instance, Yeti [23] supports multicast routing with traffic engineering and service chaining requirements for large-scale ISPs. Yeti creates labels representing forwarding information for multicast graphs and processes these labels to forward packets to targeted paths. Although there are a bunch of prior works on the Internet, most of them merely provide best-effort delivery, which only works for applications without reliability requirement.

There are some works [43, 46] aim to provide reliability for datacenter applications upon approaches with best-effort delivery. However, existing reliable multicast solutions mainly adopt a TCP-like software stack and cannot meet the demand for high-speed communication in datacenters. In contrast, Gleam leverages the advanced RDMA stacks to process multicast traffic, providing high-speed reliable communication.

Multicast scalability. Datacenter applications impose a demand for high scalability. As the traditional IP multicast [20], along with its native group management, IGMP and tree construction protocol, PIM [28], are poor in scalability, many works [25, 37, 44] attempt to address the scalability issue, *i.e.*, supporting as much as possible multicast groups. For example, Elmo [44] encodes the routing link of a multicast tree into rules formatted as packet header. Thus Elmo switch only needs to maintain rule parsing logic, reducing the total switch-maintained states. Orca [25] utilizes the large memory space of the server, making servers assist in forwarding packets, reducing the switch’s burden on maintaining states.

These works that address the scalability issue are orthogonal with the Gleam design. Our goal in this work is to provide a general multicast protocol with prominent RDMA features and reliability guarantee rather than compressing the switch-maintained states. As mentioned before, Gleam can support at least 1K multicast groups using 0.92MB space, which is acceptable for a majority of multicast applications in datacenters. Gleam can support even more multicast groups when getting extended further upon these works.

7 Conclusion

We present Gleam, an RDMA-accelerated multicast protocol that significantly facilitates multicast communication while maintaining compatibility with commodity RNICs. Gleam

reuses existing RDMA RC transport to process multicast traffic, thus preserving the optimal bandwidth utilization and benefiting from superior RDMA functionality. Gleam replaces packet headers and aggregates feedback in the fabric to remain compatible between existing RC logic and multicast traffic. We provide a fully functional Gleam prototype, which requires no RNIC modification. Extensive testbed experiments and simulations demonstrate Gleam’s superior performance in multicast acceleration.

Gleam opens the door for efficiently leveraging the prominent RDMA stacks with in-fabric assistance to accelerate group communication patterns. While this work mainly focuses on multicast; for future works, we plan to extend Gleam for more group communication patterns, such as many-to-one (*e.g.*, *MPI_Reduce*) and many-to-many (*e.g.*, *MPI_Alltoall*), *etc.*

References

- [1] Infiniband architecture volume 1, general specifications, release 1.2.1. <https://www.infinibandta.org/specs>, 2008.
- [2] Supplement to infiniband architecture specification volume 1 release 1.2.2 annex a17: Rocev2 (ip routable roce). <https://www.infinibandta.org/specs>, 2014.
- [3] Hpl - a portable implementation of the high-performance linpack benchmark for distributed-memory computers. <https://netlib.org/benchmark/hpl/>, 2018.
- [4] Intel machine learning scalability library (mlsl). <https://github.com/intel/MLSL>, 2019.
- [5] libibverbs. <https://github.com/linux-rdma/libibverbs/blob/master/Documentation/libibverbs.md>, 2021.
- [6] Linux manual page. https://man7.org/linux/man-pages/man3/ibv_modify_qp.3.html, 2021.
- [7] Bt iptv. <https://bit.ly/3ssCvTz>, 2022.

- [8] Connectx-5. <https://www.nvidia.com/en-us/networking/ethernet/connectx-5/>, 2022.
- [9] ns-3, a discrete-event network simulator for internet systems. <https://www.nsnam.org/>, 2022.
- [10] Nvidia collective communication library (nccl). <https://developer.nvidia.com/nccl>, 2022.
- [11] oneapi collective communications library (oneccl). <https://github.com/oneapi-src/oneCCL>, 2022.
- [12] Openmpi: Open source high performance computing. <https://www.open-mpi.org/>, 2022.
- [13] Top 500 list. <https://www.top500.org/lists/top500/2022/06/>, 2022.
- [14] Uniflex communication x. <https://openucx.org/>, 2022.
- [15] Virtex ultrascale. <https://www.xilinx.com/products/silicon-devices/fpga/virtex-ultrascale.html/>, 2022.
- [16] Xiangwen Chen, Minghua Chen, Baochun Li, Yao Zhao, Yunnan Wu, and Jin Li. Celerity: A low-delay multi-party conferencing solution. In *Proceedings of the 19th ACM international conference on Multimedia*, pages 493–502, 2011.
- [17] Sheng-Hao Chiang, Jian-Jhih Kuo, Shan-Hsiang Shen, De-Nian Yang, and Wen-Tsuen Chen. Online multicast traffic engineering for software-defined networks. In *IEEE INFOCOM 2018-IEEE Conference on Computer Communications*, pages 414–422. IEEE, 2018.
- [18] Tae Won Cho, Michael Rabinovich, KK Ramakrishnan, Divesh Srivastava, and Yin Zhang. Enabling content dissemination using efficient and scalable multicast. In *IEEE INFOCOM 2009*, pages 1980–1988. IEEE, 2009.
- [19] Sudheer Chunduri, Scott Parker, Pavan Balaji, Kevin Harms, and Kalyan Kumaran. Characterization of mpi usage on a production supercomputer. In *SC18: International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 386–400. IEEE, 2018.
- [20] Jon Crowcroft and Karen Paliwoda. A multicast transport protocol. In *Symposium proceedings on Communications architectures and protocols*, pages 247–256, 1988.
- [21] D Crupnico, M Kagan, A Shahar, N Bloch, and H Chapman. Dynamically-connected transport service, may 19 2011. URL <https://www.google.com/patents/US20110116512>. *US Patent App*, 12(621,523).
- [22] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: Simplified data processing on large clusters. 2004.
- [23] Khaled Diab and Mohamed Hefeeda. Yeti: Stateless and generalized multicast forwarding. In *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*, pages 1093–1114, 2022.
- [24] Khaled Diab, Carlos Lee, and Mohamed Hefeeda. Oktopus: Service chaining for multicast traffic. In *2020 IEEE 28th International Conference on Network Protocols (ICNP)*, pages 1–11. IEEE, 2020.
- [25] Khaled Diab, Parham Yassini, and Mohamed Hefeeda. Orca: Server-assisted multicast for datacenter networks. In *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*, pages 1075–1091, 2022.
- [26] Jack J Dongarra, Piotr Luszczek, and Antoine Petit. The linpack benchmark: past, present and future. *Concurrency and Computation: practice and experience*, 15(9):803–820, 2003.
- [27] Aleksandar Dragojević, Dushyanth Narayanan, Miguel Castro, and Orion Hodson. {FaRM}: Fast remote memory. In *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14)*, pages 401–414, 2014.
- [28] Deborah Estrin, Dino Farinacci, Ahmed Helmy, David Thaler, Stephen Deering, Mark Handley, Van Jacobson, Ching-Gung Liu, Puneet Sharma, and Liming Wei. Protocol independent multicast-sparse mode (pim-sm): Protocol specification. Technical report, 1998.
- [29] William Fenner. Internet group management protocol, version 2. Technical report, 1997.
- [30] Yixiao Gao, Qiang Li, Lingbo Tang, Yongqing Xi, Pengcheng Zhang, Wenwen Peng, Bo Li, Yaohui Wu, Shaozong Liu, Lei Yan, et al. When cloud storage meets {RDMA}. In *18th USENIX Symposium on Networked Systems Design and Implementation (NSDI 21)*, pages 519–533, 2021.
- [31] Andrew Gibiansky. Bringing hpc techniques to deep learning. *Baidu Research, Tech. Rep.*, 2017.
- [32] Chuanxiong Guo, Haitao Wu, Zhong Deng, Gaurav Soni, Jianxi Ye, Jitu Padhye, and Marina Lipshteyn. Rdma over commodity ethernet at scale. In *Proceedings of the 2016 ACM SIGCOMM Conference*, pages 202–215, 2016.
- [33] Liang-Hao Huang, Hsiang-Chun Hsu, Shan-Hsiang Shen, De-Nian Yang, and Wen-Tsuen Chen. Multicast traffic engineering for software-defined networks. In

IEEE INFOCOM 2016-The 35th Annual IEEE International Conference on Computer Communications, pages 1–9. IEEE, 2016.

- [34] Yimin Jiang, Yibo Zhu, Chang Lan, Bairen Yi, Yong Cui, and Chuanxiong Guo. A unified architecture for accelerating distributed {DNN} training in heterogeneous {GPU/CPU} clusters. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 463–479, 2020.
- [35] Anuj Kalia, Michael Kaminsky, and David G Andersen. Using rdma efficiently for key-value services. In *Proceedings of the 2014 ACM Conference on SIGCOMM*, pages 295–306, 2014.
- [36] Mu Li, David G Andersen, Alexander J Smola, and Kai Yu. Communication efficient distributed machine learning with the parameter server. *Advances in Neural Information Processing Systems*, 27, 2014.
- [37] Xiaozhou Li and Michael J Freedman. Scaling ip multicast on datacenter topologies. In *Proceedings of the ninth ACM conference on Emerging networking experiments and technologies*, pages 61–72, 2013.
- [38] Matthew L Massie, Brent N Chun, and David E Culler. The ganglia distributed monitoring system: design, implementation, and experience. *Parallel Computing*, 30(7):817–840, 2004.
- [39] Rui Miao, Lingjun Zhu, Shu Ma, Kun Qian, Shujun Zhuang, Bo Li, Shuguang Cheng, Jiaqi Gao, Yan Zhuang, Pengcheng Zhang, et al. From luna to solar: the evolutions of the compute-to-storage networks in alibaba cloud. In *Proceedings of the ACM SIGCOMM 2022 Conference*, pages 753–766, 2022.
- [40] Radhika Mittal, Vinh The Lam, Nandita Dukkupati, Emily Blem, Hassan Wassel, Monia Ghobadi, Amin Vahdat, Yaogong Wang, David Wetherall, and David Zats. Timely: Rtt-based congestion control for the datacenter. In *Proceedings of the 2015 ACM SIGCOMM Conference*, page 537–550, 2015.
- [41] Sumit Kumar Monga, Sanidhya Kashyap, and Changwoo Min. Birds of a feather flock together: Scaling rdma rpcs with flock. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, pages 212–227, 2021.
- [42] Bangbang Ren, Deke Guo, Guoming Tang, Xu Lin, and Yudong Qin. Optimal service function tree embedding for nfv enabled multicast. In *2018 IEEE 38th international conference on distributed computing systems (ICDCS)*, pages 132–142. IEEE, 2018.
- [43] Luigi Rizzo. pgmcc: a tcp-friendly single-rate multicast congestion control scheme. *ACM SIGCOMM Computer Communication Review*, 30(4):17–28, 2000.
- [44] Muhammad Shahbaz, Lalith Suresh, Jennifer Rexford, Nick Feamster, Ori Rottenstreich, and Mukesh Hira. Elmo: Source routed multicast for public clouds. In *Proceedings of the ACM Special Interest Group on Data Communication*, pages 458–471. 2019.
- [45] Jiaxin Shi, Youyang Yao, Rong Chen, Haibo Chen, and Feifei Li. Fast and concurrent {RDF} queries with {RDMA-Based} distributed graph exploration. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, pages 317–332, 2016.
- [46] Jörg Widmer and Mark Handley. Extending equation-based congestion control to multicast applications. In *Proceedings of the 2001 conference on Applications, technologies, architectures, and protocols for computer communications*, pages 275–285, 2001.
- [47] Yibo Zhu, Haggai Eran, Daniel Firestone, Chuanxiong Guo, Marina Lipshteyn, Yehonatan Liron, Jitendra Padhye, Shachar Raindel, Mohamad Haj Yahia, and Ming Zhang. Congestion control for large-scale rdma deployments. *ACM SIGCOMM Computer Communication Review*, 45(4):523–536, 2015.

A More about Multicast Table Registration

The in-fabric logic of Gleam is based on the switches’ multicast forwarding table. Gleam follows the table information to copy packets and forward them to specific output ports. Although existing multicast solutions commonly adopt the table-based approach, *e.g.*, IP multicast [20], the multicast table registration in Gleam has some primary differences from them.

The first difference is that the table registration of Gleam is centralized, while the traditional works [20, 28, 29] perform a distributed registration algorithm [?]. The critical insight pushing registration from distributed to centralized is that the datacenter is highly autonomous, and the multicast membership is highly controlled. This centralized approach is widely used in various datacenter frameworks [25, 44]. Secondly, previous works only register layer-3 states to switch. Gleam, on the other hand, registers both layer-3 states and layer-4 states to switches for the connectivity and reliability supports, as introduced in §3.

Gleam’s registration protocol is centralized and relied on an application-assigned master node. Before communication starts, the master node in the multicast group collects the connection states (including the layer-3 IP and layer-4 IB information) of all other members through an out-of-band protocol (*e.g.*, TCP). Then, the master node fits these states into the

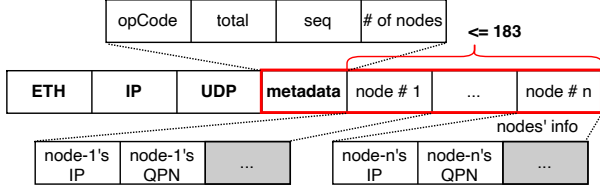


Figure 17: The *envelope* packet format.

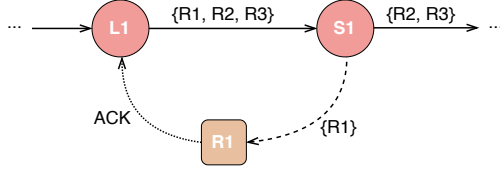


Figure 18: An example of *envelope* packets transmission.

self-developed *envelope* protocol's packet layout, which is illustrated in Fig. 17. The *envelope* packet is identified by the destination IP (*i.e.*, GroupIP) and a specific UDP port number. The payload contains metadata and detailed connection states of each node, including the node's IP and QPN. The metadata comprises group statistics, where *seq* and *total* indicate the sequence and the total number of *envelope* packets. Limited by the MTU (typically 1500Bytes), one *envelope* packet can contain at most 183 nodes; thus, the connection states of a multicast group with more than 183 members must span multiple *envelope* packets.

Upon receiving the *envelope* packet, the switch builds its local multicast forwarding table and sends one or more new *envelope* packets to downstream devices. Algorithm 4 illustrates the behavior of Gleam switches. An *envelope* packet (p) carries a GroupIP and an array of multicast member connection states ($p.array$). The switch first generates an empty table indexed by $p.groupIP$. Then the switch iterates over $p.array$ to append items to the created table. For every node in the array, the switch finds this node's routing information through the normal unicast routing table. If this node is directly connected (*e.g.*, connected to $port_i$), then the switch creates a table item with $port$ as i , marks this item's type to *connected*, and fills this node's connection states into this item. Otherwise, if this node isn't directly connected, the switch finds the set of possible output ports (set_p) for this node. If one port in set_p has been marked as *forwarded*, the switch selects this port again to distribute data optimally and save bandwidth. If all ports in set_p are new, the switch selects the least utilized port among set_p to perform a group-level load balancing.

After processing *envelope* packet and filling the multicast forwarding table, the switch generates one or more new *envelope* packets to ports included in the table. The new *envelope* packet that through each port only contains connection states of nodes that select this port. We show the actions that taken by S_1 in Fig. 18, and the whole topology is shown in Fig. 4. The *envelope* packet received by S_1 contains $R_1, R_2,$

Algorithm 4 Multicast Forwarding Table Registration

```

1:  $p \leftarrow$  received envelope packet
2:  $n \leftarrow$  the number of switch ports
3: create multicast forwarding table  $T[p.groupIP]$ 
4:  $R[n] \leftarrow \{0\}$   $\triangleright$  for creating new envelope packets
5: for node in  $p$  do  $\triangleright$  loop over nodes in  $p$  and build  $T$ 
6:   if node is directly connected to port  $i$  then
7:      $out \leftarrow i$   $\triangleright$  create new entry
8:      $Entry \leftarrow$  a new entry
9:      $Entry.port \leftarrow out$ 
10:     $Entry.type \leftarrow connected$ 
11:     $Entry.vaue \leftarrow$  node's connection states
12:     $T[p.groupIP].append(Entry)$ 
13:   else
14:      $S \leftarrow$  the set of accessible ports
15:     if port  $j \in S$  has been marked as forwarded then
16:        $out \leftarrow j$   $\triangleright$  reuse exiting entry
17:     else  $\triangleright$  create new entry
18:        $out \leftarrow$  the least utilized port in  $S$ 
19:        $Entry \leftarrow$  a new entry
20:        $Entry.port \leftarrow out$ 
21:        $Entry.type \leftarrow forwarded$ 
22:        $T[p.groupIP].append(Entry)$ 
23:     update port utilization
24:      $R[out].append(node)$ 
25: // create new envelope packets and send out
26: for  $Entry \in T[p.groupIP]$  do
27:   create envelope packet  $\bar{p}$  that contains nodes in
    $R[Entry.port]$ 
28:   send  $\bar{p}$  through  $Entry.port$ 

```

and R_3 . As instructed by Algorithm 4, S_1 should forward a data copy to $port_{L2}$ (eventually reaching R_1), and a copy to $port_{C2}$ (eventually reaching R_2 and R_3). Thus the *envelope* packets forwarded to $port_{L2}$ and $port_{C2}$ contains information of $\{R_1\}$ and $\{R_2, R_3\}$, respectively, used in downstream devices to build their local forwarding table.

Finally, if a node receives an *envelope* packet, and its IP address is included in the packet, this node will answer an ACK back to the master node to confirm its participation. After the master node collects all nodes' confirmation ACKs, the control-plane table registration is finished, and the multicast transmission can start.

B Multicast Source Switching

Gleam supports the source switching inside a multicast group. When the source of a multicast group changes, the switches can detect this by recognizing the change of the incoming port of multicast data packets. The new incoming port is recorded for later ACK forwarding. There are no other modifications to Gleam's in-fabric logic.

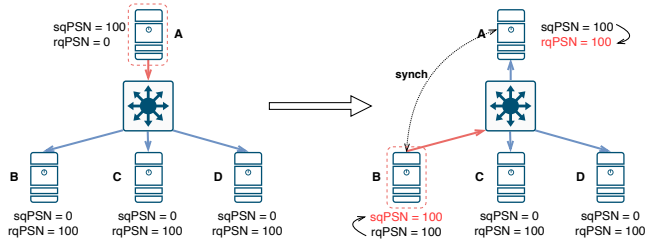


Figure 19: An example of multicast source switching

For the end-host, the old and new source nodes need to run a PSN synchronization procedure. Note that each QP maintains two PSN records. The *Send Queue PSN* (sqPSN) is used to record the output packets, and the *Receive Queue PSN* (rqPSN) is used to verify the input packets. The senders' sqPSN should equal the receiver's rqPSN at the beginning of the transmission. A PSN synchronization is needed to maintain this PSN consistency when the multicast source changes. For example, as shown in Fig. 19, node *A* has multicasted 100 packets to nodes *B*, *C*, and *D*. Assume that all nodes' sqPSN and rqPSN start from 0, the sqPSN of *A* and rqPSNs of *B*, *C*, *D* become 100 when the transmission ends. When the multicast source switches to *B*, if *B* starts transmission immediately, the PSN of sent packets would be *B*'s current sqPSN, *i.e.*, 0. These packets would be dropped by *C* and *D* as their rqPSNs are already 100. Therefore, the old and new multicast source nodes need to synchronize their PSNs. In particular, the old source node assigns its rqPSN as its sqPSN, and the new source node assigns its sqPSN as its rqPSN. This problem can also be avoided by using Dynamic Connected Transport (DCT) [21], which synchronizes the PSN of the sender and the receiver with the DC Connect packet.

C Optimization for one-to-many WRITE

As mentioned in §3.3, to support one-to-many WRITE, we need an additional message to carry the MR info of different receivers for each WRITE request. This incurs extra bandwidth overhead, especially when the number of receivers is large. A straight idea to eliminate this overhead is to enable the possibility to write all receivers with the same MR info. We claim this idea is logically reasonable and easy to implement with few modifications to the RNICs. Note that the VA in MR is *virtual* and is translated to the physical memory address by the RNIC. Therefore, it is reasonable to use an identical virtual address for all nodes, which can be translated to different physical addresses at different nodes. On the other hand, the *R_key* is mainly used as an index for the RNIC to look up the Memory Translation Table (MTT). All receivers in a multicast group can share the same *R_key* for the corresponding MR if some indexes are reserved for this purpose.

Currently, the VA is assigned by the OS kernel, and the *R_key* is obtained from the RNIC. If the RNIC is modified so

that the application can assign the VA and *R_key* for an MR at the QP establishment phase, we can enable all receivers to use the same VA and *R_key*. Consequently, the switch no longer needs to modify the MR info for different receivers, and the overhead for MR info update is avoided.