1Pipe: Scalable Total Order Communication in Data Center Networks

Bojie Li
Huawei Technologies

Gefei Zuo
University of Michigan

Wei Bai
Microsoft Research

Lintao Zhang
BaseBit Technologies

ABSTRACT

This paper proposes 1Pipe, a novel communication abstraction that enables different receivers to process messages from senders in a consistent total order. More precisely, 1Pipe provides both unicast and *scattering* (*i.e.*, a group of messages to different destinations) in a causally and totally ordered manner. 1Pipe provides a best effort service that delivers each message at most once, as well as a reliable service that guarantees delivery and provides restricted atomic delivery for each scattering. 1Pipe can simplify and accelerate many distributed applications, *e.g.*, transactional key-value stores, log replication, and distributed data structures.

We propose a scalable and efficient method to implement 1Pipe inside data centers. To achieve total order delivery in a scalable manner, 1Pipe separates the bookkeeping of order information from message forwarding, and distributes the work to each switch and host. 1Pipe aggregates order information using in-network computation at switches. This forms the "control plane" of the system. On the "data plane", 1Pipe forwards messages in the network as usual and reorders them at the receiver based on the order information.

Evaluation on a 32-server testbed shows that 1Pipe achieves scalable throughput (80M messages per second per host) and low latency (10μs) with little CPU and network overhead. 1Pipe achieves linearly scalable throughput and low latency in transactional key-value store, TPC-C, remote data structures, and replication that outperforms traditional designs by 2~20x.

CCS CONCEPTS

• **Networks** → **In-network processing**; **Data center networks**;
• **Computer systems organization** → *Reliability*;

KEYWORDS

Total Order Communication, CATOCS, Data Center Networks, In-Network Processing

ACM Reference Format:
Bojie Li, Gefei Zuo, Wei Bai, and Lintao Zhang. 2021. 1Pipe: Scalable Total Order Communication in Data Center Networks. In *ACM SIGCOMM 2021 Conference (SIGCOMM '21), August 23–28, 2021, Virtual Event, USA.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3452296.3472909

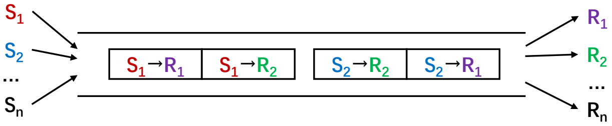


Figure 1: *1Pipe abstraction.*

1 INTRODUCTION

The lack of total order communication often complicates distributed system design. For example, when a host atomically reads or writes multiple objects on different remote hosts, there is no guarantee that the messages arrive at different remote hosts at the same time, so, locks are often required to achieve consistency. As another example, multiple shards of a distributed database generate logs to multiple replicas, and each replica may receive logs from the shards in a different interleaving ordering, thus violating data consistency.

As a reaction to this complexity, we propose 1Pipe, a communication primitive that provides "one big pipe" abstraction in a data center network (DCN). As Figure 1 shows, messages are sent in groups and serialized in a virtual pipe, which enables different receiver processes to deliver messages from sender processes in a consistent order. More precisely, 1Pipe resembles Causally and Totally Ordered Communication Support (CATOCS) [25]: (1) messages are *totally ordered*, ensuring that they are delivered in the same order to all receivers; (2) messages are delivered obeying the *causal order* in the Lamport logical clock sense [63]. In addition to unicast, 1Pipe also supports *scattering*, which groups multiple messages to different receivers at the same position of the total order. Different from traditional multicast, each message in a scattering has distinct content and destination. Users do not need to define multicast channels or groups because the network is a big CATOCS channel.

1Pipe can achieve distributed atomic multi-object read and write with a single scattering because they are delivered at the same logical time. Replication in 1Pipe takes one round-trip time (RTT). 1Pipe also provides a total ordering of communication events, thus reducing fences and improving concurrency of distributed systems.

1Pipe seems to require a central serialization point, which is not scalable. In this work, we propose a scalable and efficient implementation of 1Pipe in a DCN, where the topology is regular [43, 67], and switches have generally good programmability. Our principle is to co-design end hosts with the underlying DCN. We synchronize the clocks on hosts and ensure they are non-decreasing. The sender attaches the same timestamp to each packet in a unicast message or scattering. Each receiver delivers messages in non-decreasing timestamp order.

At its core, 1Pipe separates the bookkeeping of order information from message forwarding. 1Pipe forwards timestamped packets as usual in the network, and buffers them at the receiver side. The key challenge is to let a receiver know that all the packets below a certain timestamp have arrived. To this end, we introduce a *barrier*

This work was done when all authors were with Microsoft Research.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
SIGCOMM '21, August 23–28, 2021, Virtual Event, USA
© 2021 Association for Computing Machinery.
ACM ISBN 978-1-4503-8383-7/21/08…$15.00
https://doi.org/10.1145/3452296.3472909

timestamp on each link, which is essentially the *lower bound* of the timestamps of all future arrival packets. Each switch aggregates barrier information of all the ingress links to derive the barrier for all the egress links. In this way, barriers propagate in the DAG (Directed Acyclic Graph) network along with packet forwarding, and the receiver can deliver the messages with timestamps below the barrier in order. If some hosts or links are temporarily idle, we periodically generate hop-by-hop *beacon* packets carrying barrier information.

Regarding packet losses and failures, 1Pipe provides a *best effort* service in which lost messages are not retransmitted; and a *reliable* service in which a message is guaranteed to be delivered if sender and all receivers in the scattering do not fail. When a host or switch fails, reliable 1Pipe ensures *restricted failure atomicity*: either all or none messages in a scattering are delivered unless a receiver fails permanently or network partitions. Reliable 1Pipe uses a two-phase commit (2PC) approach, where the first phase is end-to-end packet loss recovery, and the second phase aggregates *commit barriers* through the network. It relies on a highly available network controller to coordinate failure handling, and allows applications to customize failure handling via callbacks. Reliable 1Pipe adds one round-trip time (RTT) compared to best effort 1Pipe. We will show in Sec.2.2 that best effort 1Pipe can achieve replication and read-only distributed atomic operations without the extra RTT of reliable 1Pipe.

We implement three incarnations of 1Pipe on network devices with different programming capabilities: programmable switching chips [4, 45] that can support flexible stateful per-packet processing, switch CPUs, and host CPUs in case that switch vendors do not expose accesses to switch CPUs.

We evaluate 1Pipe in a 32-server cluster with 10 switches and 3-layer fat-tree topology. 1Pipe achieves linearly scalable throughput with 512 processes, achieving 5M messages per second per process (80M msg/s per host). Best effort 1Pipe adds up to 10 μs delay to message delivery, while reliable 1Pipe adds up to 21 μs. 1Pipe has robust performance under packet loss, and can recover from failures in 50~500 μs. 1Pipe only needs 0.3% network bandwidth overhead and a CPU core per switch for periodic beacons.

As case studies, first, 1Pipe scales linearly for a transactional key-value store (KVS) in both uniform and YCSB [26] workload, whose throughput is 90% of a non-transactional system (hardware limit) and outperforms FaRM [34] by 2~20x especially under high contention. The latency of 1Pipe is consistently low. Second, 1Pipe scales linearly in TPC-C [29] benchmark, which outperforms Lock and OCC by 10x. 1Pipe's performance is resilient to packet loss. Third, by removing fences and enabling replicas to serve reads, 1Pipe improves remote data structure performance to 2~4x. Finally, 1Pipe reduces Ceph [96] replication latency by 64%.

In summary, the contributions of this paper are: (1) a novel abstraction 1Pipe that provides causally and totally ordered unicast and scattering with best-effort and reliable semantics; (2) design and implementation of scalable and efficient 1Pipe in DCNs; (3) design and evaluation of 1Pipe applications: transactional KVS, independent transactions, remote data structure, and replication.

This work does not raise any ethical issues.

2 MOTIVATION

2.1 Abstractions of 1Pipe

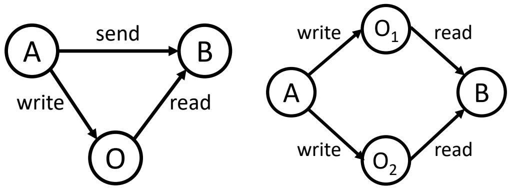
1Pipe provides a *causally and totally ordered* communication abstraction in a distributed system with multiple hosts, where each host has multiple processes. Each process has two roles: sender and receiver. Each host maintains a monotonically increasing timestamp, which represents the wall clock and is synchronized among all hosts. A message consists of one or more packets. When a process sends a group of messages (*i.e.*, a *scattering*) to different destination processes, all packets of the messages are labeled with the same timestamp of the sender host. Unlike multicast, the list of destinations can be different for each scattering, in which messages to different receivers can have different content. The total order property is: *each process delivers messages from different processes in non-decreasing timestamp order.* The causality property is: *when a receiver delivers a message with timestamp T, the timestamp of the host must be higher than T.*

onepipe_unreliable_send(vec[<dst, msg>])
TS, src, msg = onepipe_unreliable_recv()
onepipe_send_fail_callback(func(TS, dst, msg))
onepipe_reliable_send(vec[<dst, msg>])
TS, src, msg = onepipe_reliable_recv()
onepipe_proc_fail_callback(func(proc, TS))
TS = onepipe_get_timestamp()
onepipe_init()
onepipe_exit()

Table 1: *Programming API of 1Pipe. Vec[<dst, msg>] indicates a scattering of messages that have the same timestamp.*

As Table 1 shows, 1Pipe provides two services with different reliability guarantees: first, *onepipe_unreliable_send/recv* is a *best effort* service where packet loss is possible. 1Pipe guarantees causal and total order properties by buffering messages at the receiver and only delivering them when it receives a barrier timestamp aggregated from the network. So, best effort 1Pipe delivers message in 0.5 RTT plus barrier wait time. Best effort 1Pipe detects lost packets via end-to-end ACK, but does not retransmit them. Modern data centers typically have low network utilization and deploy advanced congestion control mechanisms [10, 48, 61, 75, 101]. Measurements show that intra-pod packet drop rate is on the order of 10^{-5} [44]. In RDMA networks with PFC, because congestion loss is eliminated, packet corruption rate should be below 10^{-8} according to IEEE 802.3 standard, and links with corruption rate higher than 10^{-6} are considered to be faulty [102]. So, data center applications can assume that best effort 1Pipe is almost reliable but should use *onepipe_send_fail_callback* to detect lost packets due to packet corruption or failure of the network or remote process. Loss recovery is up to the application in this case.

Second, *onepipe_reliable_send/recv* is a *reliable* service which in addition to ordering, guarantees reliability: *a message is guaranteed to be delivered if sender process, receiver process and the network do not fail.* It retransmits packets in the case of packet losses. In this paper, we only consider crash failures. When a process or network fails, message delivery stalls. 1Pipe removes in-flight messages from or to the failed process. If a message cannot be delivered, the send



(a) Write after write. *(b) Independent read, independent write.*

Figure 2: *Ordering hazards in a distributed system.*

failure callback is invoked on the sender. In addition, each process may register a callback function via *onepipe_proc_fail_callback* to get notifications of failed processes. Message delivery is resumed after all non-fail processes finish their callback functions.

Reliable 1Pipe also provides *restricted failure atomicity*, which means all-or-nothing delivery of *a scattering of messages* with the exception that if a receiver fails permanently or network partitions after the decision to deliver is made, the receiver can never deliver it. If a receiver recovers from failures, it can deliver or discard messages consistently with the other receivers in the same scattering. In fact, full failure atomicity is impossible in a non-replicated fail-stop model, because a receiver or its network link may fail permanently before delivering T almost simultaneously with another receiver delivering T [38]. Reliable 1Pipe achieves atomicity via two-phase commit and in-network barrier aggregation, so messages delivery needs 1.5 RTTs plus barrier wait time.

In fault tolerant applications, 1Pipe provides a fast path in normal cases, and falls back to the application for customized failure handling. More concretely, an application may use state machine replication to replicate its states, and register *onepipe_proc_fail_callback* which invokes a traditional consensus algorithm [65, 80]. Each message is scattered to all replicas. When failure occurs, message delivery is stalled, and 1Pipe invokes the callbacks in all non-fail processes. Restricted failure atomicity ensures that all correct replicas deliver the same sequence of messages. If the correct replicas reach a quorum, the callbacks return, and message delivery is resumed. Otherwise, there are too many failed replicas, and the application can choose between consistency and availability. If it chooses consistency and waits for some replicas to recover, the recovered replicas can deliver the same sequence of messages.

2.2 Use Cases of 1Pipe

2.2.1 Total Ordering of Communication Events. Production data centers provide multiple paths between two nodes [8, 43]. Due to different delays of different paths, several categories of ordering hazards [41, 89] may take place (Figure 2).

Write after write (WAW). Host A writes data to another host O, then sends a notification to host B. Send can be considered as a write operation. When B receives the notification, it issues a read to O, but may not get the data due to the delay of A’s write operation.

Independent read, independent write (IRIW). Host A first writes to data O_1 and then writes to metadata O_2. Concurrently, host B reads metadata O_2 and then reads data O_1. It is possible that B reads the metadata from A but the data is not updated yet.

Ordering hazards affect system performance. To avoid the WAW hazard, A needs to wait for the first write operation to complete (an RTT to O, called a *fence*) before sending to B, thus increasing latency. To avoid the IRIW hazard, A needs to wait for write O_1 to complete before initiating write O_2, and B needs to wait for read O_2 to complete before initiating read O_1. The fence latency will be amplified when more remote objects need to be accessed in order.

1Pipe can remove both WAW and IRIW hazards due to causality and total ordering. In WAW case, by monotonicity of host timestamp, $A \rightarrow O$ is ordered before $A \rightarrow B$. By causality, $A \rightarrow B$ is ordered before $B \rightarrow O$. Consequently, $A \rightarrow O$ is ordered before $B \rightarrow O$. Therefore, the write operation is processed before the read operation at host O, thus avoiding WAW hazard. By removing the fence between $A \rightarrow O$ and $A \rightarrow B$, 1Pipe reduces the end-to-end latency from 2.5 RTTs to 1.5 RTTs, in the absence of packet losses.

If an application needs to process many WAW tasks *in sequence*, the power of 1Pipe is amplified. Using the traditional method, the application needs 1 RTT of idle waiting during each WAW task, so, the throughput is bounded by 1/RTT. In contrast, using 1Pipe, the application can send dependent messages in a pipeline.

The argument above neglects possible packet losses. In best effort 1Pipe, there is a small possibility that message $A \rightarrow O$ is lost in flight, so every object needs to maintain a version, and B needs to check the version of O. If it does not match the version in the notification message from A, then B needs to wait for A to retransmit O and re-notify B. A registers send failure callback and performs rollback recovery when A is notified of a send failure.

If we use reliable 1Pipe, objects no longer need versioning, but the end-to-end latency increases from 2.5 to 3.5 RTTs. However, A can still send messages in a pipeline, and have much higher throughput than the traditional way. In addition, reliable 1Pipe can preserve causality in failure cases [16]: if A fails to write to O, message $A \rightarrow B$ will not be delivered, similar to Isis [19].

Similarly, 1Pipe removes IRIW hazard and improves minimum end-to-end latency from 3 RTTs to 1 RTT by eliminating two fences. The minimum latency is achieved when A and B initiate read and write simultaneously.

1Pipe’s power to remove ordering hazards comes from its ability to totally order communication events. This power is also a perfect match with *total store ordering* (TSO) memory model [89] in a distributed shared memory (DSM) system. In TSO, each processor observes a consistent ordering of writes from all other cores. In other words, processors must not observe WAW and IRIW hazards. Compared to weaker memory models, TSO reduces synchronization in concurrent programming [77, 93], thereby simplifying programming and reducing fence overheads.

2.2.2 1-RTT Replication. Replication is essential for fault tolerance. Traditional multi-client replication requires 2 RTTs because client requests must be serialized (*e.g.*, sent to a primary) before sending to replicas [81]. With 1Pipe, we can achieve 1-RTT replication without making assumptions on log content, because the network serializes messages. A client can directly send a log message to all replicas with a scattering, and each replica orders logs according to the timestamp (ties are broken by the client ID). Because reliable 1Pipe has an extra RTT, we use unreliable 1Pipe, and handle packet losses and failures in a more clever way. First, to ensure ordered delivery of logs between each pair of client and replica, they maintain a sequence number. The replica rejects messages with non-consecutive sequence numbers. Second, to detect inconsistent logs due to packet losses, each replica maintains a checksum of all previous log messages from all clients. When a replica receives a message, it adds the message timestamp to the checksum, and returns the checksum to the client. The response message does not need to be ordered by 1Pipe. If a client sees all checksums are equal from the responses, the logs of replicas are consistent at least until the client's log message, and the client knows that the replication succeeds. Otherwise, there must be lost messages to some replica or failure of a client or a replica. In the case of packet loss, the client simply retransmits the messages since the first rejected one. In the case of suspected replica failure, the client notifies all replicas (or a cluster manager) to initiate a failure recovery protocol, which uses a traditional consensus protocol to remove inconsistent log entries and make checksums match. When there is no packet loss and failure, replication only needs 1 RTT.

Similar to replication, 1Pipe can achieve state machine replication (SMR) [63] or virtual synchrony [17]. In a SMR-based distributed system, each message is broadcast to all processes, and each process uses the same sequence of input messages. SMR can solve arbitrary synchronization problems [63]. An example is mutual exclusion that requires the resource to be granted in the order that the request is made [63]. With reliable 1Pipe, using SMR to implement the lock manager can solve the mutual exclusion problem.

2.2.3 Distributed Atomic Operation (DAO). A DAO is a transaction that atomically reads or updates objects in multiple hosts. DAO is widely used in transactional key-value store [32], caches in web services, distributed in-memory computing, and index cache of distributed storage. Traditionally, a DAO needs 3 RTTs: (1) lock the objects; (2) if all locks succeed, send operations to the participants; (3) unlock the objects. Using reliable 1Pipe, a DAO is simply a scattering with the same timestamp from the initiator. Each recipient processes messages from all DAOs in timestamp order. So, the DAOs are serializable. If a recipient or the network permanently fails, atomicity violation is not observable because the objects cannot be accessed by any subsequent operations.

As an optimization, we can use unreliable 1Pipe for read-only DAOs because if it fails due to packet losses, the initiator can retry it. In terms of SNOW [72] and NOCS [73] theorems, 1Pipe provides 1-RTT read-only DAOs with strict serializability, no storage overhead and close-to-optimal throughput, but at the expense of blocking operations until receiving the barrier timestamp.

2.2.4 Other Scenarios. In general transactions with Opacity [90], to obtain read and write timestamps, a transaction needs to query a non-scalable centralized sequencer [35, 57] or wait for clock uncertainty (*e.g.*, Spanner [27] waits ~10ms and FaRMv2 [90] waits ~20μs). 1Pipe can use local time as transaction timestamps directly without waiting because lock messages of previous transactions must be delivered to shards before data accesses of the current transaction.

1Pipe timestamp is also a global synchronization point. For example, to take a consistent distributed snapshot [24], the initiator broadcasts a message with timestamp T to all processes, which directs all processes to record its local state and in-flight sent messages with a higher timestamp than T.

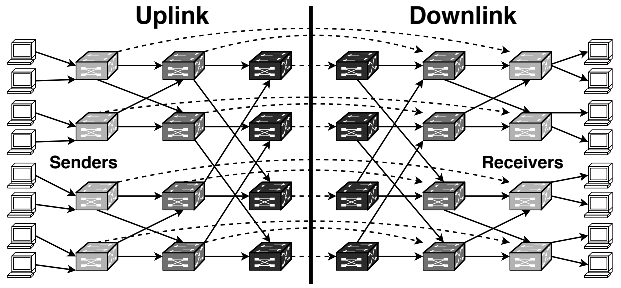


Figure 3: *Routing topology of a typical data center network. Each physical switch is split into two logical switches, one for uplink and one for downlink. Dashed virtual link between corresponding uplink and downlink switch indicates "loopback" traffic from a lower-layer switch or host to another one.*

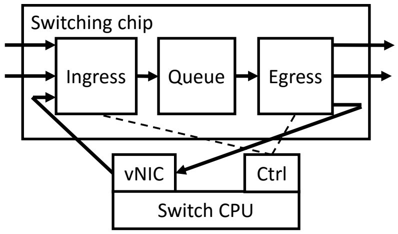


Figure 4: *Architecture of a typical network switch.*

3 BACKGROUND

In this section, we introduce unique characteristics of data center network (DCN), which make a scalable and efficient implementation of 1Pipe possible.

3.1 Data Center Network

Modern data centers typically adopt multi-rooted tree topologies [8, 43] to interconnect hundreds of thousands of hosts. In a multi-rooted tree topology, the shortest path between two hosts first goes up to one of the lowest common ancestor switches, then goes down to the destination. Therefore, the routing topology form a directed acyclic graph (DAG), as shown in Figure 3. This loop-free topology enables hierarchical aggregation of barrier timestamps.

A highly available SDN *controller* runs on the management plane to detect failures of switches and links, then reconfigure routing tables on failures [42, 91].

3.2 Programmable Switches

A data center switch consists of a switching chip [3, 45] and a CPU (Figure 4). The switch operating system [2] runs on the CPU to reconfigure the switching chip. The switching chip forwards selected traffic (typically control plane traffic, *e.g.*, DHCP and BGP) to the CPU via a virtual NIC.

The switching chip is composed of an ingress pipeline, multiple FIFO queues and an egress pipeline. When a packet is received from an *input link*, it first goes through the ingress pipeline to determine the output link and queueing priority, then is put in the corresponding FIFO queue. The egress pipeline pulls packets

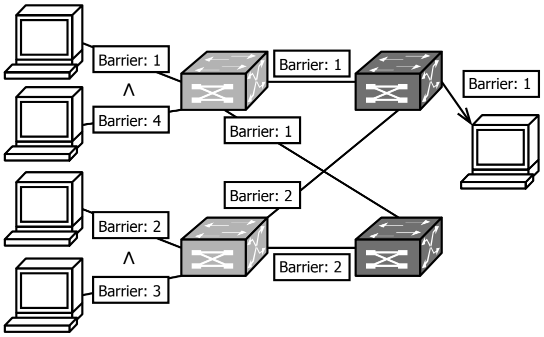


***Figure 5:** Hierarchical barrier timestamp aggregation.*

from queues according to priority, applies header modifications and sends them to *output links*. One key property of the switch is that the queuing model ensures *FIFO* property for packets with the same priority, ingress port and egress port.

The switch can provide good programmability. First, the CPU can be used to process (a small amount of) packets [71]. Second, the switching chip is increasingly programmable in recent years. For example, Tofino chip [45] supports flexible packet parsers and stateful registers. Users can program Tofino using P4 [20] to achieve *customized stateful per-packet processing.*

Despite good programmability, the switch typically has limited buffer resource to hold packets. The average per-port packet buffer is typically hundreds of kilobytes [14, 15] in size. As a result, it is challenging to buffer many packets at the switches in a data center.

4 BEST EFFORT 1PIPE

Best effort 1Pipe provides total order, causal, and FIFO message delivery, but does not retransmit lost packets and does not provide atomicity. A naive approach to realize this is ordering all messages with a centralized sequencer, which would be a bottleneck. Instead, we will introduce how to achieve scalable ordering in the regular DCN topology.

4.1 Message and Barrier Timestamp

Message timestamp. 1Pipe sender assigns a non-decreasing timestamp for each message. Messages in a scattering have the same timestamp. Given recent efforts on μs accurate clock synchronization in data centers [28, 40, 66, 69], we synchronize monotonic time of hosts, and use the local clock time as message timestamps. Clock skew slows down delivery but does not violate correctness. Message timestamp determines the delivery order at a receiver. The receivers deliver arrival messages in ascending order of their timestamps (ties are broken through sender ID).

Barrier timestamp. When a receiver delivers a message with timestamp T, it must be sure that it has received and delivered all the messages whose timestamps are smaller than T. A straightforward approach is to only deliver messages with non-decreasing timestamps, and drop all out-of-order messages. However, since different network paths have different propagation and queuing delays, this approach will drop too many messages, *e.g.*, 57% received messages are out-of-order in our experiment where 8 hosts send to one receiver. To this end, we introduce the concept of *barrier timestamp*. A barrier timestamp is associated with either a link or a node (*i.e.*, a switch or an end host) in the routing graph. The barrier timestamp is the lower bound of message timestamps of all *future arrival messages* from the link or at the node. Each receiver maintains its own barrier timestamp and deliver the messages whose timestamps are smaller than the barrier timestamp.

If the transport between sender and receiver is FIFO, a receiver can easily figure out the barrier if it has received messages from all the senders: the barrier is the minimum timestamp of latest messages from all the senders. Therefore, a naive solution would be for every sender to send timestamped messages to every receiver so that the receivers can figure out the barrier and deliver messages. Unfortunately, this solution requires sending messages to receivers not in the scattering group, which does not scale.

Hierarchical barrier timestamp aggregation. 1Pipe exploits the knowledge of the queuing structure of the network, making the lower bound aggregation much more scalable and efficient than if it was implemented at only a logical level. 1Pipe leverages programmable switches to *aggregate* the barrier timestamp information. Given the limited switch buffer resource, 1Pipe does not reorder messages in the network. Instead, 1Pipe forwards messages in the network as usual, but reorders them at the receiver side based on the barrier timestamp information provided by the switch.

In 1Pipe, we attach two timestamp fields to each message packet. The first is *message timestamp* field, which is set by the sender and will not be modified. The second field is *barrier timestamp*, which is initialized by the sender but will be modified by switches along the network path. The property of the barrier timestamp field is:

When a switch or a host receives a packet with barrier timestamp B from a network link L, it indicates that the message timestamp and barrier timestamp of future arrival packets from link L will be larger than B.

To derive barriers, the sender initializes both fields of all the packets in a message with the non-decreasing message timestamp. The switch maintains a register R_i for each input link $i \in \mathcal{I}$, where $\mathcal{I}$ is the set of all input links. After forwarding a packet with barrier timestamp B from input link i to output link o, the switch performs two updates. First, it updates register $R_i := B$. Second, it modifies the barrier timestamp of the packet to B_{new} as follows:

$$B_{new} := \min\{R_i | i \in \mathcal{I}\} \tag{4.1}$$

Using (4.1), each switch independently derives the barrier timestamp based on all input links. As shown in Figure 5, barrier timestamps are aggregated hierarchically through layers of switches hop-by-hop, and finally the receiver gets the barrier of all reachable hosts and links in the network. This algorithm maintains the property of barrier timestamps, given the FIFO property of each network link.

When the receiver receives a packet with barrier timestamp B, it first buffers the packet in a priority queue that sorts packets based on the message timestamp. The receiver knows that the message timestamp of all future arrival packets will be larger than B. Hence, it delivers all buffered packets with the message timestamp below B to the application for processing. If a process receives a message with timestamp above B, it is dropped, and a NAK message is returned to the sender. So, if a link or switch is not strictly FIFO, out-of-order messages will not violate correctness. Notice that barrier aggregation only relies on hop-by-hop FIFO links instead of a FIFO end-to-end transport. Therefore, 1Pipe can work with a variety of multi-path routing schemes [9, 21, 33, 47, 58, 95, 100].

Causality. To preserve causal order in the Lamport logical clock sense [63], the local clock time should be higher than delivered timestamps. This is implied by timestamp aggregation because each process has both sender S and receiver R roles, and a barrier T received by R is aggregated from all senders including S. Because the local clock is monotonic, S's timestamp is higher than T when R receives barrier T.

4.2 Beacons on Idle Links

As shown before, at each hop, the per-packet barrier timestamp is updated to the minimum barrier timestamp value of all possible input links. As a result, an idle link stalls the barrier timestamp, thus throttling the whole system. To avoid idle links, we send *beacons* periodically on every link.

What is a beacon? Unlike the message packet, the beacon packet only carries the barrier timestamp field and has no payload data.

How to send beacons? We send beacon packets on a per-idle-link basis. Beacon packets can be sent by both hosts and switches, but the destination must be its one-hop neighbors. This hop-by-hop property ensures that the beacon overhead is unrelated to network scale. For a beacon generated by the host, the barrier timestamp is the host clock time. For a beacon generated by the switch, the barrier timestamp is initialized according to (4.1).

When to send beacons? We introduce a beacon interval T_{beacon}. When a host or a switch output link does not observe any message packet for T_{beacon} time, it generates a beacon packet. We should choose a suitable T_{beacon} to balance bandwidth overhead and system delay. The beacons are sent at synchronized times on different hosts, so that when network delays are identical, a switch receives beacons almost simultaneously. If beacons were sent on random times, the switch must wait for the last beacon to come, which increases expected delay of a message by nearly one beacon interval. With synchronized beacons, the expected delay overhead is only half of the beacon interval.

Handling failures. When a host, link, or switch fails, the barrier timestamp of its neighbors stop increasing. In order to detect failures, each switch has a timeout timer per input link. If no beacon or data packet is received for a timeout, *e.g.*, 10 beacon intervals, the input link is considered to be dead and removed from the input link list. After removal of the failed link, the barrier timestamp resumes increasing. This failure handling mechanism is decentralized.

Addition of new hosts and links. For a new host, it synchronizes clock with the time master. When a link is added between a host and switch or between two switches, because the switch must maintain monotonicity of its B_{new}, it suspends updates to B_{new} until the barrier received from the new link is greater than B_{new}.

5 RELIABLE 1PIPE

Now, we present the design of reliable 1Pipe that can handle packet loss and failure.

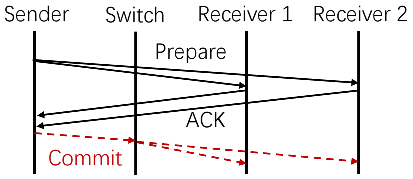


***Figure 6:** Two Phase Commit in reliable 1Pipe.*

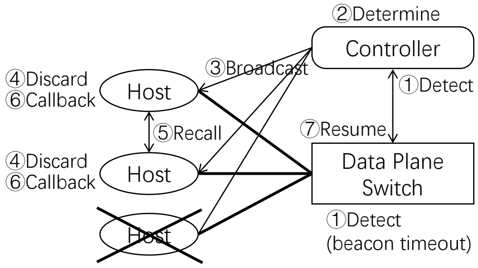


***Figure 7:** Failure recovery in reliable 1Pipe.*

5.1 Handling Packet Loss

When a receiver delivers a message with timestamp T, it must make sure that all messages below T are delivered. So, if a receiver is unaware of packet loss, it cannot reliably deliver messages according to the barrier timestamp. Even if the switch is capable to detect lost packets, there is still a problem. For example, host A sends to B, then sends to C via a different path. Three events happen according to the following order: $A \longrightarrow B$ is lost; $A \longrightarrow C$ arrives; A crashes. In this case, $A \longrightarrow C$ is delivered, while $A \longrightarrow B$ cannot be recovered. The failure to deliver $A \longrightarrow B$ and the delivery of $A \longrightarrow C$ violate reliable ordering property.

Our key idea to handle packet losses is a *Two Phase Commit (2PC)* approach:

- **Prepare phase**: The sender puts messages into a send buffer, and transmits them with timestamps. Switches along the path do NOT aggregate timestamp barriers for data packets. The receiver stores messages in a receive buffer, and replies with ACKs. The sender uses ACKs to detect and recover packet losses.
- **Commit phase**: When a sender collects all the ACKs for data packets with timestamps below or equal to T, it sends a *commit message* that carries *commit barrier* T. The commit message is sent to the neighbor switch rather than the receivers, as the red arrow in Figure 6 shows. Each switch aggregates the minimum commit barriers on input links, and produce commit barriers that propagate to output links. This timestamp aggregation procedure is exactly the same as Sec.4.1. A receiver delivers messages below or equal to T in the receive buffer when it receives a commit barrier T. Similar to best effort 1Pipe, commit messages also need periodic beacons on idle links.

5.2 Handling Failure

Like in Sec. 4.2, crash failure of a component is detected by its neighbors using timeout. However, a failed component cannot be simply removed, because otherwise, the in-flight messages sent by the failed component cannot be consistently delivered or discarded.

To achieve restricted failure atomicity in Sec.2.1, we use the network controller in data centers to detect failures via beacon timeout. The controller itself is replicated using Paxos [65] or Raft [80], so, it is highly available, and only one controller is active at any time.

The controller needs to determine which processes fail and when they fail. The former question is easier to answer. *A process that disconnects from the controller in a routing graph is regarded as failed.* For example, if a host fails, all processes on it are considered as failed. If a Top-of-Rack (ToR) switch fails, and hosts in the rack are only connected to one switch, then all processes in the rack fail.

The latter question, when processes fail, is harder to answer. The challenge is that we cannot reliably determine the last committed and last delivered timestamp of a process. Because there is a propagation delay from committing a timestamp to delivering the timestamp to receivers, it is possible to find a timestamp T committed by a failed process P but not propagated to any receiver, so that all receivers have received messages from P before T in the receive buffer (and hence can deliver them), but no messages after T have been delivered, so they can be discarded. *Failure timestamp* of P is defined as such, which is computed as the maximum last commit timestamp reported by all neighbors of P. If multiple failures occur simultaneously, we try to find a set of correct nodes in a routing graph that separates failed nodes and all correct receivers. If such a set cannot be found due to network partition, then we use a greedy algorithm to find a set to separate as many receivers as possible. The non-separable receivers sacrifice atomicity because some messages after T may have been delivered.

The procedure to handle failure is as follows and shown in Figure 7: (see Appendix for correctness analysis)

- **Detect**: The neighbors of failed components notifies controller along with its last commit timestamp T.
- **Determine**: Controller determines failed processes and their failure timestamps according to the routing graph.
- **Broadcast**: Controller broadcasts the failed processes P and its failure timestamp T to all correct processes.
- **Discard**: Each correct process discards messages sent from P with timestamp higher than T in the receive buffer.
- **Recall**: Each correct process discards messages sent to P in send buffer, which are waiting for ACK from P. If a discarded message is in a scattering, according to failure atomicity, the scattering needs to be aborted, *i.e.*, messages to other receivers *in the same scattering* need to be recalled. The sender sends a *recall* message to such receivers, then each of the receivers discards the messages in the receive buffer and responds ACK to the sender. The sender completes Recall after collecting the ACKs.
- **Callback**: Each correct process executes the process failure callback registered in Table 1, which enables applications to customize failure handling. Then, it responds controller with a completion message.
- **Resume**: Controller collects completions from all correct processes, and then notifies network components to remove input link from the failed component, thereby resuming barrier propagation.

Controller Forwarding. If a network failure affects connectivity between S and R, the Commit phase in 2PC and the Recall step in failure handling may stall because S repeatedly retransmits a message but cannot receive ACK from R. In this case, S asks controller to forward the message to R, and waits for ACK from the controller. If controller also cannot deliver the message, R will be announced as failed, and the undeliverable recall message is recorded. If controller receives ACK of a recall message but cannot forward it to S, S will be announced as failed. In summary, if a process does not respond controller within timeout, it is considered as failed.

Receiver Recovery. If a process recovers from failure, *e.g.*, the network link or switch recovers, the process needs to consistently deliver or discard messages in the receive buffer. The controller notifies process of its own failure. Then, the process contacts controller to get host failure notifications since its failure and undeliverable recall messages. After delivering buffered messages, the recovered process needs to join 1Pipe as a new process. This is because if a process can fail and recover multiple times, the controller would need to record all failure and recovery timestamps, adding complexity.

Limitations. If a process fails permanently, the last timestamp it has delivered cannot be known exactly, so, 1Pipe only ensures all correct receivers and recovered receivers deliver messages consistently. In addition, when network partition occurs, separated receivers may deliver messages after failure timestamp. 1Pipe relies on the application to coordinate such failures.

6 IMPLEMENTATION

6.1 Processing on End Hosts

We implement an 1Pipe library, *lib1pipe*, at the end host. The library is built on top of RDMA verbs API. 1Pipe obtains timestamp from CPU cycle counter and assigns it to messages in software. Because RDMA RC buffers messages in different QPs, we cannot ensure timestamp monotonicity on the NIC-to-ToR link. Ideally, we would like to use a SmartNIC and attach timestamps to packets when they egress to the port. However, because we only have access to standard RDMA NICs, we use RDMA UD instead. Each 1Pipe message is fragmented into one or more UD packets.

1Pipe implements end-to-end flow and congestion control in software. When a destination process is first accessed, it establishes a connection with the source process and provisions a receive buffer for it, whose size is the receive window. A packet sequence number (PSN) is used for loss detection and defragmentation. Congestion control follows DCTCP [10] where ECN mark is in the UD header. When a scattering is sent by the application, it is stored in a send buffer. If the send buffer is full, the send API returns fail. Each destination maintains a send window, which is the minimum of the receive and congestion windows. When all messages of a scattering in send buffer are within the send window for the corresponding destination, they are attached with the current timestamp and sent out. This means that when some destinations or network paths of a scattering are congested, it is held back in the send buffer rather than slowing down the entire network. To avoid live-locks, a scattering acquires "credits" from the send windows. If the send window for a destination is insufficient, the scattering is held in a wait queue without releasing credits. This makes sure that large scatterings can eventually be sent, at the cost of wasting credits that could be used to send other scatterings out-of-order. Beacon packets are sent to the ToR switch, and they are not blocked by flow control.

A UD packet in 1Pipe adds 24 bytes of headers: 3 timestamps including message, best-effort barrier, and commit barrier; PSN;

an opcode and a flag that marks end of message. A timestamp is a 48-bit integer, indicating the number of nanoseconds passed on the host. We use PAWS [50] to handle the timestamp wrap around.

When *lib1pipe* initializes, it registers to the controller and spawns a polling thread to: (1) generate periodic beacon packets; (2) poll RDMA completion queues and process received packets, including generating end-to-end ACKs and retransmitting lost packets; (3) reorder messages in the receive buffer and deliver them to application threads. *lib1pipe* uses polling rather than interrupt because RDMA RTT is only $1 \sim 2\mu$s, while interrupt would add $\sim 10\mu$s of delay [97].

Processes within a host are exposed directly to the Top-of-Rack (ToR) switch(es), and the ToR aggregates timestamps of all processes in the rack. In future work, the software process of *lib1pipe* may be offloaded to a programmable NIC [37, 59], which assigns timestamps to messages on egress pipeline.

The controller is a replicated service that stores routing graph, process information, failure notifications, and undeliverable recall messages in etcd [5]. In a large network, future work can distribute the controller to a cluster, each of which serves a portion of the network.

6.2 In-Network Processing

We implement in-network processing at three types of the network switches with different programming capabilities.

6.2.1 Programmable Switching Chip. We implement the in network processing using P4 [20] and compile it to Tofino [45]. Because a Tofino switch has 4 pipelines, it is regarded as 4 ingress and 4 egress switches connected via 16 all-to-all links. Each of the 8 "switches" derives barriers independently. 1Pipe needs 2 state registers *per input link*, storing the two barriers for best effort and reliable 1Pipe, respectively. For each packet, barrier register of the input link is updated in the first stateful pipeline stage of the switch. Because each stage can only compute the minimum of two barriers, the switch uses a binary tree of registers with $O(\log N)$ pipeline stages to compute the minimum link barrier B_{new}, where N is the number of ports. For a typical 32 port switch, it would cost 5 stages out of the 16 ingress stages. At the final pipeline stage, the barrier field in packet is updated to B_{new}. The control plane software routinely checks link barriers and reports failure if a link barrier significantly lags behind. The expected delay of best effort 1Pipe is (base delay + clock skew) when links are fully utilized, or (base delay + beacon interval/2 + clock skew) when most links are idle.

6.2.2 Switch CPU. For a switch without a programmable switching chip, *e.g.,* Arista 7060 which uses Broadcom Tomahawk chip, we implement in-network processing on the switch CPU. Although commodity switches cannot process packets in data plane, they have a CPU to process control-plane packets, analogous to directly connecting a server to a port of the switch. Compared to server CPUs and NICs, the switch CPU is typically less powerful (*e.g.,* 4 cores at 1 GHz) and has lower bandwidth (*e.g.,* 1 Gbps). Because the switch CPU cannot process every packet, data packets are forwarded by the switching chip directly. The CPU sends beacons periodically on each output link, regardless of whether the link is idle or busy. Received barriers in beacons are stored in registers per input link. A thread on the CPU periodically computes the minimum of link barriers and broadcasts new beacons to all output links. Computing the minimum barrier takes hundreds of cycles, which is not a bottleneck compared to the cost of broadcast. Because data and beacon packets are FIFO in switch queues and on network links, the barrier property is preserved. On receivers, buffered data packets are delivered to the application according to barriers in beacon packets. Compared with the programmable chip, switch CPU has higher latency due to CPU processing. So, the expected delay is base delay + (switch CPU processing delay × number of hops + beacon interval/2 + clock skew).

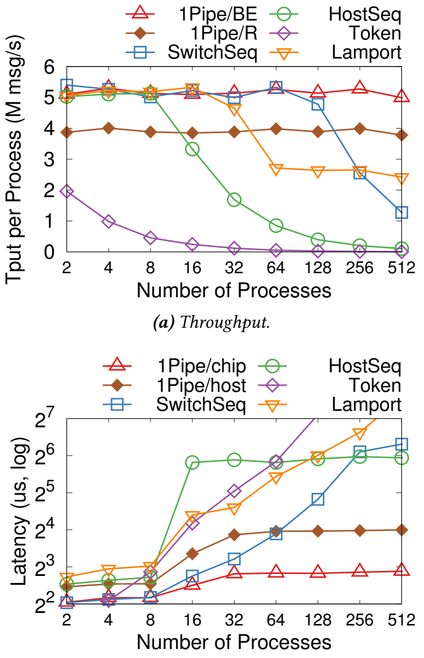
6.2.3 Delegate Switch Processing to a Host. If the switch vendor does not expose access interfaces to switch CPUs, we can offload the beacon processing to end hosts. We designate an *end-host representative* for each network switch. The challenge is that best effort 1Pipe requires the barrier timestamp to be the lower bound of future message timestamps on a network link L. So, beacons with barrier timestamps on L must pass through L. That is, for two directly connected switches S_1, S_2 and their representatives H_1, H_2, beacon packets from H_1 to H_2 need to go through the link $S_1 \longrightarrow S_2$. If the routing path between two representatives does not go through $S_1 \longrightarrow S_2$, beacon packets needs to detour: they are sent with three layers of IP headers: $H_1 \longrightarrow S_1$, $S_1 \longrightarrow S_2$, and $S_2 \longrightarrow H_2$. We install tunnel termination rules in each network switch to de-capsulate one layer of IP header, so the beacon packet will traverse through $H_1 \longrightarrow S_1 \longrightarrow S_2 \longrightarrow H_2$.

Beacon packets use one-sided RDMA write to update barriers on representative host. Similar to Sec.6.2.2, minimum barriers are periodically computed and broadcast to downstream representatives. The expected delay is base delay + ((RTT between switch and host + host processing delay) × number of hops + beacon interval/2 + clock skew). Because CPUs on end hosts may have shorter processing delay (via RDMA) than switch CPU (via OS IP stack), host delegation may have shorter overall delay. This is why we use host delegation for evaluations in Sec.7.

7 EVALUATION

7.1 Methodology

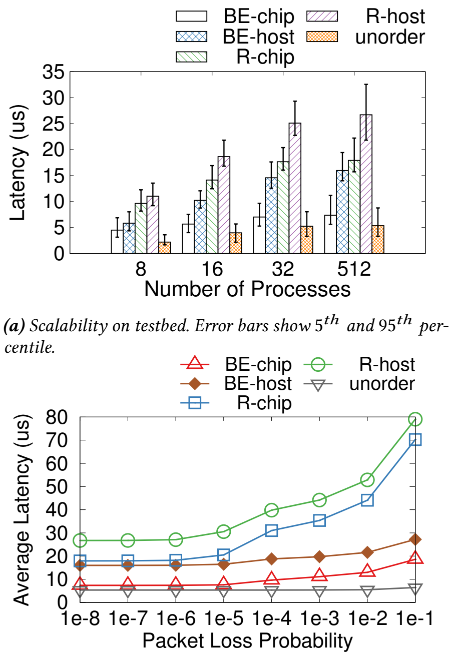
Our testbed has 10 Arista 7060CX-32S 100G switches [1] and 32 servers, forming a 3-layer fat-tree topology (4 ToR, 4 Spine, and 2 Core) similar to Figure 3. The network has no oversubscription because our traffic pattern is all-to-all. Each server has 2 Xeon E5-2650 v2 CPUs and a Mellanox ConnectX-4 NIC running RoCEv2 [12]. We dedicate a CPU core as representative of each switch and NIC to process beacons (Sec.6.2.3). The host representative is directly connected to the switch, so beacon packets do not need to detour. For microbenchmarks in Sec.7.2, we use Tofino [45] switches in place of Arista switches. In small-scale experiments (1~32 processes), each process runs on a distinct server. Each process uses a pair of threads for sending and receiving, respectively. With less or equal to 8 servers, they colocate in one rack. With 16 servers, they are in two racks in a row. For experiments with 64~512 processes, each server hosts the same number of processes. Clocks are synchronized via PTP [28] every 125 ms, achieving an average clock skew



(a) Throughput.

(b) Latency.

***Figure 8:** Scalability comparison of total order broadcast algorithms.*



(a) Scalability on testbed. Error bars show 5^{th} and 95^{th} percentile.

(b) Simulation of varying packet loss rates.

***Figure 9:** Message delivery latency of 1Pipe variants.*

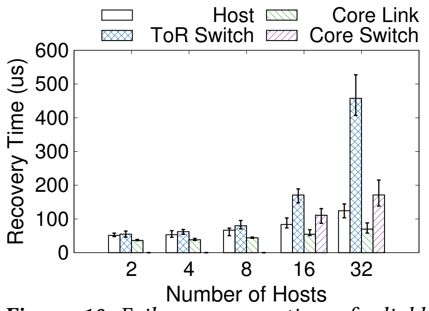


***Figure 10:** Failure recovery time of reliable 1Pipe. Error bars show 5^{th} and 95^{th} percentile.*

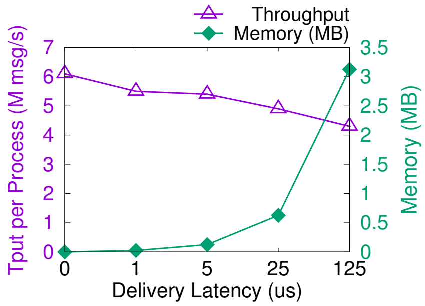


***Figure 11:** Reorder overhead on a host.*

of 0.3 μs (1.0 μs at 95% percentile), which agrees with Mellanox's whitepaper [6]. We choose beacon interval to be 3 μs.

7.2 Microbenchmarks

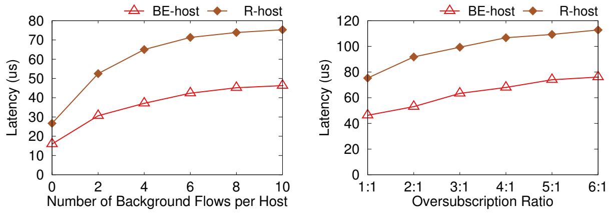
Scalability. 1Pipe can achieve total order broadcast. Figure 8a compares the scalability of 1Pipe with other total order broadcast approaches using token [86], Lamport timestamp [63], and centralized sequencer at host NIC [57] or programmable switch [51]. We test an all-to-all traffic pattern where each process broadcasts 64-byte messages to all processes. 1Pipe scales linearly to 512 processes, achieving 5M messages per second per process (*i.e.*, 80M msg/s per host). The throughput of 1Pipe is limited by CPU processing and RDMA messaging rate. Reliable 1Pipe (1Pipe/R) has 25% lower throughput than best effort 1Pipe (1Pipe/BE) due to 2PC overhead. Programmable chip and host representative (not shown) deliver the same high throughput because 1Pipe decouples message forwarding from barrier propagation.

In contrast, the sequencer is a central bottleneck and introduces extra network delays to detour packets (Figure 8b). The latency soars when throughput of sequencer saturates and congestion appears. Token has low throughput because only one process may send at any time. We apply a common optimization to Lamport timestamp [63] which exchanges received timestamps per interval rather than per message. It has a trade-off between latency and throughput, *e.g.*, for 512 processes, even if 50% throughput is used for timestamp exchange, broadcasting the messages takes 200 μs.

Message delivery latency. Figure 9a shows the message delivery latency of 1Pipe when the system is idle and thus has zero queuing delay. 1Pipe/BE with programmable chip delivers the lowest latency overhead compared to the unordered baseline. The average overhead (1.7~2.3μs) is almost constant with different number of the network layers and processes, which is half of the beacon interval plus average clock skew. The tail latency overhead (1.7~3.3μs) is half of the beacon interval plus maximum clock skew. End host representatives introduce extra forwarding delay from the switch to the end host, which is ~ 2μs and contributes 10μs for the 5-hop topology. In our testbed, ≤8, 16, and ≥32 processes have 1, 3, and 5 network hops, respectively. Reliable 1Pipe adds an RTT (2 ~ 10μs) to best effort 1Pipe due to Prepare phase in 2PC. The RTT and host forwarding delay are proportional to network hop count.

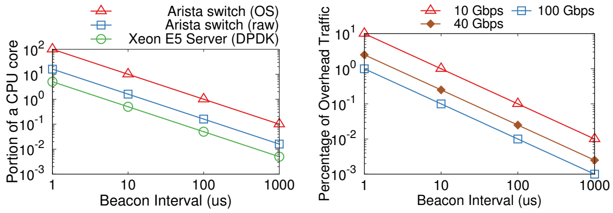
As Sec.2.1 discussed, packet loss rate of links are typically lower than 10^{-8}, but faulty links may have loss rates above 10^{-6}. In Figure 9b, we simulate random message drop in *lib1pipe* receiver to evaluate how packet loss affects latency in the testbed with 512 processes. When loss rate is higher than 10^{-5}, latency of 1Pipe starts to grow. In both BE- and R-1Pipe, a lost beacon packet on any link will stall delivery of barrier until the next beacon, and all receivers need to wait for the worst link. In R-1Pipe, a lost message in prepare phase will trigger retransmission, which will stall the network for an RTT (and possibly multiple RTTs if retransmissions are lost). So, R-1Pipe is more sensitive to packet loss. Packet loss has little impact on throughput because 1Pipe can transfer new messages while retransmitting lost packets.

Besides packet loss, queuing delay caused by background traffic can also increase 1Pipe latency. As Figure 12a shows, with 10 background TCP flows per host, the latency inflation of BE-1Pipe and R-1Pipe are 30 and 50 μs, respectively. A higher oversubscription ratio would also increase latency due to increased buffering at the core of the network. In Figure 12b, we increase oversubscription ratio of the network, and the delay increases due to congestion and



(a) Background flows. *(b) Oversubscription.*

***Figure 12:** The impact of queuing delay on 1Pipe latency.*



(a) CPU processing overhead. *(b) Network bandwidth overhead.*

***Figure 13:** Beacon overhead under different beacon intervals. CPU processing overhead for Arista switch is extrapolated.*

PFC pauses in core network. We believe more advanced congestion control mechanisms [11, 39, 70, 76, 83] can mitigate this problem.

Messages much larger than MTU will stall other messages, *e.g.*, an 1 MB message will increase 80 μs latency of other messages.

Failure recovery. Failure detection in 1Pipe is typically faster than heartbeat timeout in most applications, because the beacon interval in 1Pipe is very low. Figure 10 depicts failure recovery time, which measures the average time of barrier timestamp stall for correct processes. In our testbed, a failure is detected if beacon is not received for 10 beacon intervals (30 μs). In addition to failure detection time, the recovery procedure in Sec.5 requires 6 network diameters plus the time to transmit and process messages. Core link and switch failures do not affect connectivity, so, only the controller needs to be involved, and no process is considered to be failed. Host, NIC, host link, and ToR switch failures cause processes to disconnect from the system, so, the recovery takes longer because each correct process needs to discard messages from or to them. There is a significant jump for ToR switch because all processes in the rack fail, leading to more failure recovery messages.

CPU overhead. The CPU overhead of 1Pipe has two parts: reordering at receivers and beacon processing at switches. The message delivery throughput degrades slightly with more messages to reorder. As Figure 11 shows, the maximal send and receive buffer size increases linearly with latency, but only takes a few megabytes on a 100 Gbps link.

Figure 13a shows the number of cores required for beacon processing of a 32-port switch. A host CPU core can sustain 3 μs beacon interval of the switch, which is our testbed setting. If switch CPUs are used instead, the raw packet processing capacity of a switch CPU is roughly 1/3 of a host CPU core. If we can bypass the kernel network stack and process packets efficiently at Arista switches, a single switch CPU core can sustain 10 μs beacon interval.

Network overhead. As Figure 13b shows, with high link bandwidth and a reasonable beacon interval (*e.g.*, 3 μs), beacon traffic is a tiny portion (*e.g.*, 0.3%) of link bandwidth. Because beacons are hop-by-hop, the overhead is determined by beacon interval and does not increase with system scale.

Scalability to larger networks. On the latency perspective, the base and beacon processing delays are proportional to the number of hops in the network, which is typically logarithm to the number of hosts [8, 43]. The clock skew increases due to higher latency between clock master and hosts, and higher probability of bad clocks with high drift rates [69]. We did not analyze the clock skew quantitatively yet. For reliable 1Pipe, the expected number of packet losses in an RTT is proportional to the number of hosts times the number of hops. For 32K hosts, if all links are healthy (with loss rate 10^{-8}), the latency increases by $0 \sim 3\mu s$ compared to loss-free; if all links are sub-healthy (with loss rate 10^{-6}), the latency increases by $3 \sim 17\mu s$. On the throughput perspective, the beacon overhead is unrelated to network scale, while the hosts would use larger memory and more CPU cycles to reorder the messages. The memory size is BDP (Bandwidth-Delay Product), and the reordering time is logarithm to BDP.

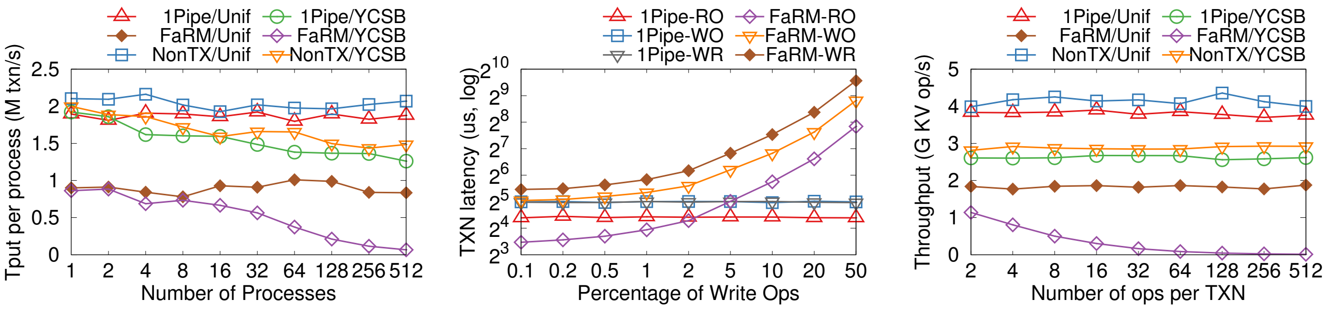
The major scalability challenge is failure handling. Failure of any component will stall the entire network. In best effort 1Pipe, failure handling is localized because the fault component is removed by the neighborhood in a 30 μs-like timeout, so the remaining parts of the network experience a delivery latency inflation. Although this inflated latency is fixed, the frequency of occurrence is proportional to network scale. In reliable 1Pipe, failure handling is coordinated by a centralized controller, which needs to contact all processes in the system, so, the failure recovery delay increases proportionally with system scale, which is 3~15 μs per host.

7.3 Applications

7.3.1 Transactional Key-Value Store. We evaluate a distributed transactional key-value store where each server process stores a portion of KVs in memory using C++ *std::unordered_map* without replication. A transaction (TXN) is composed of multiple independent KV read or write operations. TXN initiators dispatch KV operations to server processes by hash of key. Read-only (RO) TXNs are served by best effort 1Pipe, while read-write (WR) and write-only (WO) TXNs use reliable 1Pipe. For comparison, we implement non-replicated and non-durable FaRM [34] which serves RO TXNs in 1 RTT by reading the KVs and checking lock and version. WR and WO TXNs in FaRM use OCC [62] and two-phase commit. As a theoretical performance upper bound, we also compare with a non-transactional system.

Each TXN has 2 KV ops by default, where read and write are randomly chosen for each op. Keys are 64-bit integers generated either uniformly random or by Zipf distribution in YCSB [26]. YCSB has hot keys. The value size is randomly generated according to Facebook's ETC workload [13]. We record average TXN latency at 95% of peak throughput.

In Figure 14a, 50% of TXNs are read-only. In both uniform and YCSB distribution, 1Pipe delivers scalable throughput (per-process throughput does not degrade), which is 90% of a non-transactional key-value store (NonTX). As number of processes increase, YCSB scales not as linearly as uniform, because contention on hot keys lead to load imbalance of different servers. With 512 processes,



(a) Scalability. *(b) Average latency of YCSB workload.* *(c) Different transaction sizes.*

Figure 14: *Performance of a transactional key-value store.*

YCSB has 70% throughput of uniform both for 1Pipe and NonTX. In uniform workload that is free of contention, FaRM delivers 50% throughput of 1Pipe because WR and WO TXNs need 3 or 4 RTTs. In YCSB workload, FaRM does not scale because it *locks* keys for 2 RTTs during WO/WR TXN commit, and the hot keys block all conflicting TXNs. In contrast, 1Pipe does not lock keys. Each server processes TXNs on the same key in order.

In Figure 14b, we adjust the percentage of write ops and measure latency of RO, WO, and WR TXNs with 512 processes. The latency of 1Pipe is almost constant because servers process read and write ops on the same key in order. WO and WR use reliable 1Pipe, which is slower than RO that uses best effort 1Pipe. For pure RO workload, FaRM has lower latency than 1Pipe because it completes in 1 RTT and does not wait for network-wide reorder. Non-contended FaRM WO and WR consumes 3 and 4 RTTs, respectively, which is slightly worse than 1Pipe. However, with high write percentage, FaRM latency skyrockets due to lock contention on hot keys and TXN aborts due to inconsistent read version.

In Figure 14c, we alter the number of keys per TXN, and measure total KV op/s with 512 processes. 95% of TXNs are read-only. 1Pipe and NonTX are agnostic of TXN size because their throughputs are only bounded by CPU processing and network messaging rate. With a low write percentage (5%), FaRM/YCSB delivers 40% throughput of 1Pipe with 2 KV ops per TXN, but the performance plummets with larger TXN size, because TXN abort rate increases with the number of keys in a TXN.

7.3.2 Independent General Transactions. Now we extend Distributed Atomic Operations (DAO, Sec.2.2.3) to two important classes of distributed transactions: read-only snapshot transactions [27] and independent transactions [51] (or called one-shot transactions [56]) that involve multiple hosts but the input of each host does not depend on the output of other hosts. The two most frequent transactions in TPC-C benchmark (New-Order and Payment) [29] are independent transactions. The major difference between DAO and independent transactions is that the latter often requires replication to ensure durability and fault tolerance. The TXN initiator utilizes the method of Eris [51], which scatters operations to all replicas of all shards in one reliable scattering. So, each TXN can finish in one round-trip (actually two RTTs due to Prepare phase). If a host fails, the other replicas of the same shard reach quorum via traditional consensus.

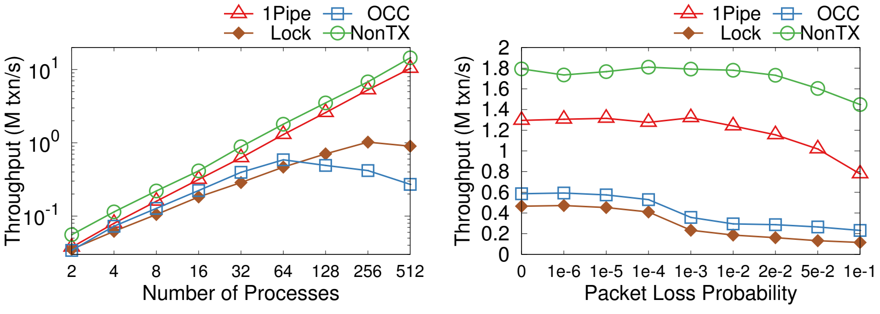
We benchmark New-Order and Payment TXNs in TPC-C [29], which constitute 90% of TPC-C workload. For simplicity, we do not implement non-independent TXNs in TPC-C, which should fall back to traditional concurrency control mechanisms. We use 4 warehouses which are stored in-memory with 3 replicas. Concurrency control and replication are implemented with a scattering of commands sent to all shards and replicas, similar to Eris [51] but replaces its central sequencer with timestamps. We assume TXNs never abort. As shown in Figure 15a, two-phase locking (2PL) and OCC do not scale, because each Payment TXN updates its corresponding warehouse entry and each New-Order reads it [98], leading to 4 hot entries. The throughput of OCC and 2PL reaches peak at 256 and 64 processes, respectively. With more processes, the throughput becomes lower [79]. In contrast, 1Pipe scales linearly with number of processes. With 512 processes, 1Pipe achieves 10.35M TXNs per second, which is 71% of a non-transactional baseline system, 10x of lock and 17x of OCC.

Figure 15b shows TXN throughput under different simulated packet loss rates. We fix process number to be 64. With 1Pipe, although packet loss affects TXN latency (not measured in TPC-C, but should be similar to Figure 9b), the impact on throughput is insignificant. However, in 2PL and OCC commit, a locked object cannot be released until the TXN completes, so, TXN throughput under contention is inversely proportional to TXN latency. TXN latency increases with packet loss rate because replicas wait for the last retransmitted packet to maintain sequential log ordering.

Finally, we evaluate failure recovery of replicas by disconnecting the physical link of a host. 1Pipe detects failure and removes the replica in 181 ± 21μs. The affected TXNs are aborted and retried, with an average delay of 308 ± 122μs. It is much faster than using application heartbeats to detect failures, which takes milliseconds. After the link reconnects, the replica synchronizes log from other replicas in 25 ms.

7.3.3 Remote Data Structures. 1Pipe can remove ordering hazards in remote data structure access (Sec.2.2.1). We implement a distributed concurrent hash table that uses a linked list to store key-value pairs (KVs) in the same hash bucket. The hash table is sharded on 16 servers. Different from Sec.7.3.1 where servers process KV ops, in this section, clients access the remote hash table using RDMA *one-sided* read, write, and CAS. The baseline system uses leader-follower replication. The workload has 16 parallel clients and uniform keys.

As Figure 16 shows, without replication, 1Pipe improves per-client KV insertion throughput to 1.9x because 1Pipe removes the *fence* between writing KV pair and updating the pointer in hash



***(a)** Scalability.*

***(b)** Resilience of packet loss.*

***Figure 15:** TPC-C transaction benchmark.*

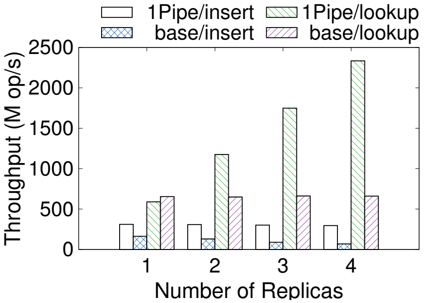


***Figure 16:** Per-client throughput of a replicated remote hash table.*

bucket. KV lookup throughput reduces by 10%, due to additional reordering delay. If the hash table is replicated traditionally, a write op is sent to the leader, which involves CPU software to replicate to followers. With 3 replicas, 1Pipe improves KV insertion throughput to 3.4x. In 1Pipe, all KV operations are ordered by timestamp, so all replicas can serve lookup requests, and the throughput scales with number of replicas. In contrast, with leader-follower replication, to maintain serializability of lookups and updates, only the leader can serve lookups.

7.3.4 Replication in Distributed Storage. We apply 1Pipe to a distributed storage system, Ceph [96]. Ceph OSD uses a primary-backup replication scheme, where the backups are also written sequentially. With 3 replicas, a client waits for 3 disk writes and 6 network messages (3 RTTs) in sequence. Because 1Pipe supports 1-RTT replication in non-failure cases (Sec.2.2.2), the client can write 3 replicas in parallel, thus the end-to-end write latency is reduced to 1 disk write and 1 RTT. Experiment shows that in an idle system with Intel DC S3700 SSDs, the latency of 4KB random write reduces from 160 ± 54μs to 58 ± 28μs (64% reduction).

8 RELATED WORK

Causal and totally ordered communication (CATOCS). Critics [25] and proponents [16, 94] of CATOCS have long discussed the pros and cons of such a primitive. 1Pipe provides a scattering primitive with restricted atomicity, and achieves scalability with in-network computation that incurs little overhead, thus removing two criticisms (can't say together and efficiency). Scattering also enables atomic access to a portion (rather than all) of shards. Although 1Pipe is not a panacea for ordering problems, *e.g.*, it is not sufficient to support serializable general transactions, it shows effectiveness in applications in Sec.7.3.

There has been extensive research on total order broadcast [31]. One line of work leverages logically centralized coordination, *e.g.*, centralized sequencers [18, 51, 54] or a token passed among senders or receivers [36, 60, 78, 86]. As a result, it is challenging to scale the system. Another line of work buffers messages at receivers and builds a causality graph at receivers [63, 84], merges streams deterministically or achieves agreement among receivers [17, 23, 53, 82]. This causes extra delay and network bandwidth overhead. A third line of work assumes a synchronous network [64], but lossy links and bad clocks with high skew violate correctness. In 1Pipe, such components will slow down the whole system but correctness is preserved. In 1Pipe, such components should be detected, and affected hosts can use healthy nodes as proxy, where timestamps are assigned at the proxy.

A fourth line of work, tree-based algorithms, addresses the tradeoff between efficiency and scalability [87]. At each non-leaf node, multiple ordered streams of packets are merged into one ordered stream, called *deterministic merge* [7, 46]. This usage of timestamp and ordering is similar to 1Pipe while applied to other fields such as network switches [49], multi-core processors [55], and sensor networks [22]. However, it is not practical to implement deterministic merge in commodity network switches. First, switches in data centers have small per-port buffer size [14]. Second, commodity switches cannot reorder packets in priority queues according to per-packet metadata [52, 92].

Network and system co-design. Recent years witness a trend of co-designing distributed systems with programmable network switches. Mostly Ordered Multicast [85] and NO-Paxos [68] use a switch as a centralized sequencer or serialization point to achieve totally ordered broadcast. Different from broadcast, 1Pipe provides a scattering primitive and a scalable implementation. Eris [51] proposes in-network concurrency control using switch as a centralized sequencer. NetChain [52], NetLock [99], NetPaxos [30], and DAIET [88] offload important distributed middlewares to programmable switches. Omnisequencing [74] finds out that DCN topology can provide causal (but not total order) message delivery.

9 CONCLUSION

We propose a causal and total order communication abstraction, 1Pipe, that delivers messages in sender's clock time order with restricted failure atomicity. 1Pipe achieves scalability and efficiency by utilizing programmable data center networks to separate aggregating order information from forwarding data packets. 1Pipe can simplify and accelerate many applications, and we expect future work to explore more. One limitation of 1Pipe is that it did not consider Byzantine failure, and we leave the security problems to future work.

ACKNOWLEDGEMENTS

We thank Cheng Li, Tianyi Cui, and Zhenyuan Ruan for the technical discussions. We thank our shepherd, Nathan Bronson, and other anonymous reviewers for their valuable feedback and comments.

REFERENCES

[1] [n. d.]. Arista 7060CX-32 and 7260CX-64. ([n. d.]). https://www.arista.com/en/products/7060x-series.

[2] [n. d.]. Arista EOS. ([n. d.]). https://www.arista.com/en/products/eos.

[3] [n. d.]. Broadcom Tomahawk. ([n. d.]). https://www.broadcom.com/products/ethernet-connectivity/switching/strataxgs/bcm56960-series.

[4] [n. d.]. Cavium XPliant Ethernet switch product family. ([n. d.]). http://www.cavium.com/XPliant-Ethernet-Switch-Product-Family.html.

[5] [n. d.]. etcd: A distributed, reliable key-value store for the most critical data of a distributed system. ([n. d.]). https://etcd.io

[6] 2013. Highly Accurate Time Synchronization with ConnectX-3 and TimeKeeper. (2013). https://www.mellanox.com/related-docs/whitepapers/WP_Highly_Accurate_Time_Synchronization.pdf.

[7] Marcos Kawazoe Aguilera and Robert E Strom. 2000. Efficient atomic broadcast using deterministic merge. In *Proceedings of the nineteenth annual ACM symposium on Principles of distributed computing*. ACM, 209–218.

[8] Mohammad Al-Fares, Alexander Loukissas, and Amin Vahdat. 2008. A Scalable, Commodity Data Center Network Architecture. In *Proceedings of the ACM SIGCOMM 2008 Conference on Data Communication (SIGCOMM '08)*. Association for Computing Machinery, New York, NY, USA, 63–74. https://doi.org/10.1145/1402958.1402967

[9] Mohammad Alizadeh, Tom Edsall, Sarang Dharmapurikar, Ramanan Vaidyanathan, Kevin Chu, Andy Fingerhut, Vinh The Lam, Francis Matus, Rong Pan, Navindra Yadav, et al. 2014. CONGA: Distributed congestion-aware load balancing for datacenters. In *Proceedings of the 2014 ACM conference on SIGCOMM*. 503–514.

[10] Mohammad Alizadeh, Albert Greenberg, David A Maltz, Jitendra Padhye, Parveen Patel, Balaji Prabhakar, Sudipta Sengupta, and Murari Sridharan. 2010. Data center tcp (dctcp). In *Proceedings of the ACM SIGCOMM 2010 conference*. 63–74.

[11] Mohammad Alizadeh, Shuang Yang, Milad Sharif, Sachin Katti, Nick McKeown, Balaji Prabhakar, and Scott Shenker. 2013. pfabric: Minimal near-optimal datacenter transport. *ACM SIGCOMM Computer Communication Review* 43, 4 (2013), 435–446.

[12] Infiniband Trade Association et al. [n. d.]. RoCEv2, September 2014. ([n. d.]).

[13] Berk Atikoglu, Yuehai Xu, Eitan Frachtenberg, Song Jiang, and Mike Paleczny. 2012. Workload analysis of a large-scale key-value store. In *Proceedings of the 12th ACM SIGMETRICS/PERFORMANCE joint international conference on Measurement and Modeling of Computer Systems*. 53–64.

[14] Wei Bai, Kai Chen, Shuihai Hu, Kun Tan, and Yongqiang Xiong. 2017. Congestion Control for High-speed Extremely Shallow-buffered Datacenter Networks. In *Proceedings of the First Asia-Pacific Workshop on Networking*. ACM, 29–35.

[15] Wei Bai, Shuihai Hu, Kai Chen, Kun Tan, and Yongqiang Xiong. 2020. One More Config is Enough: Saving (DC) TCP for High-speed Extremely Shallow-buffered Datacenters. In *IEEE INFOCOM*. 2007.

[16] Ken Birman. 1994. A response to Cheriton and Skeen's criticism of causal and totally ordered communication. *ACM SIGOPS Operating Systems Review* 28, 1 (1994), 11–21.

[17] Ken Birman and Thomas Joseph. 1987. Exploiting virtual synchrony in distributed systems. In *Proceedings of the eleventh ACM Symposium on Operating systems principles*. 123–138.

[18] Kenneth Birman, Andre Schiper, and Pat Stephenson. 1991. Lightweight causal and atomic group multicast. *ACM Transactions on Computer Systems (TOCS)* 9, 3 (1991), 272–314.

[19] Kenneth P Birman, Amr El Abbadi, Wally Dietrich, Thomas A Joseph, and Thomas Raeuchle. 1984. *An overview of the ISIS project.* Technical Report. Cornell University.

[20] Pat Bosshart, Dan Daly, Glen Gibb, Martin Izzard, Nick McKeown, Jennifer Rexford, Cole Schlesinger, Dan Talayco, Amin Vahdat, George Varghese, et al. 2014. P4: Programming protocol-independent packet processors. *ACM SIGCOMM Computer Communication Review* 44, 3 (2014), 87–95.

[21] Jiaxin Cao, Rui Xia, Pengkun Yang, Chuanxiong Guo, Guohan Lu, Lihua Yuan, Yixin Zheng, Haitao Wu, Yongqiang Xiong, and Dave Maltz. 2013. Per-packet load-balanced, low-latency routing for clos-based data center networks. In *Proceedings of the ninth ACM conference on Emerging networking experiments and technologies*. 49–60.

[22] Suchetana Chakraborty, Sandip Chakraborty, Sukumar Nandi, and Sushanta Karmakar. 2011. A reliable and total order tree based broadcast in wireless sensor network. In *Computer and Communication Technology (ICCCT), 2011 2nd International Conference on*. IEEE, 618–623.

[23] Tushar Deepak Chandra and Sam Toueg. 1996. Unreliable failure detectors for reliable distributed systems. *Journal of the ACM (JACM)* 43, 2 (1996), 225–267.

[24] K Mani Chandy and Leslie Lamport. 1985. Distributed snapshots: Determining global states of distributed systems. *ACM Transactions on Computer Systems (TOCS)* 3, 1 (1985), 63–75.

[25] David R Cheriton and Dale Skeen. 1994. *Understanding the limitations of causally and totally ordered communication*. Vol. 27. ACM.

[26] Brian F Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. 2010. Benchmarking cloud serving systems with YCSB. In *Proceedings of the 1st ACM symposium on Cloud computing*. ACM, 143–154.

[27] James C Corbett, Jeffrey Dean, Michael Epstein, Andrew Fikes, Christopher Frost, Jeffrey John Furman, Sanjay Ghemawat, Andrey Gubarev, Christopher Heiser, Peter Hochschild, et al. 2013. Spanner: Google's globally distributed database. *ACM Transactions on Computer Systems (TOCS)* 31, 3 (2013), 8.

[28] Kendall Correll, Nick Barendt, and Michael Branicky. 2005. Design considerations for software only implementations of the IEEE 1588 precision time protocol. In *Conference on IEEE*, Vol. 1588. 11–15.

[29] THE TRANSACTION PROCESSING COUNCIL. [n. d.]. TPC-C Benchmark V5. ([n. d.]). http://www.tpc.org/tpcc/

[30] Huynh Tu Dang, Daniele Sciascia, Marco Canini, Fernando Pedone, and Robert Soulé. 2015. Netpaxos: Consensus at network speed. In *Proceedings of the 1st ACM SIGCOMM Symposium on Software Defined Networking Research*. 1–7.

[31] Xavier Défago, André Schiper, and Péter Urbán. 2004. Total order broadcast and multicast algorithms: Taxonomy and survey. *ACM Computing Surveys (CSUR)* 36, 4 (2004), 372–421.

[32] Akon Dey, Alan Fekete, Raghunath Nambiar, and Uwe Rohm. 2014. YCSB+ T: Benchmarking web-scale transactional databases. In *Data Engineering Workshops (ICDEW), 2014 IEEE 30th International Conference on*. IEEE, 223–230.

[33] Advait Dixit, Pawan Prakash, Y Charlie Hu, and Ramana Rao Kompella. 2013. On the impact of packet spraying in data center networks. In *2013 Proceedings IEEE INFOCOM*. IEEE, 2130–2138.

[34] Aleksandar Dragojević, Dushyanth Narayanan, Orion Hodson, and Miguel Castro. 2014. FaRM: Fast remote memory. In *Proceedings of the 11th USENIX Conference on Networked Systems Design and Implementation*. 401–414.

[35] Aleksandar Dragojević, Dushyanth Narayanan, Edmund B Nightingale, Matthew Renzelmann, Alex Shamis, Anirudh Badam, and Miguel Castro. 2015. No compromises: distributed transactions with consistency, availability, and performance. In *Proceedings of the 25th symposium on operating systems principles*. ACM, 54–70.

[36] Richard Ekwall, André Schiper, and Péter Urbán. 2004. Token-based atomic broadcast using unreliable failure detectors. In *Reliable Distributed Systems, 2004. Proceedings of the 23rd IEEE International Symposium on*. IEEE, 52–65.

[37] Daniel Firestone, Andrew Putnam, Sambhrama Mundkur, Derek Chiou, Alireza Dabagh, Mike Andrewartha, Hari Angepat, Vivek Bhanu, Adrian Caulfield, Eric Chung, et al. 2018. Azure accelerated networking: Smartnics in the public cloud. In *15th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 18)*. 51–66.

[38] Michael J Fischer, Nancy A Lynch, and Michael S Paterson. 1985. Impossibility of distributed consensus with one faulty process. *Journal of the ACM (JACM)* 32, 2 (1985), 374–382.

[39] Peter X Gao, Akshay Narayan, Gautam Kumar, Rachit Agarwal, Sylvia Ratnasamy, and Scott Shenker. 2015. phost: Distributed near-optimal datacenter transport over commodity network fabric. In *Proceedings of the 11th ACM Conference on Emerging Networking Experiments and Technologies*. 1–12.

[40] Yilong Geng, Shiyu Liu, Zi Yin, Ashish Naik, Balaji Prabhakar, Mendel Rosenblum, and Amin Vahdat. 2018. Exploiting a Natural Network Effect for Scalable, Fine-grained Clock Synchronization. In *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*. USENIX Association, 81–94.

[41] Kourosh Gharachorloo, Daniel Lenoski, James Laudon, Phillip Gibbons, Anoop Gupta, and John Hennessy. 1990. *Memory consistency and event ordering in scalable shared-memory multiprocessors*. Vol. 18. ACM.

[42] Albert Greenberg. 2015. SDN for the Cloud. In *Keynote in the 2015 ACM Conference on Special Interest Group on Data Communication*.

[43] Albert Greenberg, James R Hamilton, Navendu Jain, Srikanth Kandula, Changhoon Kim, Parantap Lahiri, David A Maltz, Parveen Patel, and Sudipta Sengupta. 2009. VL2: a scalable and flexible data center network. In *ACM SIGCOMM computer communication review*, Vol. 39. ACM, 51–62.

[44] Chuanxiong Guo, Lihua Yuan, Dong Xiang, Yingnong Dang, Ray Huang, Dave Maltz, Zhaoyi Liu, Vin Wang, Bin Pang, Hua Chen, et al. 2015. Pingmesh: A large-scale system for data center network latency measurement and analysis. *ACM SIGCOMM Computer Communication Review* 45, 4 (2015), 139–152.

[45] Vladimir Gurevich. 2017. Programmable Data Plane at Terabit Speeds. *P4 Workshop* (2017).

[46] Vassos Hadzilacos and Sam Toueg. 1994. *A modular approach to fault-tolerant broadcasts and related problems.* Technical Report. Cornell University.

[47] Keqiang He, Eric Rozner, Kanak Agarwal, Wes Felter, John Carter, and Aditya Akella. 2015. Presto: Edge-based load balancing for fast datacenter networks. *ACM SIGCOMM Computer Communication Review* 45, 4 (2015), 465–478.

[48] Shuihai Hu, Wei Bai, Gaoxiong Zeng, Zilong Wang, Baochen Qiao, Kai Chen, Kun Tan, and Yi Wang. 2020. Aeolus: A Building Block for Proactive Transport in Datacenters. In *Proceedings of the Annual conference of the ACM Special Interest Group on Data Communication on the applications, technologies, architectures, and protocols for computer communication*. 422–434.

[49] Ofer Iny. 2005. Method and system for switching packets. *US Patent No. US7525995B2* (2005).

[50] Van Jacobson, Robert Braden, and David Borman. 1992. TCP extensions for high performance. (1992).

[51] Ellis Michael Jialin Li and Dan R. K. Ports. 2017. Eris: Coordination-Free Consistent Transactions Using In-Network Concurrency Control. In *Proceedings of the 26th Symposium on Operating Systems Principles*. ACM.

[52] Xin Jin, Xiaozhou Li, Haoyu Zhang, Nate Foster, Jeongkeun Lee, Robert Soulé, Changhoon Kim, and Ion Stoica. 2018. NetChain: Scale-Free Sub-RTT Coordination. In *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*. USENIX Association, 35–49.

[53] Flavio P Junqueira, Benjamin C Reed, and Marco Serafini. 2011. Zab: High-performance broadcast for primary-backup systems. In *2011 IEEE/IFIP 41st International Conference on Dependable Systems & Networks (DSN)*. IEEE, 245–256.

[54] M Frans Kaashoek and Andrew S Tanenbaum. 1996. An evaluation of the Amoeba group communication system. In *Proceedings of 16th International Conference on Distributed Computing Systems*. IEEE, 436–447.

[55] Stefan Kaestle, Reto Achermann, Roni Haecki, Moritz Hoffmann, Sabela Ramos, and Timothy Roscoe. 2016. Machine-Aware Atomic Broadcast Trees for Multicores.. In *OSDI*, Vol. 16. 33–48.

[56] Robert Kallman, Hideaki Kimura, Jonathan Natkins, Andrew Pavlo, Alexander Rasin, Stanley Zdonik, Evan PC Jones, Samuel Madden, Michael Stonebraker, Yang Zhang, et al. 2008. H-store: a high-performance, distributed main memory transaction processing system. *Proceedings of the VLDB Endowment* 1, 2 (2008), 1496–1499.

[57] Anuj Kalia Michael Kaminsky and David G Andersen. 2016. Design guidelines for high performance RDMA systems. In *2016 USENIX Annual Technical Conference*. 437.

[58] Naga Katta, Mukesh Hira, Changhoon Kim, Anirudh Sivaraman, and Jennifer Rexford. 2016. Hula: Scalable load balancing using programmable data planes. In *Proceedings of the Symposium on SDN Research*. 1–12.

[59] Antoine Kaufmann, SImon Peter, Naveen Kr Sharma, Thomas Anderson, and Arvind Krishnamurthy. 2016. High performance packet processing with flexnic. In *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems*. 67–81.

[60] Jongsung Kim and Cheeha Kim. 1997. A total ordering protocol using a dynamic token-passing scheme. *Distributed Systems Engineering* 4, 2 (1997), 87.

[61] Gautam Kumar, Nandita Dukkipati, Keon Jang, Hassan MG Wassel, Xian Wu, Behnam Montazeri, Yaogong Wang, Kevin Springborn, Christopher Alfeld, Michael Ryan, et al. 2020. Swift: Delay is Simple and Effective for Congestion Control in the Datacenter. In *Proceedings of the Annual conference of the ACM Special Interest Group on Data Communication on the applications, technologies, architectures, and protocols for computer communication*. 514–528.

[62] Hsiang-Tsung Kung and John T Robinson. 1981. On optimistic methods for concurrency control. *ACM Transactions on Database Systems (TODS)* 6, 2 (1981), 213–226.

[63] Leslie Lamport. 1978. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM* 21, 7 (1978), 558–565.

[64] Leslie Lamport. 1984. Using time instead of timeout for fault-tolerant distributed systems. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 6, 2 (1984), 254–280.

[65] Leslie Lamport. 1998. The part-time parliament. *ACM Transactions on Computer Systems (TOCS)* 16, 2 (1998), 133–169.

[66] Ki Suh Lee, Han Wang, Vishal Shrivastav, and Hakim Weatherspoon. 2016. Globally synchronized time via datacenter networks. In *Proceedings of the 2016 conference on ACM SIGCOMM 2016 Conference*. ACM, 454–467.

[67] Charles E Leiserson. 1985. Fat-trees: universal networks for hardware-efficient supercomputing. *IEEE transactions on Computers* 100, 10 (1985), 892–901.

[68] Jialin Li, Ellis Michael, Naveen Kr Sharma, Adriana Szekeres, and Dan RK Ports. 2016. Just Say NO to Paxos Overhead: Replacing Consensus with Network Ordering.. In *OSDI*. 467–483.

[69] Yuliang Li, Gautam Kumar, Hema Hariharan, Hassan Wassel, Peter Hochschild, Dave Platt, Simon Sabato, Minlan Yu, Nandita Dukkipati, Prashant Chandra, et al. 2020. Sundial: Fault-tolerant Clock Synchronization for Datacenters. In *14th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 20)*. 1171–1186.

[70] Yuliang Li, Rui Miao, Hongqiang Harry Liu, Yan Zhuang, Fei Feng, Lingbo Tang, Zheng Cao, Ming Zhang, Frank Kelly, Mohammad Alizadeh, et al. 2019. HPCC: high precision congestion control. In *Proceedings of the ACM Special Interest Group on Data Communication*. 44–58.

[71] Guohan Lu, Chuanxiong Guo, Yulong Li, Zhiqiang Zhou, Tong Yuan, Haitao Wu, Yongqiang Xiong, Rui Gao, and Yongguang Zhang. 2011. ServerSwitch: A Programmable and High Performance Platform for Data Center Networks.. In *Nsdi*, Vol. 11. 2–2.

[72] Haonan Lu, Christopher Hodsdon, Khiem Ngo, Shuai Mu, and Wyatt Lloyd. 2016. The SNOW Theorem and Latency-Optimal Read-Only Transactions.. In *OSDI*. 135–150.

[73] Haonan Lu, Siddhartha Sen, and Wyatt Lloyd. 2020. Performance-Optimal Read-Only Transactions. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*. 333–349.

[74] Ellis Michael and Dan RK Ports. 2018. Towards causal datacenter networks. In *Proceedings of the 5th Workshop on the Principles and Practice of Consistency for Distributed Data*. 1–4.

[75] Radhika Mittal, Nandita Dukkipati, Emily Blem, Hassan Wassel, Monia Ghobadi, Amin Vahdat, Yaogong Wang, David Wetherall, David Zats, et al. 2015. TIMELY: RTT-based congestion control for the datacenter. In *ACM SIGCOMM Computer Communication Review*, Vol. 45. ACM, 537–550.

[76] Radhika Mittal, Alexander Shpiner, Aurojit Panda, Eitan Zahavi, Arvind Krishnamurthy, Sylvia Ratnasamy, and Scott Shenker. 2018. Revisiting network support for RDMA. In *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication*. 313–326.

[77] Adam Morrison and Yehuda Afek. 2013. Fast concurrent queues for x86 processors. In *ACM SIGPLAN Notices*, Vol. 48. ACM, 103–112.

[78] Louise E. Moser, P Michael Melliar-Smith, Deborah A. Agarwal, Ravi K. Budhia, and Colleen A. Lingley-Papadopoulos. 1996. Totem: A fault-tolerant multicast group communication system. *Commun. ACM* 39, 4 (1996), 54–63.

[79] Shuai Mu, Yang Cui, Yang Zhang, Wyatt Lloyd, and Jinyang Li. 2014. Extracting more concurrency from distributed transactions. In *11th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 14)*. 479–494.

[80] Diego Ongaro and John K Ousterhout. 2014. In search of an understandable consensus algorithm.. In *USENIX Annual Technical Conference*. 305–319.

[81] Seo Jin Park and John Ousterhout. 2019. Exploiting commutativity for practical fast replication. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI '19)*. 47–64.

[82] Fernando Pedone and André Schiper. 1998. Optimistic atomic broadcast. In *International Symposium on Distributed Computing*. Springer, 318–332.

[83] Jonathan Perry, Amy Ousterhout, Hari Balakrishnan, Devavrat Shah, and Hans Fugal. 2015. Fastpass: A centralized zero-queue datacenter network. *ACM SIGCOMM Computer Communication Review* 44, 4 (2015), 307–318.

[84] Larry L Peterson, Nick C Buchholz, and Richard D Schlichting. 1989. Preserving and using context information in interprocess communication. *ACM Transactions on Computer Systems (TOCS)* 7, 3 (1989), 217–246.

[85] Dan RK Ports, Jialin Li, Vincent Liu, Naveen Kr Sharma, and Arvind Krishnamurthy. 2015. Designing Distributed Systems Using Approximate Synchrony in Data Center Networks.. In *NSDI*. 43–57.

[86] B Rajagopalan and Philip K McKinley. 1989. A token-based protocol for reliable, ordered multicast communication. In *Reliable Distributed Systems, 1989., Proceedings of the Eighth Symposium on*. IEEE, 84–93.

[87] Luis Rodrigues, Rachid Guerraoui, and André Schiper. 1998. Scalable atomic multicast. In *Computer Communications and Networks, 1998. Proceedings. 7th International Conference on*. IEEE, 840–847.

[88] Amedeo Sapio, Ibrahim Abdelaziz, Abdulla Aldilaijan, Marco Canini, and Panos Kalnis. 2017. In-network computation is a dumb idea whose time has come. In *Proceedings of the 16th ACM Workshop on Hot Topics in Networks*. 150–156.

[89] Peter Sewell, Susmit Sarkar, Scott Owens, Francesco Zappa Nardelli, and Magnus O Myreen. 2010. x86-TSO: a rigorous and usable programmer's model for x86 multiprocessors. *Commun. ACM* 53, 7 (2010), 89–97.

[90] Alex Shamis, Matthew Renzelmann, Stanko Novakovic, Georgios Chatzopoulos, Aleksandar Dragojević, Dushyanth Narayanan, and Miguel Castro. 2019. Fast general distributed transactions with opacity. In *Proceedings of the 2019 International Conference on Management of Data*. 433–448.

[91] Arjun Singh, Joon Ong, Amit Agarwal, Glen Anderson, Ashby Armistead, Roy Bannon, Seb Boving, Gaurav Desai, Bob Felderman, Paulie Germano, et al. 2015. Jupiter rising: A decade of clos topologies and centralized control in google's datacenter network. *ACM SIGCOMM computer communication review* 45, 4 (2015), 183–197.

[92] Anirudh Sivaraman, Suvinay Subramanian, Mohammad Alizadeh, Sharad Chole, Shang-Tse Chuang, Anurag Agrawal, Hari Balakrishnan, Tom Edsall, Sachin Katti, and Nick McKeown. 2016. Programmable Packet Scheduling at Line Rate. In *Proceedings of the 2016 Conference on ACM SIGCOMM 2016 Conference*. ACM, 44–57.

[93] Joseph Tassarotti, Derek Dreyer, and Viktor Vafeiadis. 2015. Verifying read-copy-update in a logic for weak memory. In *ACM SIGPLAN Notices*, Vol. 50. ACM, 110–120.

[94] Robbert van Renesse. 1994. Why bother with CATOCS? *ACM SIGOPS Operating Systems Review* 28, 1 (1994), 22–27.

[95] Erico Vanini, Rong Pan, Mohammad Alizadeh, Parvin Taheri, and Tom Edsall. 2017. Let it flow: Resilient asymmetric load balancing with flowlet switching. In *14th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 17)*. 407–420.

[96] Sage A Weil, Scott A Brandt, Ethan L Miller, Darrell DE Long, and Carlos Maltzahn. 2006. Ceph: A scalable, high-performance distributed file system. In *Proceedings of the 7th symposium on Operating systems design and implementation*. 307–320.

[97] Jisoo Yang, Dave B Minturn, and Frank Hady. 2012. When poll is better than interrupt.. In *FAST*, Vol. 12. 3–3.

[98] Xiangyao Yu, George Bezerra, Andrew Pavlo, Srinivas Devadas, and Michael Stonebraker. 2014. Staring into the abyss: An evaluation of concurrency control

with one thousand cores. *Proceedings of the VLDB Endowment* 8, 3 (2014), 209–220.

[99] Zhuolong Yu, Yiwen Zhang, Vladimir Braverman, Mosharaf Chowdhury, and Xin Jin. 2020. Netlock: Fast, centralized lock management using programmable switches. In *Proceedings of the Annual conference of the ACM Special Interest Group on Data Communication on the applications, technologies, architectures, and protocols for computer communication.* 126–138.

[100] Hong Zhang, Junxue Zhang, Wei Bai, Kai Chen, and Mosharaf Chowdhury. 2017. Resilient datacenter load balancing in the wild. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication.* 253–266.

[101] Yibo Zhu, Haggai Eran, Daniel Firestone, Chuanxiong Guo, Marina Lipshteyn, Yehonatan Liron, Jitendra Padhye, Shachar Raindel, Mohamad Haj Yahia, and Ming Zhang. 2015. Congestion control for large-scale RDMA deployments. *ACM SIGCOMM Computer Communication Review* 45, 4 (2015), 523–536.

[102] Danyang Zhuo, Monia Ghobadi, Ratul Mahajan, Klaus-Tycho Förster, Arvind Krishnamurthy, and Thomas Anderson. 2017. Understanding and mitigating packet corruption in data center networks. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication.* 362–375.

APPENDIX

Appendices are supporting material that has not been peer-reviewed.

Correctness Analysis. In reliable 1Pipe, a scattering M from sender S to receivers R_i with timestamp T should be delivered if and only if the S does not fail before T and S has received all ACKs from R_i. If S fails before T (*i.e.*, the failure timestamp of S is less than T), then M is not committed (or the commit message has not propagated to any receiver, which is regarded as not committed). If any receiver R_i fails before receiving the prepare message of M or sending the ACK message, S should recall M.

Now, we analyze the behavior of message $M : S \rightarrow R$ under each type of failure. A process has both sender and receiver roles, and they are considered separately.

- Packet loss on a fair-loss link: deliver because 2PC retransmits packets.
- R fails before sending ACK in Prepare phase: discard according to Recall step.
- R fails after sending ACK in Prepare phase: The message is in R's receive buffer. We only ensure atomicity in a fail-recover model because recording the last message that R has delivered is impossible. The controller records failure timestamps of senders and undeliverable recall messages, so that recovered receivers can deliver or discard messages consistently in each scattering. If R fails permanently, atomicity is violated.
- S fails before sending commit timestamp T: discard according to Discard step. For example, when S fails while executing Recall step of M due to previous failure of a receiver, its commit timestamp T must be lower than M's timestamp, so all receivers would discard M due to failure of S.
- S fails after sending commit timestamp T: deliver if and only if the failure timestamp T' of S is larger or equal to T. Failure timestamp is determined by controller, which ensures that S has committed T' but no message after T' is delivered. Controller obtains T' by gathering the maximum failure timestamp from a cut in a routing graph consisting of healthy switches that separates S and all receivers. If the data center has separate production and management networks, and assuming they do not fail simultaneously, such a cut can always be found. Otherwise, a network partition may lead to atomicity violation.
- The receiver R' of another message M' in the same scattering fails before sending ACK in Prepare phase: because the ACK is not received, discard according to Recall step.
- The receiver R' of another message M' in the same scattering fails after sending ACK in Prepare phase: deliver according to 2PC.
- The network path between S and R fails (*e.g.*, due to routing problem), but S and R are reachable from the controller: controller forwards messages between S and R.
- The network path between S and R fails, and S or R is unreachable from the controller: the unreachable process is considered as failed. For example, if a host or the only network link from a host fails, all processes on it are disconnected from the network, so they are considered to fail.
- The network path to R fails after R receives commit barrier T: T is already delivered to R.
- The network path to R fails before R receives commit barrier T: deliver after R recovers according to Receiver Recovery. If R fails permanently, atomicity is violated.