Fast Total Ordering for Modern Data Centers

Amy Babay and Yair Amir
Department of Computer Science at Johns Hopkins University
{babay, yairamir}@cs.jhu.edu

Technical Report CNDS-2016-1 - January 2016
http://www.dsn.jhu.edu

***Abstract*—The performance profile of local area networks has changed over the last decade, but many practical group communication and ordered messaging tools rely on core ideas invented over a decade ago. We present the Accelerated Ring protocol, a novel ordering protocol that improves on the performance of standard token-based protocols by allowing processes to pass the token before they have finished multicasting. This performance improvement is obtained while maintaining the correctness and other beneficial properties of token-based protocols.**

On 1-gigabit networks, a single-threaded daemon-based implementation of the protocol reaches network saturation, and can reduce latency by 45% compared to a standard token-based protocol while simultaneously increasing throughput by 30%. On 10-gigabit networks, the implementation reaches throughputs of 6 Gbps, and can reduce latency by 30-35% while simultaneously increasing throughput by 25-40%. A production implementation of the Accelerated Ring protocol has been adopted as the default ordering protocol for data center environments in Spread, a widely-used open-source group communication system.

I. Introduction

Data center applications rely on messaging services that guarantee reliable, ordered message delivery for a wide range of distributed coordination tasks. Totally ordered multicast, which (informally) guarantees that all processes receive messages in exactly the same order, is particularly useful for maintaining consistent distributed state in systems as diverse as financial systems, distributed storage systems, cloud management, and big data analytics platforms.

Many ordering protocols have been developed to build such messaging services. Défago et al. survey these total ordering protocols and classify them based on their ordering mechanisms [1]. A particularly successful type of ordering protocol is the class of *token-based* protocols.[1] Token-based protocols typically arrange the processes participating in the protocol in a logical ring and order messages using a special control message (called the token) that carries the information needed to order new messages and is passed around the ring. A participant may send messages when it receives the token and is able to assign sequence numbers to its messages before sending them (using the information carried by the token), so messages are ordered at the time they are sent.

[1]Défago et al. split token-based protocols into *moving sequencer* and *privilege-based* protocols. We use the term *token-based* to refer primarily to privilege-based protocols.

Token-based protocols are attractive because of their simplicity; a single mechanism, the token, provides ordering, stability notification, flow control, and fast failure detection. Moreover, these protocols naturally support flexible semantics and continuous operation with well-defined guarantees during network partitions.

Token-based protocols also achieved high network utilization at the time they were introduced; for example, the Totem Ring protocol [2], [3] achieved about 75% network utilization on 10-megabit Ethernet using processors standard for 1995. The simplicity, flexibility, and high performance of token-based protocols led to their use in practical messaging services, including the Spread toolkit [4], the Corosync cluster engine [5], and the Appia communication framework [6], [7].

Non-token-based ordering protocols (e.g. Paxos- or sequencer-based) are also used in practical services, but these approaches often offer subtly different semantics, so the protocols are not entirely interchangeable. We discuss successful systems based on a variety of ordering mechanisms in Section V, commenting on their semantics, as well as their performance in the environments we consider.

Modern data center environments present different networking and processing trade-offs than in the past, creating performance challenges for protocols relying on core ideas invented over a decade ago. When Fast Ethernet replaced 10-megabit Ethernet, network speed increased by a factor of ten, and network span shrank by the same factor (from 2000 to 200 meters). This kept basic network characteristics essentially the same, allowing the same protocols to continue to utilize the network well. However, on networks common in today's data centers, these protocols do not reach the same network utilization as in the past while maintaining reasonable latency.

We were exposed to this problem through our experience with applications using Spread, a widely-used, publicly available group communication toolkit that provides flexible semantics for message delivery and ordering, using a variant of the Totem Ring protocol. Spread reached about 80% network utilization on 100-megabit Fast Ethernet, using processors common for 2004 [8]. However, out-of-the-box, we measured the latest Spread release prior to our work as reaching 50% network utilization on a 1-gigabit network. Careful tuning of the flow control parameters allows Spread to reach 800 Mbps in throughput, but the cost in latency is very high.

One reason that protocols originally designed for 10-megabit Ethernet maintained their high network utilization with low latency on 100-megabit Fast Ethernet but not on 1-gigabit, 10-gigabit, and faster networks, is that these faster networks could not use the same techniques as Fast Ethernet to scale throughput (this would have required a 1-gigabit network span to be limited to 20 meters). Moving to these faster networks required changing the network architecture and adding buffering to switches. This changed networking trade-offs. While throughput increased by a factor of 10, 100, or more and latency was substantially reduced, the decrease in latency was significantly lower than the corresponding improvement in throughput.

This change in the trade-off between a network's throughput and its latency alters the performance profile of token-based protocols, as these protocols are particularly sensitive to latency. The ability to multicast new messages rotates with the token, so no new messages can be sent from the time that one participant finishes multicasting to the time that the next participant receives the token, processes it, and begins sending new messages. Note that we are concerned with latency intrinsic to the network, not with latency due to particular machines running slowly. In production deployments, we can ensure that the ordering middleware has the resources it needs (e.g. with real-time priorities in Linux).

For processors common today, 10-gigabit and faster networks change trade-offs further by making networking considerably faster relative to single-threaded processing. Unlike in the past, this trade-off is not likely to change soon, as the processing speed of a single core is no longer increasing at a rate comparable to the rate of improvement in network throughput, with Moore's law reaching its limit.

The gap between the performance of existing protocols and the performance that is possible in modern environments led us to design the Accelerated Ring protocol. The Accelerated Ring protocol compensates for, and even benefits from, the switch buffering that limits the network utilization of other token-based protocols. Using a simple idea, the Accelerated Ring protocol is able to improve both throughput and latency compared to existing token-based protocols without significantly increasing the complexity or processing cost of the protocol. The Accelerated Ring protocol is still able to take advantage of the token mechanism for ordering, stability notification, flow control, and fast failure detection.

Single-threaded implementations of the protocol are able to saturate 1-gigabit networks and considerably improve performance on 10-gigabit networks without consuming the CPU of more than a single core. Limiting CPU consumption to one core is important for this type of ordering service, as it is intended to serve applications that may be CPU intensive; a service that consumes most of the processing power of the machine is likely to be limiting in many practical deployments.

Like other (privilege-based) token-based protocols, the Accelerated Ring protocol passes a token around a logical ring, and a participant is able to begin multicasting upon receiving the token. The key innovation is that, unlike in other protocols, a participant may release the token before it finishes multicasting. Each participant updates the token to reflect all the messages it will multicast during the current rotation of the token around the ring before beginning to multicast. It can then pass the token to the next participant in the ring at any point during the time it is multicasting. Since the token includes all the necessary information, the next participant can begin multicasting as soon as it receives the token, even if its predecessor on the ring has not yet finished multicasting.

However, the fact that the token can reflect messages that have not yet been sent requires careful handling of other aspects of the protocol. For example, messages cannot be requested for retransmission as soon as the token indicates that their sequence numbers have been assigned, since this may result in many unnecessary retransmissions: a participant may not have received all the messages reflected in the token, not because they were lost, but because they were not yet sent.

Sending the token before all multicasts are completed allows the token to circulate the ring faster, reduces or eliminates periods in which no participant is sending, and allows for controlled parallelism in sending. As a result, the Accelerated Ring protocol can simultaneously provide higher throughput and lower latency than a standard token-based protocol.

We evaluate the Accelerated Ring protocol in a library-based prototype, a daemon-based prototype, and a production implementation in Spread. Spread has existed for many years, serving production systems since 1998. Examples of current use include cloud management, distributed storage products, networking hardware, financial trading systems, and web server coordination. The demands of real applications introduced features that incur significant costs, such as large group names, support for hundreds of clients per daemon, support for a large number of simultaneous groups with different sets of clients, and multi-group multicast (the ability to send a single message to all members of multiple distinct groups, with ordering guarantees across groups). Therefore, in addition to evaluating the protocol in this complete production system, we evaluate the protocol in a simple, library-based prototype to quantify the benefit of the protocol outside of a complex system, with no additional overhead for client communication.

However, part of Spread's success is due to its client-daemon architecture. This architecture provides a clean separation between the middleware and the application, allows a single set of daemons in a data center setup to support several different applications, and allows open group semantics (a process does not need to be a member of a group to send to that group). Because of this, we additionally evaluate the protocol in a daemon-based prototype that does not incur the cost of all Spread's complexity but provides a realistic solution, including client communication, for a single group.

We compare each implementation of the Accelerated Ring protocol to a corresponding implementation of the original Totem Ring protocol. Sending messages with 1350-byte payloads, on 1-gigabit networks, the Spread implementation of the Accelerated Ring protocol reaches network saturation (over

920 Mbps measuring only payload), and can reduce latency by 45% compared to the original protocol, while simultaneously increasing throughput by 45-60%. On 10-gigabit networks, Spread reaches over 2 Gbps, and can improve both throughput and latency by 10-20%. The daemon- and library-based prototypes reach 3.1 Gbps and 4.6 Gbps, respectively, and can simultaneously improve latency by 20-35% and throughput by 25-50%. With 8850-byte payloads, throughput reaches 5.2 Gbps for Spread, 6 Gbps for the daemon-based prototype, and 7.3 Gbps for the library-based prototype.

The contributions of this work are:

1) The invention of the Accelerated Ring protocol, a new ordering protocol that takes advantage of the trade-offs in modern data center environments.
2) A thorough evaluation of the protocol in a library-based prototype, a daemon-based prototype, and Spread on 1-gigabit and 10-gigabit networks.
3) The release of the Accelerated Ring protocol as part of Spread's open-source code, and the adoption of this protocol as Spread's standard ordering protocol for local area networks and data center environments.

II. System and Service Model

The Accelerated Ring protocol provides reliable, totally ordered multicast and tolerates message loss (including token loss), process crashes and recoveries, and network partitions and merges. We assume that Byzantine faults do not occur and messages are not corrupted.

The Accelerated Ring protocol provides Extended Virtual Synchrony (EVS) semantics [9], [10]. EVS extends the Virtual Synchrony (VS) model [11] to partitionable environments. EVS, like VS, provides well-defined guarantees on message delivery and ordering with respect to a series of configurations, where a configuration consists of a set of connected participants that can communicate among themselves but not with participants that are not part of that set, and a unique identifier for the configuration.

The Accelerated Ring protocol provides *Agreed* and *Safe* delivery services. These services are completely and formally specified in [9], [10], but the most relevant properties are:

1) Agreed delivery guarantees that messages delivered within a particular configuration are delivered in the same total order by all members of that configuration. The total order respects causality.
2) Safe delivery guarantees that if a participant delivers a message in some configuration, each other member of the configuration has received that message and will deliver it, unless it crashes. This property is often called *stability*.

The protocol can also provide FIFO and Causal delivery services, but their latency is similar to that of Agreed delivery. Therefore, they are not discussed here.

The complete Accelerated Ring protocol consists of an ordering protocol and a membership algorithm. Since the Accelerated Ring protocol directly uses the membership algorithm of Spread, which is based on the Totem membership algorithm [2], [10], we focus on the ordering protocol, which is novel, in this paper. However, both components are necessary to support the above system model and service semantics.

III. Accelerated Ring Protocol

The following description specifies the normal-case operation of the Accelerated Ring protocol with a static set of participants. We assume that the membership of the ring has been established, and the first regular token has been sent. Crashes, network partitions, and token losses are not discussed here, as they are handled by the membership algorithm.

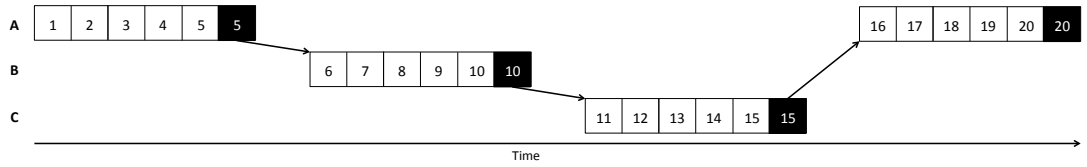
A. Overview

Figure 1 shows an example execution with three participants sending a total of twenty messages in the original Totem Ring protocol and the Accelerated Ring protocol.

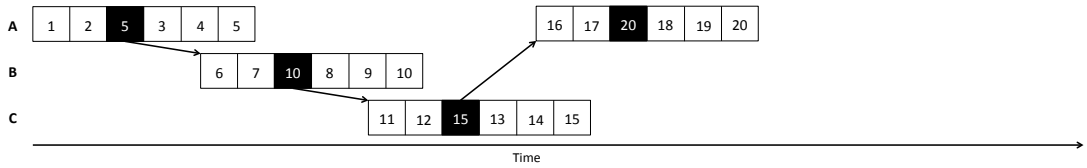
In the original protocol (Figure 1a), each participant sends five messages when it receives the token and then passes the token to the next participant. The token carries the sequence number of the last message that was sent, so the next participant knows exactly which sequence numbers it can assign to its messages. For example, when Participant B receives the token with sequence number 5 from Participant A, it can send message 6, since it knows that no sequence number higher than 5 has been assigned to a message.

In the accelerated protocol (Figure 1b), each participant similarly sends five data messages upon receiving the token, but it sends some of those messages *after* passing the token to the next participant. In this example, each participant sends its first two data messages, then sends the token, and then sends its last three data messages. Note, however, that the token carries exactly the same sequence numbers as in the original protocol. Even though Participant A does not send messages 3, 4, and 5 until after it has passed the token to Participant B, it has already decided exactly which messages it will send. Therefore, the token that Participant B receives still carries the sequence number 5, reflecting all the messages that Participant A will send during the current rotation of the token around the ring (also called a *token round*). Just as in the original protocol, Participant B learns that no sequence number higher than 5 has been assigned to a message and can send messages with sequence numbers starting at 6. Unlike protocols that improve performance by providing a speculative result before reaching a full agreement on ordering like Zyzzyva [12] or ToTo [13], the Accelerated Ring protocol marks each message with its final sequence number at the time it is sent. A participant will not deliver message 6 until it receives all messages 1-5 and can deliver it in the correct order.

To see how the Accelerated Ring protocol simultaneously improves throughput and latency, notice that the accelerated protocol takes less time to complete a token round than the original protocol. Figure 1 clearly shows the first 15 messages are sent and the token is returned to Participant A earlier in the accelerated protocol than in the original protocol. The Accelerated Ring protocol improves throughput by sending the same 15 messages in less time and improves latency by



(a) Original Ring Protocol



(b) Accelerated Ring Protocol

Fig. 1. Example execution with 3 participants sending a total of 20 data messages in the original Ring protocol and the Accelerated Ring protocol. Data messages are shown as white boxes, while token messages are shown as black boxes. The number in each box represents the sequence number on a data message and the sequence field on the token.

getting the token back to Participant A faster, so it waits less time to send message 16. The parallelism that gives us this performance improvement is enabled by the buffering of modern switches discussed in Section I.

To gain an intuition for the correctness of the accelerated protocol, notice that if Participant B receives messages 1 and 2 but not messages 3-5 before receiving the token in the execution of the Accelerated Ring protocol in Figure 1b, then from Participant B's point of view, that execution is exactly like the execution of the original Ring protocol in the case that messages 3-5 are lost, and the original Ring protocol copes with message loss. However, in the original protocol, Participant B would then immediately request messages 3-5 for retransmission. In the accelerated protocol, these messages are most likely in transit, rather than lost, so requesting them would lead to unnecessary retransmissions and reduce performance. Therefore, in the Accelerated Ring protocol, we request missing messages one round after noticing them, guaranteeing that they have already been sent and should have arrived. Thus, the improved throughput and latency of the Accelerated Ring protocol comes at the cost of requiring an extra round to recover lost messages (but each round is faster).

The number of messages a participant can send when it receives the token depends on the *Personal_window* parameter: the maximum number of new data messages a participant can send in one token round. The *Accelerated_window* specifies the maximum number of these messages that can be sent *after* passing the token. In Figure 1b, the *Personal_window* is five and the *Accelerated_window* is three. If a participant has fewer than *Personal_window* messages to send, it still sends as many as possible after the token. If a participant in Figure 1b only had two messages to send, it would send both after the token.

Below we describe the Accelerated Ring protocol in detail. During normal operation, participants take actions in response to receiving token messages and data messages. We describe how these messages are handled in Sections III-B and III-C.

B. Token Handling

Token messages carry control information that is used to establish a correct total order and provide flow control. Token messages contain the following fields:

1) *seq*: Last sequence number assigned to a message. When a participant receives a token, it can multicast new messages with sequence numbers starting at *seq* + 1.
2) *aru* (all-received-up-to): Field used to determine the highest sequence number such that each participant has received every message with a sequence number less than or equal to that sequence number. Messages all participants have received can be delivered with Safe delivery semantics and/or garbage-collected.
3) *fcc* (flow control count): Total number of multicast messages sent during the last token round (including retransmissions). Participants use this field when determining the maximum number of messages they may send in the current round.
4) *rtr* (retransmission requests): List of sequence numbers corresponding to messages that must be retransmitted (because they were lost by some participant).

The steps a participant executes upon receiving a token are:

1) Pre-token multicasting: calculating the total number of messages to send and multicasting messages, including all retransmissions and potentially some new messages (Section III-B1)
2) Updating and sending the token (Section III-B2)
3) Post-token multicasting: multicasting new messages that the participant decided on but did not multicast during pre-token multicasting (Section III-B3)
4) Delivering and discarding messages (Section III-B4)

1) Pre-token Multicasting: In the pre-token multicasting phase, a participant determines the complete set of messages it will multicast during the current token round, including both new data messages and retransmissions. This allows the participant to correctly update the token to reflect all the

messages it will send in the current round. However, the participant does not need to send all of its new messages for the round during this phase.

The participant first answers retransmission requests, retransmitting any messages it has that are requested in the *rtr* field of the token. All retransmissions must be sent during the pre-token phase; otherwise, they may be unnecessarily requested again.

The participant then calculates *Num_to_send*, the total number of new data messages it will multicast in the current round (including both the pre-token and post-token phases). *Num_to_send* is calculated as the minimum of:

- The number of data messages it currently has waiting to be sent.
- *Personal_window*: the maximum number of new data messages a single participant can send in one token round.
- *(Global_window - received_token.fcc - num_retrans)*: the number of new messages a participant can send while respecting the global flow control (i.e. the total number of messages that can be sent in one token round). The *Global_window* is the maximum number of data messages that can be sent in a single token round by all participants combined. *num_retrans* denotes the number of retransmissions this participant sent in this round.

Finally, the participant may multicast some of its new messages. Each new message is stamped with its sequence number (assigned consecutively, starting at *seq* + 1) before it is sent. The number of new messages sent in the pre-token phase depends on the *Accelerated_window*, which specifies the maximum number of messages a participant can send *after* passing the token. The number of new messages a participant sends in the pre-token phase is $Num\_to\_send - Accelerated\_window$. If *Num_to_send* is less than or equal to the *Accelerated_window*, the participant sends no new messages in the pre-token phase.

2) Updating and Sending the Token: Before passing the token to the next participant, the current token holder updates each of the token fields.

The *seq* field is updated by adding *Num_to_send* to the value of the *seq* field on the received token. This sets the *seq* field to the highest sequence number that has been assigned to a message.

The *aru* field is updated according to the rules in [2]. Each participant tracks its local *aru*, which is the highest sequence number such that the participant has received all messages with sequence numbers less than or equal to that sequence number. If the participant's local *aru* is less than the *aru* on the token it receives, it lowers the token *aru* to its local *aru*. If the participant lowered the token *aru* in a previous round, and the received token's *aru* has not changed since the participant lowered it, it sets the token *aru* equal to its local *aru*. If the received token's *aru* and *seq* fields are equal, and the participant does not need to lower the token's *aru*, it adds *Num_to_send* to the *aru* on the token it received.

The *fcc* field is also updated as in [2]. The total number of retransmissions and new messages the participant sent in the previous round are subtracted from the received token's *fcc* value, and the total number of retransmissions and new messages the participant is sending in the current round are added to the received token's *fcc* value.

The *rtr* field is updated to remove retransmission requests that the participant answered in this round and add requests for any messages that this participant has missed. The fact that participants can pass the token before completing their multicasts for the round complicates retransmission requests, since the *seq* field of the token may include messages that have not actually been sent yet. In order to avoid unnecessarily retransmitting messages, the participant only requests retransmissions for missing messages up through the value of the *seq* field on the token it received in the *previous* round, rather than the token it just received.

The updated token is then sent to the next participant.

3) Post-token Multicasting: During this phase, the participant completes its sending for the current round. If *Num_to_send* is less than the *Accelerated_window*, it sent no new messages during the pre-token phase and sends all of these messages (if any) during the post-token phase. Otherwise, it sends *Accelerated_window* data messages during this phase.

4) Delivering and Discarding: As the final step of token handling, the participant uses the procedure in [2] to deliver and discard messages. A participant can deliver a Safe message once it has received and delivered all messages with lower sequence numbers and knows that all other participants have received and will deliver the message (unless they crash). Therefore, the participant delivers all messages with sequence numbers less than or equal to the minimum of the *aru* on the token it sent this round and the *aru* on the token it sent last round. Since every participant had the opportunity to lower the *aru* during the round, every participant must have a local *aru* of at least the minimum of these two values. Since every participant has these messages, they will never be requested for retransmission again and can also be discarded.

C. Data Handling

Data messages carry application data to be transmitted, as well as meta-data used for ordering messages. Each data message contains the following fields:

1) *seq*: Sequence number of this message. This specifies the message's position in the total order.
2) *pid*: ID of the participant that initiated this message.
3) *round*: Token round in which the message was initiated.
4) *payload*: Application data. This is not inspected or used by the protocol.

When a participant receives a data message, it inserts it into its buffer of messages, ordered by sequence number. If the message allows the participant to advance its local *aru* (i.e. the sequence number of the message equals its local *aru + 1*), it delivers its undelivered messages in the order of their sequence numbers until it reaches a sequence number higher than the local *aru* (i.e. a message it has not received) or a message requiring Safe delivery. Messages requiring Safe

delivery cannot be delivered until the token *aru* indicates that they have been received by all participants.

D. Selecting a Message to Handle

In general, token and data messages are handled as they arrive. However, when messages arrive nearly simultaneously, both token and data messages can be available for processing at the same time. The protocol decides which to handle first by assigning logical priorities to message types. When a message type has *high priority*, no messages of the other type will be processed as long as some message of that type is available.

The key issue is that a participant should not process a token until it has processed all the data messages reflected in the last token it processed (or as many as it will receive, if there is loss). If these messages have not yet been processed, but were not lost, they will be unnecessarily requested. However, in order to maximize performance, the token should be processed as soon as possible. Note that decisions about when to process messages of different types can impact performance but do not affect the correctness of the protocol.

A participant always gives data messages high priority after processing a token. However, two different methods may be used for deciding when to raise the token's priority again after processing the token for a given round. In the first method, a participant gives the token high priority as soon as it processes any data message its immediate predecessor in the ring sent in the next token round (indicated by the *round* field of the data message). In the second method, a participant waits to give the token high priority until it processes a data message that its immediate predecessor sent in the next round *after* having sent the token for that round. The first method maximizes the speed of token rotation. The second slows the token slightly to reduce the number of unprocessed data messages that can build up but maintains the acceleration of the token by raising its priority as soon as it is known to have been sent.

E. Implementation Considerations

Our implementations of the Accelerated Ring protocol use IP-multicast to transmit data messages and UDP unicast to pass the token. We use IP-multicast since it is generally available in the local area networks and data center environments for which the protocol is designed, especially when building infrastructure. However, if IP-multicast is not available, unicast can be used to construct logical multicast at the cost of reduced performance; this capability is available as an option in Spread.

For Spread and the daemon-based prototype, daemons communicate with local clients using IPC sockets. Spread also supports remote clients that connect via TCP, but this is not recommended for local area networks, where it is best to co-locate Spread daemons and clients.

In Section III-D, we describe switching priority between token and data messages. In practice, we accomplish this by sending the two message types on different ports and using different sockets for receiving them. Thus, when data messages have high priority, we do not read from the token-receiving socket unless no data message is available, and vice versa.

In the prototypes, we use the first, more aggressive method for giving the token high priority, since this gives the best performance with properly tuned flow-control parameters. However, for Spread we chose the second, less aggressive method, because this method is less sensitive to misconfiguration, which is important for open-source systems used in a wide variety of production environments. When the *Accelerated_window* is zero, the second method behaves like the original Ring protocol, while the first method may still accelerate the token by processing it as early as possible.

IV. Evaluation

We experimentally evaluate the performance profile of the Accelerated Ring protocol and compare it to the performance of the original Totem Ring protocol [2], [3]. We evaluate both protocols in library-based and daemon-based prototypes as well as in complete production implementations in Spread.

A. Benchmarks

All benchmarks use 8 Dell PowerEdge R210 II servers, with Intel Xeon E3-1270v2 3.50 GHz processors, 16 GB of memory, and 1-gigabit and 10-gigabit Ethernet connections. The servers were connected using a 1-gigabit Catalyst 2960 Cisco switch and a 10-gigabit 7100T Arista switch.

To evaluate each protocol, we ran the system at varying throughput levels (measured in clean application data only), ranging from 100 Mbps up to the maximum throughput of the system. At each throughput level, we measured the average latency to deliver a message for both Agreed and Safe delivery.

For Spread and the daemon-based prototype, each of the 8 participating servers ran one daemon, one sending client that injected messages at a fixed rate, and one receiving client. Each sending client sent the same number of messages, and each receiving client received all the messages sent by all sending clients. For the library-based prototype, each server ran one process that injected and received messages.

A broad range of parameter settings provide good performance. *Personal_windows* of a few tens (e.g. 20-40) of messages with *Accelerated_windows* of half to all of the *Personal_window* yield good results in all environments we tested.

For the daemon-based prototype and Spread, we report results with the smallest *Personal_window* and corresponding *Accelerated_window* that let the system reach its maximum throughput. For the library-based prototype, we controlled throughput by adjusting the personal window and having each process send as many messages as it was allowed (while respecting the flow control parameters) each time it received the token; smaller personal windows result in lower throughput (corresponding to the situation where each process has only a small number of new messages to send each time it receives the token).

1) 1-gigabit Experiments: For these experiments, all data messages contained a 1350-byte payload. This size allows the entire message to fit in a single IP packet with a standard 1500-byte MTU, while allowing sufficient space for protocol

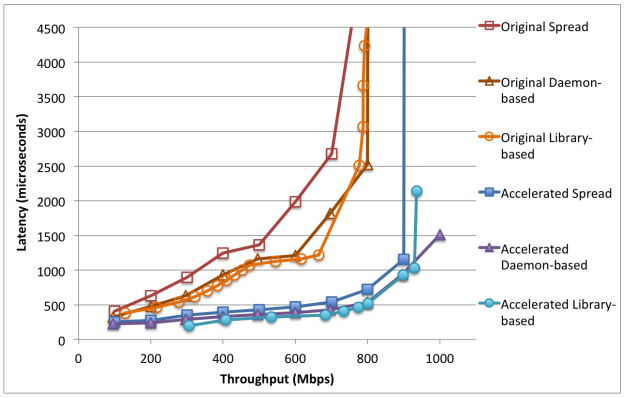


Fig. 2. Agreed delivery latency vs. throughput, 1 Gb network

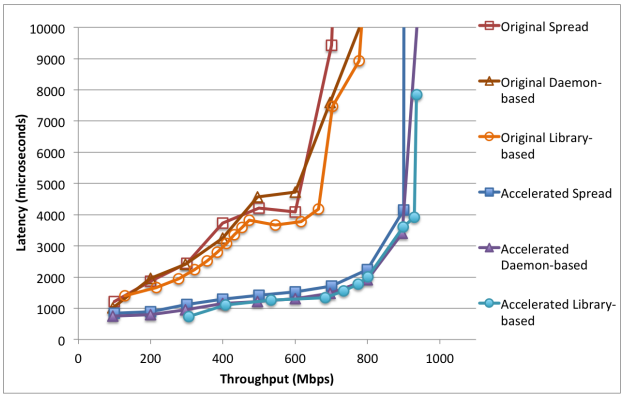


Fig. 3. Safe delivery latency vs. throughput, 1 Gb network

headers. Spread requires large headers to support features required by users, such as descriptive group and sender names.

Figures 2 and 3 show the relationship between throughput and latency for Agreed and Safe delivery, respectively. These figures show a clear difference between the profiles of the original Ring protocol and the Accelerated Ring protocol, across all implementations. When Spread uses the original protocol, its latency for Agreed delivery is at least 400 μs, even for the lowest throughput level tested (100 Mbps). In contrast, with the accelerated protocol, Spread is able to reach 400 Mbps throughput with latency below 400 μs. At 900 Mbps, Spread's latency is under 1.2 ms using the accelerated protocol, which is about the same as the latency for the original protocol at 400 Mbps. With the original protocol, Spread supports up to 500 Mbps, with latency around 1.3 ms, before latency begins to climb rapidly. The accelerated protocol supports 800 Mbps with latency around 720 μs, simultaneously improving throughput by 60% and latency by over 45%.

The accelerated protocol also improves maximum throughput compared to the original protocol. Using the accelerated protocol, Spread reaches over 920 Mbps. Since we measure only clean application data, and Spread adds substantial headers, this is practically saturating the 1-gigabit network.

Safe delivery shows a similar pattern. For all implementations, the original protocol supports up to 600 Mbps before latency begins to rise sharply, and latency is 3.7 - 4.7 ms at this point. The accelerated protocol supports 800 Mbps with latency around 2 ms, improving throughput by over 30% and latency by over 45% at the same time. The accelerated protocol achieves throughput over 900 Mbps with latency comparable to that of the original protocol at 600 Mbps.

On 1-gigabit networks, processing is fast relative to the network, so the differences between the three implementations (library-based and daemon-based prototypes and Spread) are generally small. However, for Agreed delivery, Spread has distinctly higher latency than the prototypes when the original protocol is used. In the original protocol, all received data messages must be processed before the token, and when Agreed delivery is used, these messages will generally be delivered to clients immediately upon being processed. Since delivery is relatively expensive in Spread, due to the need to analyze group names and send to the correct clients, this slows down the protocol and increases latency. This performance difference does not appear for Safe delivery, since delivering to clients is not on the critical path of processing the token when that service is used: for Safe delivery, a participant only delivers messages immediately *after* releasing the token and multicasting its new messages, so delivery does not slow down token processing. However, Safe delivery provides a stronger service than Agreed delivery and is much more expensive in terms of overall latency. In order to obtain the latency advantage of Agreed delivery, messages must be delivered to clients as soon as they are received in order. The accelerated protocol is able to maintain the latency advantage for Agreed delivery while moving delivering to clients off the critical path by allowing the token to be processed before all received messages have been processed. Therefore, the difference between Spread and the other implementations essentially disappears when the accelerated protocol is used.

2) 10-gigabit Experiments: Figures 4 and 6 show performance profiles from the same experiments shown in Figures 2 and 3, but on a 10-gigabit network. For Agreed delivery, using the original protocol, Spread can provide throughput up to about 1 Gbps before the protocol starts to reach its limits and latency climbs. Its average latency at this throughput is 385 μs. Using the accelerated protocol, Spread can provide 1.2 Gbps throughput with an average latency of about 310 μs, for a simultaneous improvement of 20% in both throughput and latency. The maximum throughput Spread reaches with latency under 1 ms using the original protocol is 1.6 Gbps, but using the accelerated protocol, Spread is able to reach 2 Gbps with similar latency, for a 25% improvement in throughput.

Unlike on 1-gigabit networks, on 10-gigabit networks, processing is slow relative to the network. Therefore, the differing overheads of the different implementations significantly affect performance. While Spread can support 1.2 Gbps throughput with latency around 310 μs using the accelerated protocol, the daemon-based and library-based prototypes support up to 2.9 Gbps and 3.5 Gbps, respectively, at the same latency.

Because processing is a bottleneck for Spread on 10-gigabit networks, the prototype implementations better show the power of the protocol. For the daemon-based prototype, the

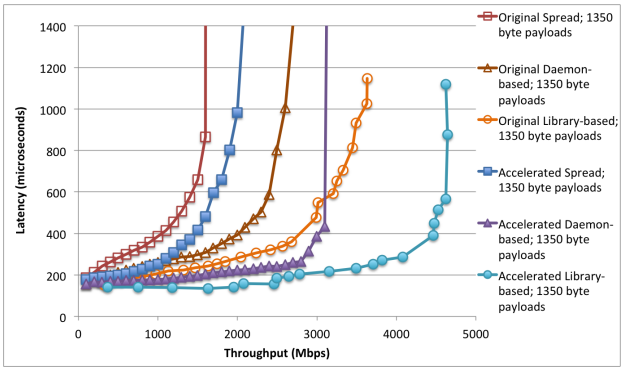


Fig. 4. Agreed delivery latency vs. throughput, 10 Gb network

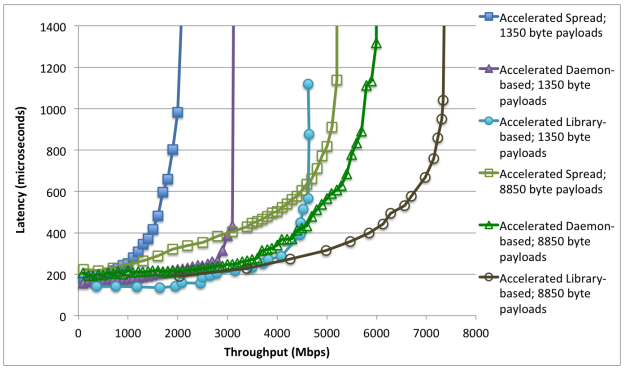


Fig. 5. Agreed delivery latency vs. throughput for 1350-byte messages and 8850-byte messages, 10 Gb network

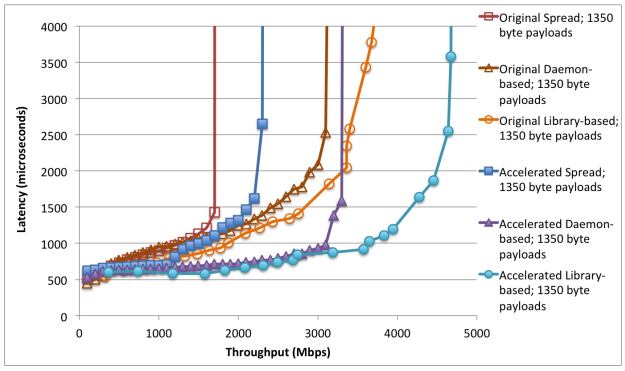


Fig. 6. Safe delivery latency vs. throughput, 10 Gb network

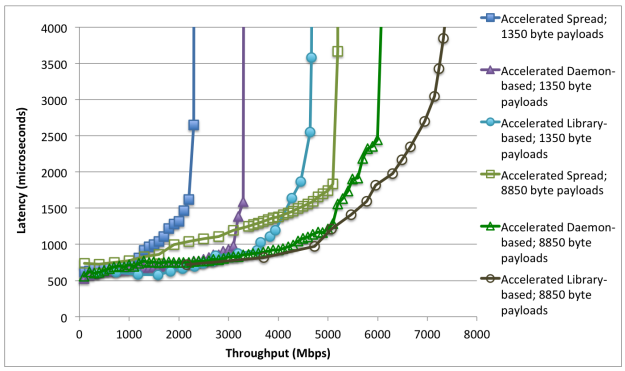


Fig. 7. Safe delivery latency vs. throughput for 1350 byte messages and 8850 byte messages, 10 Gb network

original protocol supports 2 Gbps with latency around 390 μs. The accelerated protocol supports 2.8 Gbps throughput with latency around 265 μs, for a simultaneous improvement of 40% in throughput and over 30% in latency.

The performance profile for Safe delivery is similar but with higher overall latencies for the stronger service, as well as slightly higher overall throughputs, due to the fact that message delivery is not in the critical path. Using the original protocol, Spread supports 1.1 Gbps throughput with an average latency of 930 μs; using the accelerated protocol, Spread can support the same throughput with 25% lower latency (about 700 μs), achieve 20% higher throughput with the same latency (about 1.35 Gbps), or improve both throughput and latency by about 10% each (with latency of about 810 μs for 1.2 Gbps throughput).

As for Agreed delivery, the difference between the protocols is even clearer for the prototype implementations. For the daemon-based prototype, the original protocol supports 2.5 Gbps throughput with 1.5 ms latency, while the accelerated protocol supports 3.1 Gbps with 980 μs latency, improving throughput by 25% and latency by 35% at the same time.

Using the accelerated protocol, Spread achieves a maximum throughput of 2.3 Gbps (a 35% improvement over the original protocol's maximum of 1.7 Gbps), the daemon-based prototype reaches 3.3 Gbps, and the library-based prototype reaches 4.6 Gbps.

The accelerated protocol always provides lower latency than the original protocol for the same throughput in 1-gigabit experiments or for Agreed delivery. However, Figure 8 shows that for Safe delivery on 10-gigabit networks the original protocol can provide better latency than the accelerated protocol at very low aggregate throughputs. This is because raising the token *aru* can cost up to an extra round in the accelerated protocol (because the *aru* typically cannot be raised in step with the token's *seq* value). At low throughputs on 10-gigabit networks, token rounds are already very fast, so the accelerated protocol cannot speed up each round as much, and the extra round becomes significant. At 100 Mbps, Spread's average latency is 620 μs with the accelerated protocol, which is nearly

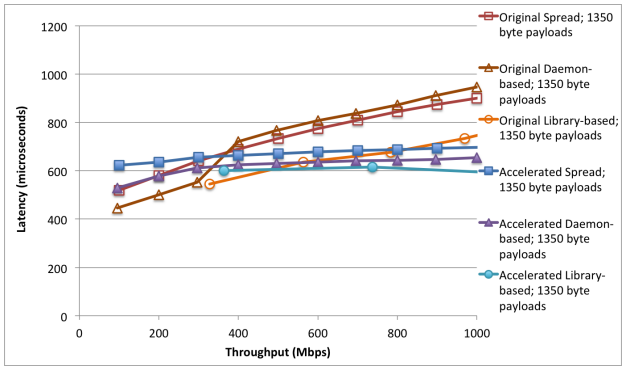


Fig. 8. Safe delivery latency for low throughputs, 10 Gb network

20% higher than the original protocol's 520 μs. However, once throughput reaches 4-5% of the network's capacity (400-500 Mbps), the accelerated protocol consistently provides lower latency.

3) 10-gigabit Experiments with Larger UDP Datagrams: On 10-gigabit networks, processing power becomes the limiting factor, preventing both the original Ring protocol and the Accelerated Ring protocol from fully utilizing the network. If higher throughput is needed, one approach is to amortize processing costs over larger message payloads. Spread includes a built-in ability to pack small messages into a single protocol packet, but the size of a protocol packet is limited to fit within a standard 1500-byte MTU; large messages are fragmented into multiple packets. However, we can use UDP datagrams of up to 64 kilobytes, allowing message fragmentation to be done at the kernel level, with the trade-off that losing a single frame causes the whole datagram to be lost.

To evaluate the performance profile of the Accelerated Ring protocol when processing costs are amortized over larger payloads, we ran the same experiments with message payloads of 8850 bytes and UDP datagrams of up to 9000 bytes. This choice is inspired by the 9000-byte jumbo frame size, allowing the same 150 bytes for headers as in the experiments with 1350-byte payloads. We chose not to use jumbo frames at the network level to avoid restricting the applicability to only deployments in which they are available, although using jumbo frames may improve performance further.

Figures 5 and 7 compare the performance profiles of the Accelerated Ring implementations using 1350-byte payloads (which are the same as in Figures 4 and 6) to the profiles using 8850-byte payloads. For both Agreed and Safe delivery, larger messages allow the protocol to reach considerably higher maximum throughputs. For Agreed delivery, the maximum throughput for Spread improves 150%, from 2.1 Gbps to 5.3 Gbps; for the daemon-based prototype it improves 87%, from 3.2 Gbps to 6 Gbps; and for the library-based protocol it improves 58%, from 4.6 Gbps to 7.3 Gbps. Because the benefit of larger messages comes from amortizing processing costs, we see the biggest improvements when processing overhead is highest. Figure 7 shows similar improvements for Safe delivery.

4) Experiments Under Loss: To evaluate the protocols under loss, we used the daemon-based prototypes, with each daemon instrumented to randomly drop a certain percentage of the data messages it received. As in the normal-case experiments, we used 8 servers, each running a daemon, a sending client, and a receiving client. Note that because messages are dropped independently at each daemon, the overall rate of retransmissions in the system is much higher than the loss rate at each daemon. In our experiments, the loss rate at each daemon ranged from 0-25%, but the actual rate of retransmissions in the system was typically 5.5 - 6.8 times higher, making this an extremely demanding experiment (the retransmission rate can be greater than 100%, since retransmissions may also be lost). With a loss rate of 25% at each daemon, the probability that a given message does not need to be retransmitted at all is only 13%.

Note that we only consider data-message loss in this evaluation, not token loss. Token loss is handled by the membership algorithm, which is identical for the two protocols. Moreover, token loss is rare, since the token is sent on a different port than data messages and received on a separate socket; since a process only needs to receive one token per round, it should always have sufficient buffer space to receive the token.

Figure 9 shows the effect of loss on latency for both Agreed and Safe delivery. In this experiment, the sending clients sent 1350-byte messages at an aggregate rate of 480 Mbps. This rate is 20% of the throughput the original protocol supports on a 10-gigabit network, which was chosen so that both protocols could support the total throughput of the experiment, even with the retransmissions. The solid lines in the figure show the average latency over all messages, while the dashed lines show the average latency over the worst (i.e. highest latency) 5% of messages from each sender.

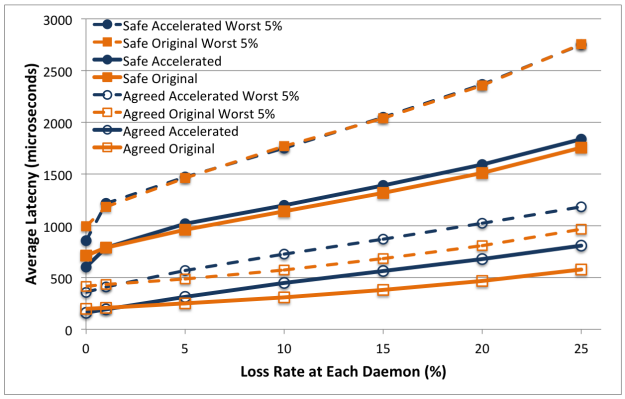


Fig. 9. Latency vs loss, 480 Mbps goodput, 10 Gb network

The Figure 9 shows that for both Agreed and Safe delivery, the accelerated protocol has higher average latency than the original protocol once the loss rate reaches 5%. This is expected, since the accelerated protocol waits an extra round to request retransmissions. For Agreed delivery, the accelerated protocol's latency can be up to 47% higher than that of the original protocol. However, for Safe delivery, the difference is only 5-7%. Safe delivery takes longer overall in terms of token rounds, so the fact that the Accelerated Ring protocol reduces the time each token round takes has a larger impact on Safe delivery latency and partially compensates for the extra round needed to request retransmissions.

The highest latency messages include those that need to be retransmitted more than once and those that wait close to a full token round to be sent for the first time (because they arrived from the client at the daemon just after the daemon released the token). As the number of overall rounds increases, the accelerated protocol's advantage of shorter rounds has a greater impact and offsets the additional round needed to request retransmsissions. Therefore, for the highest latency messages (dashed lines in Figure 9), the accelerated protocol is slightly more similar to the original protocol in Agreed

delivery latency (with up to 26% higher latency) and provides about the same Safe delivery latency as the original protocol.

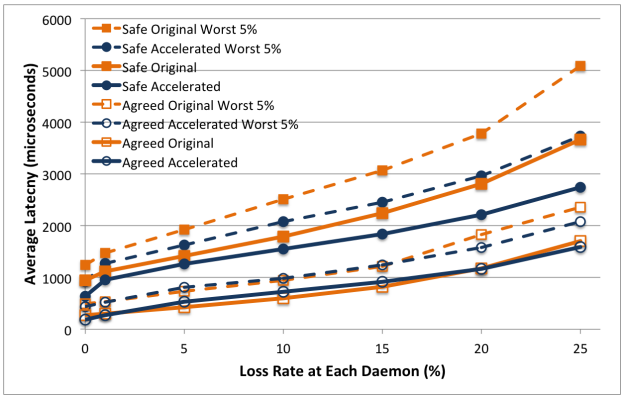


Fig. 10. Latency vs loss, 1200 Mbps goodput, 10 Gb network

Figure 10 shows the results of the same experiment as Figure 9, but with clients sending at an aggregate rate of 1200 Mbps, representing 50% of the throughput the original protocol supports on a 10-gigabit network. At this throughput level, Safe delivery latency is lower for the accelerated protocol than for the original protocol, while Agreed delivery latency is similar for both under all tested loss rates. At higher throughputs, each token round takes substantially longer in the original protocol, since each daemon has more messages to send in each round and must send them all before releasing the token. Because the accelerated protocol does not force daemons to send all of their messages before releasing the token, increased throughput has less effect on latency.

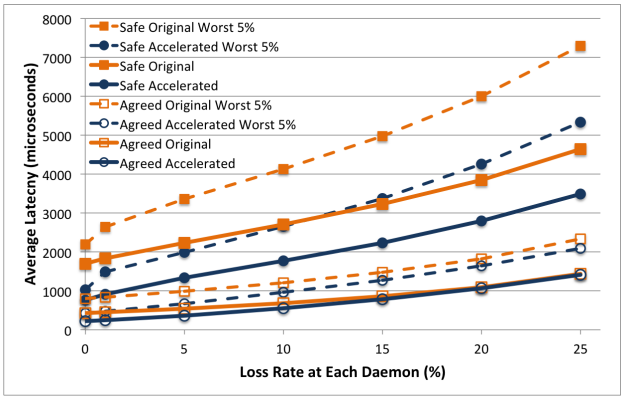


Fig. 11. Latency vs loss, 140 Mbps goodput, 1 Gb network

Figure 11 shows the results of the same experiment but on a 1-gigabit network, with sending clients sending at an aggregate rate of 140 Mbps (20% of the throughput the original protocol supports on a 1-gigabit network). On a 1-gigabit network, the latency improvement from using the accelerated protocol is so great that even with the additional round that the accelerated protocol needs to request lost messages, the accelerated protocol provides better latency than the original protocol under all tested loss rates, providing slightly lower latency for Agreed delivery (2-45%) and 22-50% lower latency for Safe delivery, depending on the loss rate.

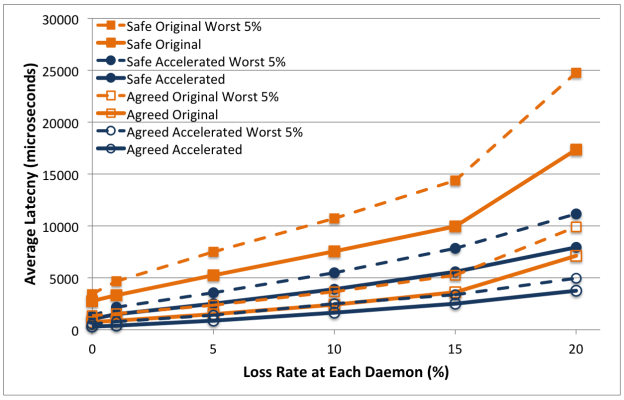


Fig. 12. Latency vs loss, 350 Mbps goodput, 1 Gb network

Focusing on the highest latency messages or sending at higher throughputs shows the same pattern as on 10-gigabit networks – the overall latencies are higher, but the advantage of the accelerated protocol over the original protocol increases.

Figure 12 shows the results with clients sending at an aggregate rate of 350 Mbps, which is 50% of the throughput the original protocol supports on a 1-gigabit network. For Agreed delivery, the average latency of the accelerated protocol is at least 30% lower than that of the original protocol for all tested loss rates, and for Safe delivery the average latency of the accelerated protocol is at least 43% lower than that of the original protocol.

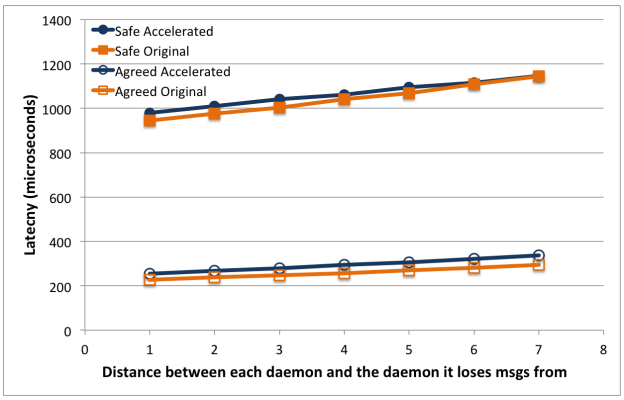


Fig. 13. Effect of the relative positions of daemons losing messages and the daemons they are losing from in the ring on latency

Figure 13 shows how latency is affected by the relative positions of a daemon losing messages and the daemon it loses messages from in the ring. In this experiment, each daemon lost 20% of the messages that were sent by the daemon a certain number of positions before it in the ring. We varied the distance between the daemons from 1 position apart (i.e. each daemon loses from its immediate predecessor on the ring) to 7 positions apart (i.e. each daemon loses from the daemon 7 positions before it in the ring – its immediate successor). As

the distance between each daemon and the daemon it is losing from increases, latency increases as well. This is because a message cannot be requested until the token reaches the daemon that lost that message. Therefore, when the daemon that missed a message is 7 positions after the daemon that sent that message, it takes nearly a full token round longer for it to be able to request the message than if the daemon that lost the message immediately followed the daemon that sent it in the ring.

B. Discussion

The evaluation clearly shows the benefit of the Accelerated Ring protocol over a standard token-based protocol. Using the accelerated protocol, Spread, with all the overhead of a real system, is able to practically saturate a 1-gigabit network. In fact, it achieves similar or better network utilization than was reported in 2004 on 100-megabit Fast Ethernet [8], with excellent latency.

Using larger UDP datagrams (but not jumbo frames), a library-based prototype is able to achieve about 73% network utilization on a 10-gigabit network. This is comparable to the original Totem Ring protocol, which was benchmarked similarly to our library-based implementation on a 10-megabit network in [2], [3]. Thus, a relatively simple but powerful protocol change, plus the use of larger UDP datagrams, allows the same core protocol to scale three orders of magnitude over 20 years.

The current version of Spread uses UDP datagrams that fit in a single frame with a 1500-byte MTU. While we are not considering using jumbo frames in the default configuration, allowing larger UDP datagrams on 10-gigabit networks may be beneficial. A drawback of UDP datagrams that span multiple network frames is that losing a single frame causes the whole datagram to be lost. However, using a protocol with good flow control, like the Accelerated Ring protocol, in a stable local area network, like those in data center environments, we expect loss to be small. Therefore, larger UDP datagrams may be a reasonable trade-off for applications that need both high throughput and all of the features of Spread. Currently, larger UDP datagrams may be used by changing a single constant parameter and recompiling Spread, but including this capability as the default configuration option in future releases warrants further experimentation and experience.

The Accelerated Ring protocol's performance improvement comes at the cost of waiting an extra token round to recover lost messages. While this can make the accelerated protocol's latency higher than that of the original protocol when there is significant loss on the network, the evaluation shows that this effect only appears on 10-gigabit networks at relatively low aggregate throughputs. At higher throughputs or on 1-gigabit networks, the accelerated protocol's latency advantage over the original protocol compensates for the extra round, keeping its overall latency lower than that of the original protocol.

V. Related Work

The Accelerated Ring protocol builds on a large body of work on reliable, totally ordered multicast. Existing protocols vary in the techniques they use to achieve total ordering, as well as the precise semantics they guarantee.

One of the earliest token-based protocols is the protocol of Chang and Maxemchuk [14]. This protocol exemplifies a token-based moving sequencer protocol under the classification of Défago et al. [1]. It orders messages by passing a token around a logical ring of processes; the process holding the token is responsible for assigning a sequence number to each message it receives and sending that sequence number to the other processes. The Pinwheel protocols [15] introduce several performance optimizations to the Chang and Maxemchuk protocol, including new mechanisms for determining message stability.

As previously discussed, the Accelerated Ring protocol is closely related to the Totem Ring protocol [2], [3]. Spread's original protocol is a variant of Totem, and the Accelerated Ring protocol directly uses the variant of the Totem membership algorithm implemented in Spread.

While we chose a token-based approach for its simplicity and ability to provide flexible semantics, other total ordering mechanisms exist. Défago et al. identify sequencer, moving sequencer, privilege-based, communication history, and destinations agreement as the five main mechanisms for total ordering and survey algorithms of each type [1].

One popular approach to ordering is to use the Paxos algorithm [16] to order messages by executing a series of consensus instances to assign sequence numbers to messages. This approach has been used to perform replication in many well-known systems, including Boxwood [17], Chubby [18], and Spanner [19].

While Paxos provides total ordering, it does not offer exactly the same semantics as the Accelerated Ring protocol. We designed the Accelerated Ring protocol to maintain the flexible Extended Virtual Synchrony semantics that our experience has shown to be useful in supporting a wide variety of applications. Paxos-like approaches lack this flexibility; they provide a service similar to Safe delivery but cannot provide weaker services for a lower cost, and they require a quorum to make progress. In addition, the total order of Paxos-based approaches does not respect FIFO ordering if a process can submit a message for ordering before learning that its previous messages were ordered. However, Paxos's semantics are appropriate for many applications, and numerous Paxos variants have been designed to optimize particular aspects of the protocol [20]–[26]. For example, Egalitarian Paxos [26] eliminates the leader used in most traditional Paxos variants to achieve better load distribution and availability in the presence of failures and uses a fast-path for non-interfering commands to improve latency by reducing the number of network round-trips required for such commands.

Ring Paxos [24] is the most relevant Paxos variant to our work, as it shares our goal of high throughput and uses

ring-based communication. It achieves high performance by using efficient communication patterns in which messages are forwarded to a coordinator that multicasts them to all processes and acknowledgments are forwarded around a logical ring composed of a quorum of processes. Multi-Ring Paxos [27] scales the throughput of Ring Paxos by running multiple rings that order messages independently. Processes that need to receive messages from more than one ring use a deterministic merge function to obtain a total ordering over all messages.

The popular Zookeeper [28] coordination service uses the ZAB [29] protocol for primary-backup replication. ZAB is similar to Paxos but enforces FIFO ordering even when a process can have multiple outstanding messages and provides stronger ordering guarantees across leader failures. Like Paxos, ZAB requires a quorum to make progress.

The ISIS toolkit [30] was one of the first practical group communication systems and was used in air traffic control systems and the New York Stock Exchange, among many other places. ISIS is based on the Virtual Synchrony model and uses vector timestamps to establish causal ordering and a sequencer-based protocol to establish total ordering.

While sequencer-based approaches like the one in ISIS can provide a variety of service levels, their handling of network partitions is typically limited, preventing them from cleanly merging partitioned members. Members that do not belong to a primary component typically need to rejoin as new members. We focus on token-based approaches in part because they can naturally provide rich semantics in a partionable model.

In the Trans protocol [31], used by the Transis system [32], processes attach positive and negative acknowledgments to messages they multicast. The processes then use these acknowledgments, along with per-process sequence numbers, to build a directed acyclic graph over the messages, which determines the order in which they must be delivered.

These alternative approaches to ordering have also been used to build successful practical messaging systems that can achieve good performance on modern networks. JGroups [33] is a popular, highly-configurable messaging toolkit that includes a total ordering protocol based on a sequencer (as well as less expensive FIFO ordering). A 2008 performance evaluation reports that, with an 8-machine cluster on a 1-gigabit network, the FIFO multicast protocol achieves 405 Mbps throughput with 1000-byte messages and reaches 693 Mbps with 5000-byte messages [34]. Using the same setup described in Section IV, we measured the total ordering protocol as reaching 650 Mbps on a 1-gigabit network with 1350-byte messages (with the FIFO protocol reaching 880 Mbps) and up to 3 Gbps on a 10-gigabit network (with the FIFO protocol reaching over 4 Gbps).

Isis2 [35], [36] is a cloud computing library that, among other capabilities, includes totally ordered multicast using a sequencer-based protocol, stronger safety guarantees using Paxos, and a weaker, cheaper FIFO multicast. Isis2 allows users to work with replicated distributed objects (automatically maintaining consistent copies at all processes using its multicast primitives), and provides other useful features, such as a built-in distributed hash table and support for out-of-band file transfer.

U-Ring Paxos [37] adapts Ring Paxos and Multi-Ring Paxos to work without IP-multicast by propagating both acknowledgments and the content being ordered around the ring. While providing efficient (logical) multicast in the absence of IP-multicast is important for certain environments (e.g. WANs), significant effort has been spent on optimizing IP-multicast and allowing it to support large numbers of groups; our experience has shown that it is reasonable to take advantage of its performance in data center environments, especially when building infrastructure. Using a single ring, U-Ring Paxos is reported to reach over 900 Mbps on 1-gigabit networks in configurations with 3 acceptors sending 8-kilobyte messages [37], which matches our measurements in the same 8-machine setup described in Section IV. When sending 1350-byte messages (but allowing batching), we measure U-Ring Paxos as reaching over 750 Mbps with a latency profile similar to that of the original Ring protocol for Safe delivery. On a 10-gigabit network, we measure U-Ring Paxos as reaching close to 1.5 Gbps with 1350-byte messages (with batching). Higher throughput may be achieved by taking advantage of Multi-Ring Paxos's ability to run multiple rings in parallel, but this requires additional processing resources.

VI. Conclusion

We presented the Accelerated Ring protocol, a new protocol for totally-ordered multicast that is designed to take advantage of the trade-offs of 1-gigabit and 10-gigabit networks. The Accelerated Ring protocol significantly improves both throughput and latency compared to standard token-based protocols, while maintaining the correctness and attractive properties of such protocols. The protocol is available as open-source and has been adopted as the default protocol for local area networks and data center environments in the Spread toolkit.

Acknowledgment

This work was supported in part by DARPA grant N660001-1-2-4014. Its contents are solely the responsibility of the authors and do not represent the official view of DARPA or the Department of Defense.

References

[1] X. Défago, A. Schiper, and P. Urbán, "Total order broadcast and multicast algorithms: Taxonomy and survey," *ACM Comput. Surv.*, vol. 36, no. 4, pp. 372–421, Dec. 2004.

[2] Y. Amir, L. Moser, P. Melliar-Smith, D. Agarwal, and P. Ciarfella, "Fast message ordering and membership using a logical token-passing ring," in *Proc. IEEE Int. Conf. Distributed Computing Syst. (ICDCS)*, May 1993, pp. 551–560.

[3] Y. Amir, L. E. Moser, P. M. Melliar-Smith, D. A. Agarwal, and P. Ciarfella, "The totem single-ring ordering and membership protocol," *ACM Trans. Comput. Syst.*, vol. 13, no. 4, pp. 311–342, Nov. 1995.

[4] Spread Concepts LLC, "The Spread Toolkit," http://www.spread.org, retrieved Dec. 6, 2015.

[5] "The Corosync cluster engine," http://corosync.github.io/corosync, retrieved Dec. 6, 2015.

[6] "Appia communication framework," http://appia.di.fc.ul.pt, retrieved Dec. 6, 2015.

[7] H. Miranda, A. Pinto, and L. Rodrigues, "Appia, a flexible protocol kernel supporting multiple coordinated channels," in *Proc. IEEE Int. Conf. Distributed Computing Syst. (ICDCS)*, Apr 2001, pp. 707–710.

[8] Y. Amir, C. Danilov, M. Miskin-Amir, J. Schultz, and J. Stanton, "The spread toolkit: Architecture and performance," Johns Hopkins University, Tech. Rep. CNDS-2004-1, 2004.

[9] L. Moser, Y. Amir, P. Melliar-Smith, and D. Agarwal, "Extended virtual synchrony," in *Proc. IEEE Int. Conf. Distributed Computing Syst. (ICDCS)*, Jun 1994, pp. 56–65.

[10] Y. Amir, "Replication using group communication over a partitioned network," Ph.D. dissertation, Hebrew University of Jerusalem, 1995.

[11] K. Birman and T. Joseph, "Exploiting virtual synchrony in distributed systems," in *Proc. Symp. Operating Syst. Principles*, 1987, pp. 123–138.

[12] R. Kotla, L. Alvisi, M. Dahlin, A. Clement, and E. Wong, "Zyzzyva: Speculative byzantine fault tolerance," in *Proc. Symp. Operating Syst. Principles (SOSP)*, 2007, pp. 45–58.

[13] D. Dolev, S. Kramer, and D. Malki, "Early delivery totally ordered multicast in asynchronous environments," in *Proc. Int. Symp. Fault-Tolerant Computing (FTCS)*, June 1993, pp. 544–553.

[14] J.-M. Chang and N. F. Maxemchuk, "Reliable broadcast protocols," *ACM Trans. Comput. Syst.*, vol. 2, no. 3, pp. 251–273, Aug. 1984.

[15] F. Cristian and S. Mishra, "The pinwheel asynchronous atomic broadcast protocols," in *Proc. Int. Symp. Autonomous Decentralized Syst.*, Apr 1995, pp. 215–221.

[16] L. Lamport, "The part-time parliament," *ACM Trans. Comput. Syst.*, vol. 16, no. 2, pp. 133–169, May 1998.

[17] J. MacCormick, N. Murphy, M. Najork, C. A. Thekkath, and L. Zhou, "Boxwood: Abstractions as the foundation for storage infrastructure," in *Proc. Operating Syst. Design and Implementation*, 2004, pp. 105–120.

[18] M. Burrows, "The chubby lock service for loosely-coupled distributed systems," in *Proc. Symp. Operating Syst. Design and Implementation (OSDI)*, 2006, pp. 335–350.

[19] J. C. Corbett *et al.*, "Spanner: Google's globally distributed database," *ACM Trans. Comput. Syst.*, vol. 31, no. 3, pp. 8:1–8:22, Aug. 2013.

[20] L. Lamport and M. Massa, "Cheap paxos," in *Proc. IEEE/IFIP Int. Conf. Dependable Syst. and Networks (DSN)*, June 2004, pp. 307–314.

[21] L. Lamport, "Generalized consensus and paxos," Microsoft Research, Tech. Rep. MSR-TR-2005-33, March 2005. [Online]. Available: http://research.microsoft.com/apps/pubs/default.aspx?id=64631

[22] ——, "Fast paxos," *Distributed Computing*, vol. 19, no. 2, pp. 79–103, October 2006. [Online]. Available: http://research.microsoft.com/apps/pubs/default.aspx?id=64624

[23] Y. Mao, F. P. Junqueira, and K. Marzullo, "Mencius: Building efficient replicated state machines for WANs," in *Proc. Operating Syst. Design and Implementation (OSDI)*, 2008, pp. 369–384.

[24] P. Marandi, M. Primi, N. Schiper, and F. Pedone, "Ring paxos: A high-throughput atomic broadcast protocol," in *Proc. IEEE/IFIP Int. Conf. Dependable Syst. and Networks (DSN)*, June 2010, pp. 527–536.

[25] M. Biely, Z. Milosevic, N. Santos, and A. Schiper, "S-paxos: Offloading the leader for high throughput state machine replication," in *Proc. Symp. Reliable Distributed Syst. (SRDS)*, Oct 2012, pp. 111–120.

[26] I. Moraru, D. G. Andersen, and M. Kaminsky, "There is more consensus in egalitarian parliaments," in *Proc. Symp. Operating Syst. Principles (SOSP)*, 2013, pp. 358–372.

[27] P. Marandi, M. Primi, and F. Pedone, "Multi-ring paxos," in *Proc. IEEE/IFIP Int. Conf. Dependable Syst. and Networks (DSN)*, June 2012, pp. 1–12.

[28] P. Hunt, M. Konar, F. P. Junqueira, and B. Reed, "Zookeeper: Wait-free coordination for internet-scale systems," in *Proc. USENIX Annual Technical Conference*, 2010, pp. 145–158.

[29] F. Junqueira, B. Reed, and M. Serafini, "Zab: High-performance broadcast for primary-backup systems," in *Proc. IEEE/IFIP Int. Conf. Dependable Syst. and Networks (DSN)*, June 2011, pp. 245–256.

[30] K. Birman, A. Schiper, and P. Stephenson, "Lightweight causal and atomic group multicast," *ACM Trans. Comput. Syst.*, vol. 9, no. 3, pp. 272–314, Aug. 1991.

[31] P. Melliar-Smith, L. Moser, and V. Agrawala, "Broadcast protocols for distributed systems," *IEEE Trans. Parallel and Distributed Syst.*, vol. 1, no. 1, pp. 17–25, Jan 1990.

[32] Y. Amir, D. Dolev, S. Kramer, and D. Malki, "Transis: a communication subsystem for high availability," in *Proc. Int. Symp. Fault-Tolerant Computing (FTCS)*, July 1992, pp. 76–84.

[33] "JGroups - The JGroups Project," http://www.jgroups.org, retrieved Dec. 6, 2015.

[34] B. Ban, "Performance tests JGroups 2.6.4," www.jgroups.org/perf/perf2008/Report.html, August 2008, retrieved Dec. 6, 2015.

[35] K. Birman, "Isis2 Cloud Computing Library," http://isis2.codeplex.com, retrieved Dec. 6, 2015.

[36] K. Birman and H. Sohn, "Hosting dynamic data in the cloud with Isis2 and the Ida DHT," in *Workshop Timely Results in Operating Syst.*, 2013.

[37] S. Benz, "Unicast multi-ring paxos," Master's thesis, Università della Svizzera Italiana, 2013.