Fast Total Ordering for Modern Data Centers

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

Technical Report CNDS-2014-2 - December 2014
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 prior to this change. 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 the Spread toolkit, 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 existing 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, which Défago et al. split into *moving sequencer* and *privilege-based* categories.

Token-based protocols typically arrange the processes participating in the protocol in a logical ring and order messages using a token (a special control message) that carries the information needed to order new messages and is passed around the ring of processes. In moving sequencer protocols, any protocol participant can send a message at any time, but a message is not ordered until the token-holder assigns it a sequence number. In privilege-based protocols, a protocol participant may only send messages to the group upon receiving the token, but each participant is able to assign sequence numbers to its messages before sending them, so messages are ordered at the time they are sent. In the remainder of the paper, we use the term *token-based* to refer primarily to privilege-based protocols, since the protocols that have translated into practical systems are largely of this type, although some ideas may apply to moving sequencer protocols as well.

Token-based protocols are attractive because of their simplicity; a single mechanism, the token, provides ordering, stability notification, flow control, and fast failure detection. Such protocols also achieved high network utilization at the time they were introduced; the Totem Ring protocol [2], [3], for example, achieved about 75% network utilization on 10-megabit Ethernet using processors standard for 1995. The simplicity 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], the Appia communication framework [6], [7], and others.

When Fast Ethernet replaced the original 10-megabit Ethernet, network speed increased by a factor of ten, and network span shrunk by the same factor (from 2000 to 200 meters) so that basic network characteristics remained essentially the same. This allowed 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 the Spread toolkit, a publicly available, widely-used, 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 Spread 4.3 (the latest release prior to our work) as reaching 50% network utilization on a 1-gigabit network. Careful tuning of the flow control parameters, which many users are unlikely to attempt, allows Spread 4.3 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 (that 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 latency improvement 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 not concerned with latency due to particular machines running slowly (e.g. because their capacity is consumed by many running processes). In production deployments, we are able to ensure that the ordering middleware has the resources it needs (e.g. with real-time priorities in Linux). Instead, we are concerned with latency intrinsic to the network.

Furthermore, for common processors today, 10-gigabit 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.

Moreover, single-threaded implementations of this 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 a single core is important for this type of ordering service, as it is intended to provide a service to applications that may be CPU intensive; an ordering service that consumes a majority of the processing power of the machine is likely to be limiting in many common practical deployments.

Similarly to 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 information the next participant needs, the next participant can begin multicasting as soon as it receives the token, even if its predecessor on the ring has not yet completed its multicasting for the current token rotation.

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.

To obtain the maximum benefit from passing the token early, it is also necessary to consider when the token should be processed relative to data messages when a token message and data messages are received and available for processing at the same time. The token should be processed quickly, to maintain the benefit of accelerating its rotation, but if it is processed too early, unnecessary retransmissions may be requested, or excessive overlap in the sending of different participants may cause packets to be lost due to buffer exhaustion.

The ability to send 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 is able to simultaneously provide higher throughput and lower latency than a standard token-based protocol.

We evaluate the Accelerated Ring protocol in three implementations: a library-based prototype, a daemon-based prototype, and a production implementation in the Spread toolkit. Spread has existed for over 20 years, serving production systems of various scales since the late 1990s. 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.

An evaluation comparing each implementation of the Accelerated Ring protocol to a corresponding implementation of the original Ring protocol of Totem shows the following results: Sending messages with 1350 byte payloads, on 1-gigabit networks, the Spread implementation of the Accelerated Ring protocol reaches over 920 Mbps in throughput (measuring only clean application data), essentially reaching network saturation, and can reduce latency by 45% compared to the original protocol, while simultaneously increasing throughput by 45-60%. The daemon-based and library-based prototypes exhibit similar performance. On 10-gigabit networks, the Spread implementation reaches 2.3 Gbps in throughput, and can reduce latency by 10-20% while simultaneously increasing throughput by 10-20%. The daemon-based prototype reaches 3.3 Gbps in throughput, and can reduce latency by 30-35% while increasing throughput by 25-40%. The library-based prototype reaches 4.6 Gbps in throughput, and can reduce latency by 20-30% while increasing throughput by 35-50%. Sending messages with 8850 byte payloads, using UDP datagrams that are large enough to contain an entire message (but without using jumbo frames), the maximum throughputs reach 5.3 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 tradeoffs in modern data center environments.
2) A thorough evaluation of the protocol in a library-based prototype, a daemon-based prototype, and the Spread toolkit in 1-gigabit and 10-gigabit networks.
3) The release of the Accelerated Ring protocol as part of Spread's open-source code in version 4.4, and the adoption of this protocol as Spread's standard ordering protocol for local area networks and data center environments.

The remainder of the paper proceeds as follows: Section II specifies the system model and the services the Accelerated Ring protocol provides, Section III specifies the Accelerated Ring protocol, Section IV evaluates the prototype and practical implementations of the protocol, Section V discusses how this protocol relates to existing work on total ordering protocols, and Section VI concludes the paper.

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, just 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*.

Note that the Accelerated Ring protocol can also provide FIFO and Causal delivery, but the delivery latency is similar to that of Agreed delivery. Since the guarantees of Agreed delivery subsume those of FIFO and Causal delivery, these services are not discussed separately.

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 [4], 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 Accelerated Ring protocol orders messages by circulating a token around a logical ring composed of the protocol participants. The token carries the information each participant needs to correctly assign a sequence number to each message it sends, and the sequence number specifies the message's position in the total order. As discussed in Section I, this basic mechanism is used in other privilege-based token-based protocols as well.

The innovation of the Accelerated Ring protocol is that a participant may continue to multicast for a short time after passing the token. This reduces or eliminates periods in which no process is multicasting, allows for controlled parallelism in multicasting, and reduces the time needed to complete a *token round* (a rotation of the token around the ring).

The following description specifies the normal-case operation of the Accelerated Ring protocol with a static set of participants. Participant failures and network partitions and merges are not considered here, as they are handled by the membership algorithm. The membership algorithm is exactly the algorithm used by the variant of the Totem Ring protocol [2], [10] that is implemented in Spread. This description assumes that the membership of the ring has been established, and the first regular token has been sent.

During normal operation, participants take actions in response to receiving messages. A participant can receive two types of messages: token messages and data messages.

A. 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 used in the ordering protocol:

1) *seq*: The last sequence number claimed by a participant. Upon receiving a token, a participant is able to initiate multicast messages with sequence numbers starting at *seq* + 1.
2) *aru* (all-received-up-to): This field is 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 that have been received by all participants can be delivered with Safe delivery semantics and/or garbage-collected.

3) *fcc* (flow control count): The 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): A list of sequence numbers corresponding to messages that must be retransmitted (because they were lost by some participant).

Upon receiving the token, a participant multicasts messages (potentially including both retransmissions and new messages), updates the token fields, passes the token to the next participant, delivers newly deliverable messages, and discards messages that it no longer needs. The key to the acceleration of the token is that the token can be updated and passed to the next participant *before* a participant finishes multicasting for the round. This is accomplished by splitting the multicasting into a pre-token phase and a post-token phase.

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 any new messages it will initiate and any retransmissions it will send. This is necessary in order for 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 will first answer any retransmission requests that it can. Specifically, for each retransmission request in the rtr field of the token it has just received, the participant will check whether it has the corresponding message. If it does, it will retransmit that message. Note that all retransmissions must be sent during the pre-token phase; otherwise, they may be unnecessarily requested again.

After completing its retransmissions, the participant will choose the new messages it will initiate in this round (if any) and multicast some fraction of them to the group (to pass the token as quickly as possible, this fraction may be 0).

The maximum number of new messages the participant can multicast in the current round is calculated as the minimum of the number of application messages it currently has waiting to be multicast, $Personal\_window$, $(Global\_window - received\_token.fcc - num\_retrans)$, and $(Global\_aru + Max\_seq\_gap - received\_token.seq)$. The $Personal\_window$ is the maximum number of new multicast messages that can be sent in a single token round by a single participant. The $Global\_window$ is the maximum number of multicast messages that can be initiated in a single token round by all participants combined. The token's *fcc* field is used to limit the total number of messages multicast in a single round (including retransmissions and new messages). Max_seq_gap limits the gap between the highest sequence number that can be assigned to a new message and the highest sequence number known to have been received by all participants (which we call the $Global\_aru$).

The number of new messages the participant will actually send in the pre-token phase depends on the $Accelerated\_window$ parameter. The $Accelerated\_window$ is the maximum number of messages a participant is permitted to send after passing the token to the next participant in the ring. The number of new messages a participant will send in the pre-token phase is therefore the total number of messages it will send in the current round, minus the $Accelerated\_window$. If the total number of new messages a participant will send in the current round is less than or equal to the $Accelerated\_window$, it will not send any new messages in the pre-token phase.

To determine the full set of messages it will send in the current round, while multicasting only those messages that are required to be sent before the token, the participant puts messages into a queue. The participant prepares messages exactly as if it were going to send them, but instead of immediately sending messages, the participant places each message in the queue. If adding a message to the queue causes the length of the queue to exceed the $Accelerated\_window$, the participant removes the first message from the queue and multicasts it. The pre-token multicasting phase ends when the participant has enqueued all the messages it is able to send in the current round. At this point, the queue will hold either $Accelerated\_window$ messages, or all the new messages the participant will send in this round, if the total number of new messages it can send in this round is less than or equal to the $Accelerated\_window$.

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 set to the highest sequence number that has been assigned to a message. For each new message the participant enqueues during the pre-token multicast phase, seq is incremented, and the resulting sequence number is assigned to the message. Since the participant enqueues all the new messages it will send for the current round during the pre-token multicast phase, this ensures that the seq field will reflect all the messages this participant will send in the current round, including both the pre-token and post-token multicasting phases.

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 had 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, the participant increments the aru along with the seq field as it prepares and enqueues new messages during the pre-token multicast phase.

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 to 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 slightly complicates retransmission requests, since the seq field of the received 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 in the ring.

3) Post-token Multicasting: During the post-token multicast phase, the participant simply flushes the queue created in the pre-token multicast phase by multicasting each message remaining in the queue. This completes the participant's sending for the current round.

4) Delivering and Discarding: As the final step of token handling, the Accelerated Ring protocol uses the procedure described in [2], [3] to determine which messages can be delivered and discarded. A participant can deliver an Agreed message once it has received and delivered all messages with lower sequence numbers. Thus, if a participant has received and delivered all messages up through the seq value of the received token, it immediately delivers any new messages it multicasts in the current round.

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 Safe 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. The participant discards each message that meets the requirement for Safe delivery (after delivering it). Since Safe delivery requires that every participant has the message, these messages will no longer be requested for retransmission.

B. 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: The sequence number of this message. This corresponds to the message's position in the total order.
2) pid: The ID of the participant that initiated this message.
3) $round$: The token round in which this message was initiated.
4) $payload$: The application data. This is not used or inspected by the protocol.

Upon receiving a data message, a participant inserts it into its buffer of messages, ordered by its sequence number. If this message allows the participant to advance its local aru (i.e. the sequence number of the message is equal to the local $aru + 1$), it delivers all 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. To maintain the total order, no Agreed messages with sequence numbers higher than that of the first undelivered Safe message can be delivered until after that Safe message is delivered.

C. Selecting a Message to Handle

In general, token and data messages are handled as they arrive. However, when messages arrive simultaneously or nearly simultaneously, it is possible to have both token and data messages available for processing at the same time. In this case, the protocol must decide which message type to handle first. Logically, this is accomplished by assigning different priorities to the 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 for processing.

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 actually lost, they will be unnecessarily requested. However, in order to keep the token moving as quickly as possible and maximize performance, the token should be processed as soon as it is safe to do so without causing unnecessary retransmissions. 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 message. However, two different methods may be used for deciding when to raise the priority of token messages 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 (which is 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 (i.e. during its post-token multicast phase). A more detailed discussion of these methods can be found in [12].

If each participant is able to process every message immediately as it arrives, the priority switching method does not have an impact. However, when participants can receive the token before they have finished processing all previously received messages, the two priority-switching methods offer different performance trade-offs. The first maximizes the speed with which the token can circulate the ring. The second slows the token slightly to reduce the number of unprocessed data messages that can build up at a participant but still maintains the acceleration of the token by processing it at its correct place in the stream of messages (after messages sent before it, but before messages known to have been sent after it).

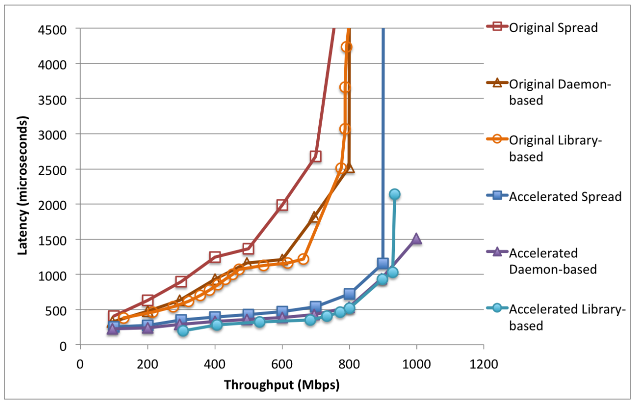

Fig. 1. Agreed delivery latency vs. throughput, 1-gigabit network

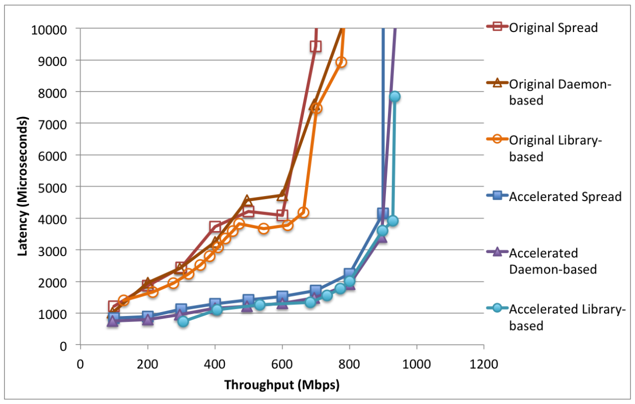

Fig. 2. Safe delivery latency vs. throughput, 1-gigabit network

D. Implementation Considerations

Our implementations of the Accelerated Ring protocol use IP-multicast to transmit data messages to all participants and UDP unicast to pass the token from one participant to the next in the ring. 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; 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. Other work has reported that Spread's maximum throughput is less than 200 Mbps on a 1-gigabit network [13]; such low throughput is likely the result of a setup in which TCP connections between Spread daemons and clients became the bottleneck.

In Section III-C, we describe switching priority between data messages and token messages. In practice, we accomplish this by sending token and data messages on different ports and using different sockets for receiving the two message types. 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.

When evaluating both prototypes, we use the first, more aggressive method for giving the token high priority, since this gives the best performance when the flow control parameters are properly tuned. However, when evaluating Spread, we use the second, less aggressive method, since this is the method implemented in the open-source Spread release. We chose to implement this method in Spread because it is less sensitive to misconfiguration and provides more stable, predictable performance, which is important for production systems used in a wide variety of environments. When the accelerated window is set to zero at all participants, the second method is identical to 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 Ring protocol of Totem [2], [3]. We evaluate both protocols in library-based and daemon-based prototype implementations as well as in complete production implementations in the Spread toolkit.

A. Benchmarks

All benchmarks use 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 the performance profile of each protocol, we ran the system at different throughput levels, ranging from 100 Mbps up to the maximum throughput of the system for each protocol and implementation. At each throughput level, we measured the average latency to deliver a message for both Agreed and Safe delivery.

These experiments were run on 8 servers. For Spread and the daemon-based prototype, each server ran one daemon, one sending client that injected messages into the system 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 a single process that both injected and received messages.

For the library-based prototype, we controlled throughput by adjusting the personal window; 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). For the daemon-based prototype and Spread, we chose the smallest personal window that allowed the system to reach its maximum throughput and the accelerated window that resulted in the highest throughput for that particular personal window, as this gave the best overall system performance.

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

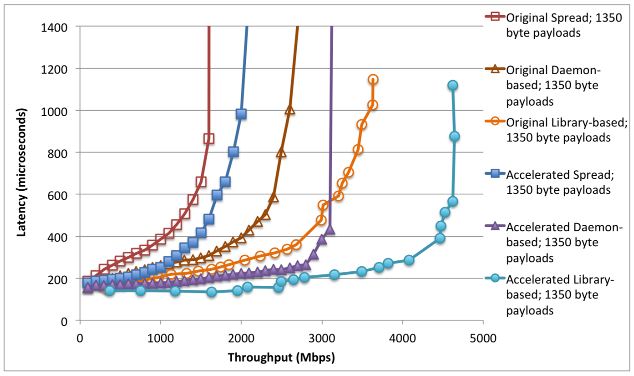

Fig. 3. Agreed delivery latency vs. throughput, 10-gigabit network

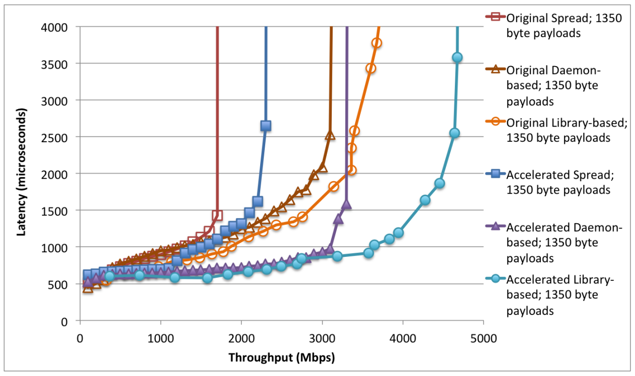

Fig. 5. Safe delivery latency vs. throughput, 10-gigabit network

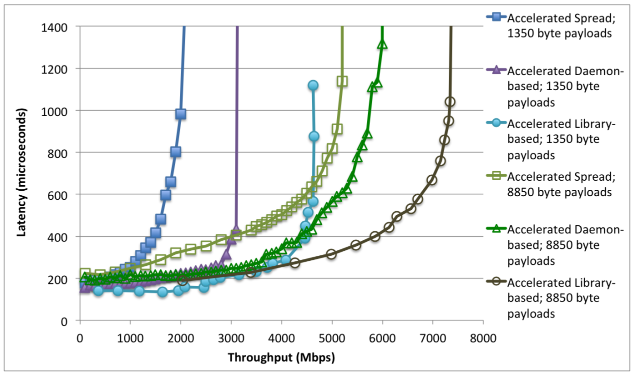

Fig. 4. Throughput vs agreed delivery latency for 1350 byte messages and 8850 byte messages, 10-gigabit network

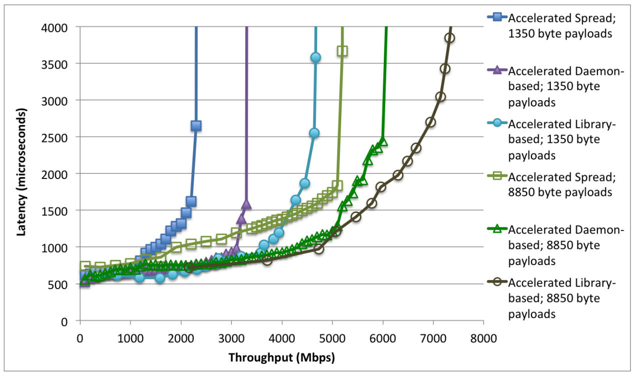

Fig. 6. Throughput vs Safe delivery latency for 1350 byte messages and 8850 byte messages, 10-gigabit network

1500 byte MTU, while allowing sufficient space for protocol headers. Spread requires large headers to support features required by users, such as descriptive group and sender names.

The results of the 1-gigabit experiments are shown in Figures 1 and 2. Figure 1 shows the relationship between throughput and latency for Agreed delivery, while Figure 2 does the same for Safe delivery. From these figures, we can see 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 microseconds, even for the lowest throughput level tested (100 Mbps). In contrast, with the Accelerated Ring protocol, Spread is able to reach 400 Mbps throughput with latency below 400 microseconds. At 900 Mbps, Spread's latency is under 1.2 milliseconds 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 milliseconds, before latency begins to climb rapidly. The accelerated protocol is able to support 800 Mbps with latency around 720 microseconds, simultaneously improving throughput by 60% and latency by over 45%.

We also see that the accelerated protocol improves maximum throughput compared to the original protocol. Using the accelerated protocol, Spread is able to reach over 920 Mbps. Since we measure only clean application data (payload), and Spread adds substantial headers to each message, this is practically saturating the 1-gigabit network.

Safe delivery shows a similar overall pattern. For all implementations, the original protocol is able to support up to 600 Mbps before latency begins to rise sharply, and latency is between 3.7 and 4.7 milliseconds at this point. The accelerated protocol supports 800 Mbps with latency around 2 milliseconds, improving throughput by over 30% and latency by over 45% at the same time. The accelerated protocol achieves throughput over 900 Mbps in all implementations 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, we see that Spread has distinctly higher latency than the prototypes when the original protocol is used. This is due to the fact that, 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 between Spread and the prototypes does not appear for Safe delivery, since delivering to clients is not on the critical path when that service is used. 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 3 and 5 show performance profiles from the same experiments shown in Figures 1 and 2, but on a 10-gigabit network. As in the 1-gigabit experiments, we can see the benefit of the Accelerated Ring protocol. 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 microseconds. Using the accelerated protocol, Spread can provide 1.2 Gbps throughput with an average latency of about 310 microseconds, for a simultaneous improvement of 20% in both throughput and latency. The maximum throughput Spread reaches with latency under 1 millisecond 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, we see that the differing overheads of the different implementations have a significant impact on performance. While Spread can support 1.2 Gbps throughput with average latency around 310 microseconds using the accelerated protocol, the daemon-based prototype supports up to 2.9 Gbps with the same latency, and the library-based prototype reaches 3.5 Gbps at that latency.

Because processing is a bottleneck for Spread on 10-gigabit networks, we consider the prototype implementations to see the full power of the protocol. For the daemon-based prototype, the original protocol supports 2 Gbps with latency around 390 microseconds. The accelerated protocol supports 2.8 Gbps throughput with latency around 265 microseconds, 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 microseconds; using the accelerated protocol, Spread can support the same throughput with 25% lower latency (about 700 microseconds), 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 microseconds for 1.2 Gbps throughput).

As in the results 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 millisecond latency, while the accelerated protocol supports 3.1 Gbps with 980 microsecond 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.

It is interesting to note that while the accelerated protocol always provides lower latency than the original protocol for the same throughput in 1-gigabit experiments or for Agreed delivery, Figure 7 shows that there is a slightly different trade-off on 10-gigabit networks for Safe delivery. When the system's aggregate throughput is very low, the original protocol provides better latency than the accelerated protocol. This is due the fact that 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, token rounds are already fast (since relatively few messages are being sent), so the accelerated protocol's ability to speed up each round is reduced, and the extra round becomes significant. At 100 Mbps, or 1% of the network's capacity, Spread's average latency is 620 microseconds with the accelerated protocol, which is close to 20% higher than the original protocol's latency of 520 microseconds. However, once throughput reaches 4-5% of the network's capacity (400-500 Mbps), the accelerated protocol consistently provides lower latency.

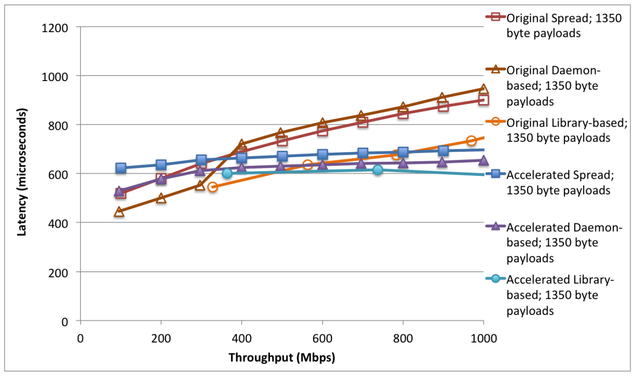

Fig. 7. Safe delivery latency for low throughputs, 10-gigabit network

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 single standard 1500 byte MTU; large messages are fragmented into multiple protocol packets. However, we can use UDP datagrams of up to 64 kilobytes, allowing message fragmentation and reassembly to be done at the kernel level, rather than at the daemon level, with the trade-off that the loss of a single frame results in the loss of the whole datagram.

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. However, 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 4 and 6 compare the performance profiles of the Accelerated Ring implementations using 1350 byte payloads (which are the same as in Figures 3 and 5) to the profiles when payloads of 8850 bytes are used. For both Agreed and Safe delivery, larger messages allow the protocol to reach considerably higher maximum throughputs. For Agreed delivery, Spread reaches 5.3 Gbps, compared with 2.1 Gbps using 1350 byte messages, for an improvement of 150%. The maximum throughput of the daemon-based prototype improves 87% from 3.2 Gbps to 6 Gbps, while that of the library-based protocol 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. As can be seen in Figure 6, the improvements are similar for Safe delivery.

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 on a 10-megabit network in a similar way to our library-based implementation 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, since these are limiting for many deployments, allowing larger UDP datagrams on 10-gigabit networks may be beneficial. A drawback of UDP datagrams that span multiple network frames is that the loss of a single frame results in the loss of the entire datagram. 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 price to pay 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.

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. discussed in Section I. 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. We highlight a few influential examples here.

The ISIS toolkit [16] 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.

In the Trans protocol [17], used by the Transis system [18], 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.

Ring Paxos [13] is based on the Paxos algorithm [19] and orders messages by executing a series of consensus instances to assign sequence numbers to messages. 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 [20] 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.

These alternative approaches have also been used to build other successful practical systems that can achieve good performance on modern networks. JGroups [21] 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 [22]. 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 [23], [24] 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 [25] 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 [25], which matches our measurements in the same 8-machine setup described in Section IV. When sending 1350 byte messages (but allowing batching), U-Ring Paxos reaches 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 technique requires additional processing resources.

It is important to note that while all of the protocols discussed here provide total ordering, there are subtle but meaningful differences in the semantics they provide. 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 cannot allow progress in multiple partitions. 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.

While sequencer-based approaches 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.

VI. Conclusion

We have 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. We show that 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 Accelerated Ring protocol is implemented in open-source prototypes suitable for research. A production implementation has been adopted as the default protocol for local area networks and data center environments in the Spread toolkit.

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 *Distributed Computing Systems, 1993., Proceedings the 13th International Conference on*, 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." [Online]. Available: http://www.spread.org

[5] "The Corosync cluster engine." [Online]. Available: http://corosync.github.io/corosync

[6] "Appia communication framework." [Online]. Available: http://appia.di.fc.ul.pt

[7] H. Miranda, A. Pinto, and L. Rodrigues, "Appia, a flexible protocol kernel supporting multiple coordinated channels," in *Distributed Computing Systems, 2001. 21st International Conference on.*, 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 *Distributed Computing Systems, 1994., Proceedings of the 14th International Conference on*, 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 *Proceedings of the Eleventh ACM Symposium on Operating Systems Principles*, ser. SOSP '87. New York, NY, USA: ACM, 1987, pp. 123–138.

[12] A. Babay, "The accelerated ring protocol: Ordered multicast for modern data centers," Master's thesis, Johns Hopkins University, May 2014.

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

[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 *Autonomous Decentralized Systems, 1995. Proceedings. ISADS 95., Second International Symposium on*, Apr 1995, pp. 215–221.

[16] 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.

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

[18] Y. Amir, D. Dolev, S. Kramer, and D. Malki, "Transis: a communication subsystem for high availability," in *Fault-Tolerant Computing, 1992. FTCS-22. Digest of Papers., Twenty-Second International Symposium on*, July 1992, pp. 76–84.

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

[20] P. Marandi, M. Primi, and F. Pedone, "Multi-ring paxos," in *Dependable Systems and Networks (DSN), 2012 42nd Annual IEEE/IFIP International Conference on*, June 2012, pp. 1–12.

[21] "JGroups - The JGroups Project." [Online]. Available: http://www.jgroups.org

[22] B. Ban, "Performance tests JGroups 2.6.4," August 2008. [Online]. Available: http://www.jgroups.org/perf/perf2008/Report.html

[23] K. Birman, "Isis2 Cloud Computing Library." [Online]. Available: http://isis2.codeplex.com

[24] K. Birman and H. Sohn, "Hosting dynamic data in the cloud with Isis2 and the Ida DHT," in *ACM Workshop on Timely Results in Operating Systems (TRIOS), at SOSP*, 2013.

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