Scalable Group Communication for Highly-Available Distributed Systems using Leader-Offload

Andrei Palade (0907350)

April 25, 2014

ABSTRACT

Reliable multicast is a powerful communication primitive for structuring distributed programs in which multiple processes must closely cooperate together. When the availability of these processes becomes a critical feature, one general approach is state machine replication which utilizes variants of leader based protocols. Due to high total service demand, the leaders become a bottleneck in the system. The throughput and latency values are limited by the leader's resources (e.g., CPU and network bandwidth), although resources are still available at other machines.

To overcome this shortcoming of leader-centric protocols, we present SEQUENCER_UUM and SEQUENCER_UUM2, two leader-centric protocols based on the Unicast-Unicast-M ulticast variant. SEQUENCER_UUM2 distributes the load between all the members of the group, achieves fast recovery from failure and improves the throughput of the network. Also, both SEQUENCER_UUM and SEQUENCER_UUM2 use resources efficiently. The system will use minimum of resources to perform a designated task. During high load, S EQUENCER_UUM2 achieves better utilization of resources and avoids becoming a bottleneck in the system.

1. INTRODUCTION

1.1 Availability

Total ordered reliable multicast is a powerful communication primitive for structuring distributed programs in which multiple processes must closely cooperate together. When the availability of these processes becomes a critical feature, one general approach is state machine replication[16]. By replicating a service on multiple machines, a guarantee is provided that even if one machine fails, the rest will be able to continue to provide this service without any interruptions. Replica coordination implies that all replicas (i.e., members that share the same state) receive and process the sequence of requests in the same relative order.

State machine replication utilizes variants of leader based protocols. Typically, these protocols are also leader-centric. This means that most of the work is done by an elected leader from the available set of machines. Therefore the leader becomes a bottleneck for the set. The throughput and latency values are limited by the leader's resources (e.g., CPU and network bandwidth), although resources are still available at other machines.

1.2 Motivation

Previous research has addressed the topic of an overloaded leader by distributing the work to a subset or all members of the set. While this approach has proved successful, the authors of these protocols have not considered the situation where the system is under load and the leader can handle the work. The experiments usually stress-test the system in order to analyse the throughput and latency. This method is not very efficient since in the real world the system is not being overloaded all the time. Also, previous work does not include an analysis of when the leader becomes a bottleneck.

As highly available systems become more widely used, reliable operation, scalability and improved performance will be required. Of primary importance are issues related to maintaining data consistency and coordinating activities of terminals in the network. These problems become more difficult when the system is asynchronous and the fault-tolerance and high availability requirements need to met.

Such problems are difficult to solve in a leader-centric environment because most of the work is delegated to the leader. Therefore the coordinator process (i.e., the leader) creates a bottleneck in the system. In this particular case, the system is limited by the leader's resources (i.e., network bandwidth and/or CPU), although there are still available resources that can be provided by the other members of the group.

1.3 Contributions

To overcome this shortcoming of leader-centric protocols, we present SEQUENCER_UUM and SEQUENCER_UUM 2.The purpose of these protocols is to be used when the resources are limited and there is a strong requirement for high availability. Despite random communication delays and failures, the protocol should maintain a replicated state of the application and allow fast recovery from failures. SEQ UENCER_UUM is based on an approach introduced by an algorithm that has proven successful in previous work[14, 3]. SEQUENCER_UUM2 is a variant of the SEQUENCER _UUM protocol. SEQUENCER_UUM2 distributes the load between all the members of the group, achieves fast recovery from failure and improves the throughput of the network. Also, both SEQUENCER_UUM and SEQUENCER_UUM 2 use resources efficiently. The system will use minimum of resources to perform a designated task. During high load, S EQUENCER_UUM2 achieves better utilization of resources and avoids becoming a bottleneck in the system.

1.4 Roadmap

In this paper, we will present the design, implemention and performance evaluation results of SEQUENCER_UUM and SEQUENCER_UUM2. The paper is organised as follows: Section 2 reviews related works in the area. Section 3 the design and implementation of SEQUENCER_UUM and S EQUENCER_UUM2. Section 5 presents the undertaken experiments and the results, while section 6 discusses and concludes the work.

2. BACKGROUND

2.1 Outline

The design and development of total order multicast primitives is one of the main research topics in distributed systems. Briefly, the total order multicast problem requires a group of distributed processes that need to reach an agreement on a common order of delivery, in presence of concurrent multicasts by any process of the group. The problem has inspired an abundance of literature and, over the past three decades, more than eighty algorithms[5] have been explored. Throughout the paper, the leader of the group will be refered to as the *Coordinator.* Table 1 presents the notation that will be used in this section.

M	set of all valid messages
Π	set of all processes (i.e., $\Pi = \{p_1,\ldots,p_n\}$)
$\Pi_{sequencer}$	set of all sequencer processes
Π_{sender}	set of all sending processes
Π_{dest}	set of all destination processes
$sender(m)$	sender of the message m
$Dest(m)$	set of destination process for message m
$seq(m)$	sequence number of message m
$\tau_{sequencer}$	type *Sequencer*
$\tau_{sequencer}(p)$	process of type *Sequencer*
ϕ	token

Table 1: Notation

2.2 Mechanisms for message ordering

In the absence of roadmap to the problem of total order multicast, *Défago et al.* proposed a classification of total order multicast algorithms. An important assumption made in this classification is the absence of failures. A process can have one of the three roles in an algorithm: sender, destination, or sequencer. A *sender* process is a process p_s that initiates a messages (i.e., $p_s \in \Pi_{sender}$). A *destination* process is a process p_d to which a message is destined (i.e., $p_d \in \Pi_{dest}$). The *sequencer* is the process that generates sequence numbers which are used for message ordering. At a certain point in time, a given process may simultaneously take several roles (e.g., sender *and* destination *and* sequencer). The identified roles are used as basic classes for total order multicast algorithms, depending whether the order is build by the sender, sequencer or destination processes. Further differences still remain between the identified classes. To address this problem a further division was introduced leading to five subclasses: fixed sequencer, moving sequencer, privilege-based, destination agreement and communication history.

Privilege-based algorithms rely on senders to multicast messages only when they are granted to do so. These algorithms do not adequate provide the scalability because a process must wait until it receives the privilege to multicast a message. Communication history algorithms are similar to privilege-based algorithms because the delivery order is determined by senders. In contrast to privilege-based algorithms, senders can multicast at any time. Communication history protocols are known for having low throughput because they rely on a quadratic number of messages to be exchanged for each message that is multicast[7]. Destination agreement algorithms ensure the delivery order from an agreement between destination processes. A large number of hybrid protocols[13, 6, 17] combining two different protocols have been proposed. Many of these protocols are optimized for very large scale networks, making use of multiple groups or optimistic strategies.

2.3 Sequencer-based total algorithms

In fixed sequencer algorithms, one process is elected as a sequencer and is responsible for coordinating the messages from sender process to destination process. There are three variants of these algorithms:

1. Unicast-Multicast (Figure 1(1)) consists of a unicast to the sequencer, followed by a multicast from the sequencer. This variant generates few messages[12].
2. Multicast-Multicast (Figure 1(2)) consists of a multicast to all destinations (including the sequencer), followed by a second multicast from the sequencer. While generates more messages that the Unicast-Multicas t, this variant reduces the load on the sequencer and makes the system easier to tolerate the crash of the sequencer. An example of this variant is ISIS[2].
3. Unicast-Unicast-Multicast (Figure 1(3)) consists of the request of a sequence number by the sender process (unicast). The sequencer replies with a sequence number(unicast). Then the sender multicast the sequenced message to the destination processes. *Armstrong et al.*[14] were the first to introduce this variant in MTP.

While each of the variants exhibit linear latency with respect to Π, they exhibit low throughput[7]. The coordinating process becomes a bottleneck because it receives the acknowledgements (ACKs) from Π_{dest} (i.e., all destination processes) and all the messages that need to be multicast from Π_{sender} (i.e., all sender processes). One limitation of fixed sequencer algorithms is that the ACKs can only be piggy-backed when all processes multicast at the same time[5]. High throughput and lower latency can be achieved by not requiring ACKs from Π_{dest}.

The S-Paxos protocol implemented by *Biely et al.*[1], which is based on Unicast-Unicast-Multicast variant (Figure 1(3)), showed better throughput and latency than Unicast-Multic ast (Figure 1(1)) implementation of Paxos protocol. *Biely et al.* also noted that the systems that use algorithms based on VariantUnicast-Unicast-Multicast are more scalable, fault-tolerant and achieve better throughput as Π increases. However, the experiments performed by *Biely et al.* used clients connected using persistent TCP connection. The Transmission Control Protocol offer an extra layer of reliability in terms of data transmissions. While the results are promising

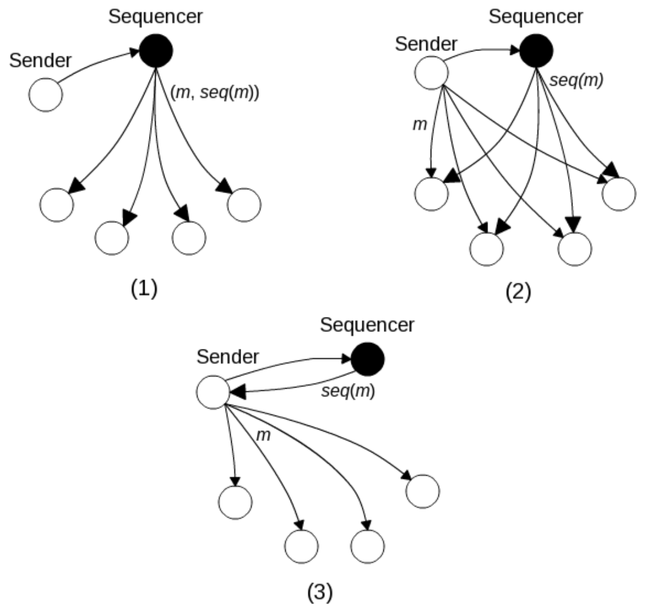

Figure 1: (1) Variant Unicast-Multicast; (2) Variant Multicast-Multicast; (3) Variant Unicast-Unicast-Multicast

with respect of offloading the sequencer in a Fixed Sequencer based algorithm, it is questionable whether the Unicast-Unicast-Multicast variant achieves the same results when the clients are connected using unreliable UDP connections.

MTP[14] is another protocol that implements the Unicast-Unicast-Multicast variant. This protocol provides reliable globally ordered delivery of messages among a group of communicating processes. MTP is built on top of IP multicasting and passes tokens to ensure the data is delivered reliably and in order. Previous implementations of token passing scheme have reduce the stability time and reduced the buffer size[10]. Also, this protocol does not ensure a leader recovery in case of failure. While the master recovery problem has been fixed in MTP-2[3], the implementation of the second variant still relies on a token-passing scheme.

Moving sequencer algorithms are based on the same principle as fixed sequencer protocols. Most of the moving sequencer algorithms are a variation of the concept of a logical ring along which a token is passed used by *Chang et al.*[4]. Pinwheel eliminates timestamps used for ordering the messages in exchange for a uniform message arrival pattern. *Kim et al.*[10] remove the logical ring topology completely and they introduce a token-passing scheme for a faster detection for message stability. These algorithms tolerate message loss by relying on a message retransmission protocol based on acknowledgements. Token-based atomic multicast algorithms are extremely efficient because they manage to reduce the network contention. However, the token method introduces an amount of overhead. *Mao et al.*[11] avoid the this method and they introduce the concept of distributing the role of the sequencer in round-robin fashion amongst the rest of the processes. This approach is successful in offloading the leader, but it suffers from poor latency. To address this problem, *Kapritsos et al.*[9] assigns each consensus instance to different “virtual clusters” instead of a different leader. The drawback of this approach is that the inter cluster communication makes the physical replica a bottleneck.

These algorithms are usually implemented in group communication toolkits which are an effective tool for building highly available reliable systems through software replication. They provide a rich set of primitives which help developers to implement any of the algorithms described above. One such reliable group communication toolkit is JGroups[1][15]. This toolkit is written in Java and builds on fundamental concepts developed in ISIS protocol[2]. The entire toolkit can be considered an API because it can be easily extended to be integrated with any group communication system. JGroups is based on a protocol stack. This provides the flexibility necessary for developers to add their own protocols by implementing the *Protocol* interface. The application and the core system run in the same process. An important advantage of JGroups is that it is heavily based on design patterns. The developers can add new protocols very easily only by implementing the *Protocol* interface. This toolkit implements SEQUENCER[2] which is based on a Unicast-Multicast variant (Figure 1(1)) of the Fixed Sequencer class. The design of this protocol is presented in section 2.4. This protocol will be referred to as the SEQUENCER protocol throughout this paper.

2.4 SEQUENCER

The SEQUENCER protocol provides total order for multicast messages by forwarding messages to the current coordinator, which then sends the messages to the cluster on behalf of the original sender. The total order is established because it is always the same sender (whose messages are delivered in FIFO order). Sending members add every forwarded message M to a buffer and remove M when they receive it. Should the current coordinator crash, all buffered messages are forwarded to the new coordinator.

3. IMPLEMENTATION

This section presents the design and implementation details of SEQUENCER_UUM2 builds on SEQUENCER_UUM. Both protocols implement the Unicast-Unicast-Multicast variant.

The SEQUENCER_UUM2 builds on SEQUENCER_UUM and innovates through allocation of a range of sequence numbers instead of a single number, whereas the SEQUENCER_UUM must perform the request-reply cycly for each message before multicast. When a sender makes a request to the *Coordinator*, counts the number of accumulated messages in the buffer and sends this number to the *Coordinator* as part of a request. When the *Coordinator* receives the request, it increments the local sequencer number with the number of requests sent by the sending process and then replies to the original sender of the message.

The SEQUENCER_UUM and SEQUENCER_UUM2 protocols are built on top of JGroups protocol. When the *Coordinator* crashes, other group members will run a reconciliation protocol to elect a new leader. Normally, the next process in line will take the *Coordinator* role. Another difference between the two proposed protcols is the way the view change is handled. In case of SEQUENCER_UUM2,

[1]`http://www.jgroups.org`
[2]`https://github.com/belaban/JGroups/blob/master/doc/design/SEQUENCER.txt`

the other processes will unicast to the newly elected *Coordinator* their highest sequence number received, whereas in SEQUENCER_UUM each of the sender needs to resend the requests accumulated in the buffer. Both protocols improve the availability of the system, but the SEQUENCER_UUM 2 is slightly faster. For example, in a system with 2 sending processes and a *Coordinator* if the *Coordinator* crashes, the two processes need to send only two messages. Whereas in case of SEQUENCER_UUM, each of the senders can have n messages accumulated in their buffer. As a conclusion, SE QUENCER_UUM2 improves the availability of the system. Because each of the sender can request a range of sequence numbers, this will ensure fairness among the senders. The *Coordinator* can not broadcast messages directly, but it has to execute the request-reply cycle. Also, another advantage of the two proposed protocols compared to SEQUENCER, the data is only sent and marshalled once (better for large messages). This will lower the CPU usage.

As a result of this work, the SEQUENCER_UUM2[3] has been included in the JGroups project which is maintainted by a consortium of developers from RedHat. The SEQUEN CER_UUM2 will appear in protocol stack of JGroups after next major release.

4. PERFORMANCE ANALYSIS

In the Unicast-Multicast variants the component with the highest total service demand is the *Coordinator*. All the senders need to forward their messages to the *Coordinator* which will TO-multicast the messages on behalf of the original sender. Due to this procedure, the *Coordinator* becomes a bottleneck. The leader becomes the key limiting factor in achieving higher throughput. Improving this procedure will provide the highest payoff in terms of throughput of the system.

The focus of this section will be on analysing the response time of the *Coordinator* and the improvements that can be made to achieve better processing time and ultimately increase the throughput of the system. The performance will be measured by comparing the analytical results to performance measurements. Table 2 introduces the notation used in this section.

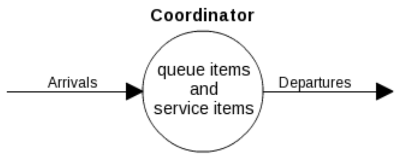

Figure 2: Black-box view of the system

The *Coordinator* behaves like any other queueing system (Figure 2). To analyse the properties of this system, we will treat the *Coordinator* as a FIFO M/M/1 queueing model (Poisson arrival processes(M), exponential service time(M) and single server(1)). This model assumes that both arrivals and service times are exponentially distributed with a single server. The model consists of two elements: the queue and the server. The state of the system is defined as the total number of elements in the system. In this case, it will be the total number of messages in the *Coordinator*.

λ	Mean arrival rate
μ	Mean service rate
ρ	Traffic intensity (Utilization)

Table 2: Notation

The utization of the *Coordinator* is given by:

$$\rho = \frac{\lambda}{\mu} \tag{1}$$

The mean number of the messages in the system is:

$$E[n] = \frac{\rho}{1-\rho} \tag{2}$$

The mean number of the messages in the queue is:

$$E[n_q] = \frac{\rho^2}{1-\rho} \tag{3}$$

The Stability condition results from equation 3. The traffic intensity ρ must be less than 1, otherwise the number of messages in the system becomes very large. When this happens the *Coordinator* will always be busy, leading to a queue build up. Conversely, if ρ approaches 1 then the *Coordinator* experiences a queue build up. This directly affects the response time of the *Coordinator*. The response time is:

$$E[r] = \frac{\frac{1}{\mu}}{1-\rho} \tag{4}$$

From the equation 4 it can be deducted that ρ must be less than 1, otherwise the response time approaches infinity. Since the *Coordinator* has highest total service demand, by measuring the response time with respect to utilization of the system, we will be able to outline in which cases the *Coordinator* becomes a bottleneck.

5. EVALUATION

5.1 Outline

This section compares the performance of SEQUENCER _UUM and SEQUENCER_UUM2 to the performance of SEQUENCER protocol, which is currently implemented in JGroups. Two types of performance metrics are considered: individual and global. Individual metrics aim to reflect the utility of each machine, while global metrics reflect the systemwide utility[8]. The purpose of this selection is to measure the performance of the *Coordinator* individually and to be able to prove that the improvements made in the *Coordinator* are available system-wide. In section 5.3 we measure the ration between the response time and utilization of the *Coordinator* for all three protocols. Section 5.4, 5.5 and 5.6 present the results of throughput, end-to-end delay, fairness and CPU usage from a system perspective.

5.2 Experimental Setup

The measurements were performed on a network of machines with a Intel$^{(R)}$ Core$^{(TM)}$ i5-3470 CPU @ 3.20GHz processor and 4GB RAM. Machines run Linux the 2.6.32 - 1 SMP kernel and are connected using a Fast Ethernet switch. The Java environment used is OpenJDK 1.7.0_55. The SEQUENCER, SEQUENCER_UUM and SEQUENCER_UUM2

[3]`https://github.com/belaban/JGroups/blob/JGRP-1821/src/org/jgroups/protocols/SEQUENCER_UUM.java`

algorithms have been implemented in JGroups v3.4.3-Final. The transport protocol will be UDP over IP Multicast. To utilize the TO primitives we defined set of stack configurations:

1. `UDP:PING:MERGE2:FD_SOCK:FD_ALL:VERIFY_SUSPECT:BARRIER:pbcast.NAKACK2:UNICAST3:pbcast.STABLE:pbcast.GMS:UFC:MFC:SEQUENCER:FRAG2:pbcast.STATE_TRANSFER`
2. `UDP:PING:MERGE2:FD_SOCK:FD_ALL:VERIFY_SUSPECT:BARRIER:pbcast.NAKACK2:UNICAST3:pbcast.STABLE:pbcast.GMS:UFC:MFC:SEQUENCER_UUM:FRAG2:pbcast.STATE_TRANSFER`
3. `UDP:PING:MERGE2:FD_SOCK:FD_ALL:VERIFY_SUSPECT:BARRIER:pbcast.NAKACK2:UNICAST3:pbcast.STABLE:pbcast.GMS:UFC:MFC:SEQUENCER_UUM2:FRAG2:pbcast.STATE_TRANSFER`

The experiments presented in this section start with a warm-up phase, followed by the measurement phase. In the warm-up phase the experiment that will be used for measurements is executed 5 times. During this phase no results are collected. In the measurements phase the experiements will be executed 20 times and the results logged.

5.3 Response time/Utilization

Figure 3 shows the response time as a function of the utilization at the *Coordinator*. As the rate increases, the utilization approaches 1 and the number of jobs in the system and response time approaches infinity. In SEQUENCER, the *Coordinator* reaches saturation when the response time is 250 milliseconds and utilization is very close to 100%. Therefore when the link utilization approaches 100%, the response time approaches infinity and the *Coordinator* becomes a bottleneck in the system. In case of SEQUENCER_UUM, as the rate increases, the utilization slighly improves. Figure 4 shows that as the rate increases in the system, at 250 milliseconds the utilization is between 90% and 100%. In SEQUENCER_UUM2 (Figure 5), the utilization is capped at 90% as the rate increases. A noticeable difference between the utilization of SEQUENCER_UUM2 and the other protocols is that, SEQUENCER_UUM2 reaches this utilization rate at a much lower rate because the number of messages in the system is considerably lower than the other two protocols. This is due to the range allocation during a request-reply cycle compared to the work to be undertaken in SEQUENCER_UUM or SEQUENCER case.

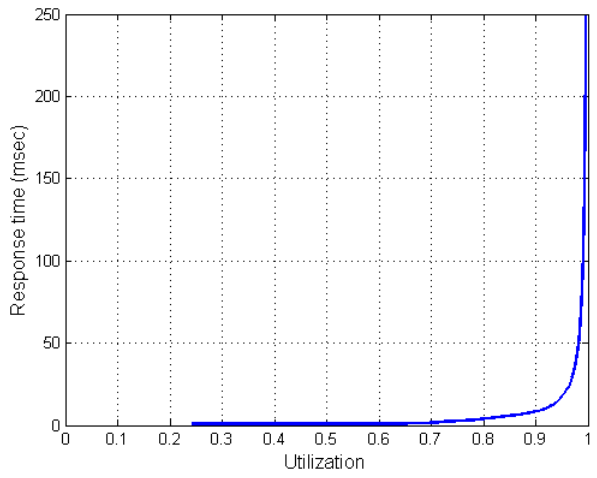

Figure 3: *Coordinator* response time as a function of utilization in SEQUENCER.

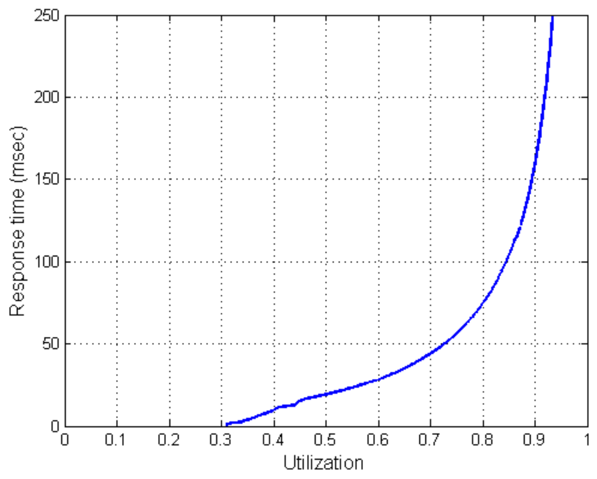

Figure 4: *Coordinator* response time as a function of utilization in SEQUENCER_UUM.

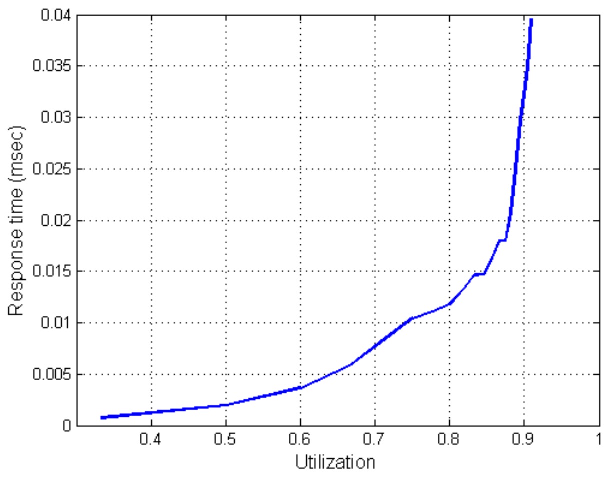

Figure 5: *Coordinator* response time as a function of utilization in SEQUENCER_UUM2.

5.4 Throughput

To assess the throughput and the end-to-end delay of the discussed protocols, the following experiment has been proposed: for every presented protocols, n processes in the system multicast messages. At the end of the experiment, the throughput of the SEQUENCER protocol is chosen as a base to compute the ratio of throughput(SEQUENCER_UUM)/throughput(SEQUENCER) and ratio of throughput(SEQUENCER_UUM2)/throughput(SEQUENCER). The ratio is made on the average values computed by each process. The parameters of this experiment are shown in Table 3.

The results of the experiment can be observed in Figure 6. The Y axis represents the ratio of measured throughputs. The X axis represents the number of senders in the group. Each sender produces messages at the maximum throughput it can sustain.

As the number of senders increases, the throughput of the SEQUENCER degradeds slightly due to the fact that all the senders send their messages through the *Coordinator*. In the SEQUENCER_UUM protocol, the throughput slightly improves, considering the extra message request done by the senders.

In case of the SEQUENCER_UUM2, the throughput improves as the number of senders increases because the number of requests made to the *Coordinator* is lower than in SEQUENCER_UUM. This is means that the senders can request a range of sequence numbers from the coordinator and multicast directly the accumulated messages.

Physical nodes	10
Processes	10
Senders	10
Message size	1Kb
Burst	10000 messages

Table 3: Experiment parameters used to measure the throughput and the end-to-end of the system.

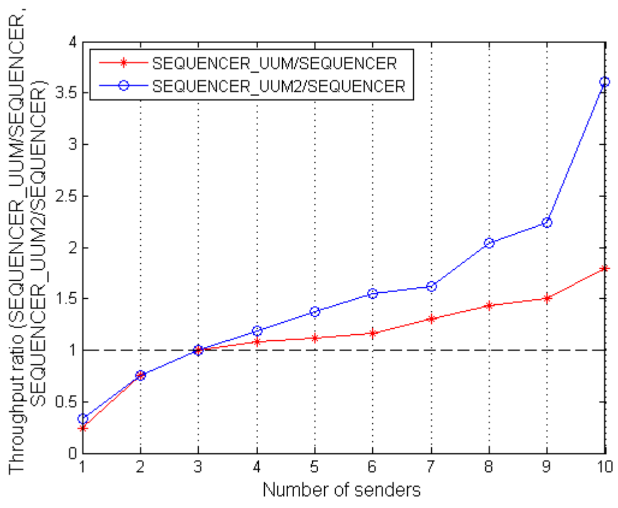

Figure 6: Ratio of system (sender) throughputs for SEQUENCER_UUM/SEQUENCER and SEQUENCER_UUM2/SEQUENCER. Table 3 contains the parameters used in these experiments.

Figure 7 presents the End-to-End delay for all three protocols. The Y axis represents the average delay in milliseconds when sending a message. The X axis represents the number of senders in the group. As the number of sender increases the delay is expected to increase as well. However, both SEQ UENCER_UUM and SEQUENCER_UUM2 are faster than SEQUENCER. For example, the average delay for seding a message in a group with 10 senders is, compared to the SEQUENCER, 1.25 faster in case of the SEQUENCER_UU M and 2.25 better for SEQUENCER_UUM2.

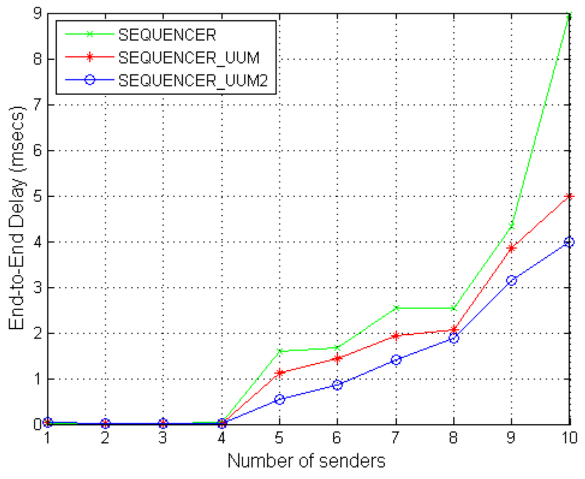

Figure 7: Delay assessment of the SEQUENCER, SEQUENCER_UUM and SEQUENCER_UUM2. Table 3 contains the parameters used in these experiments.

5.5 Fairness

Physical nodes	3
Processes	3
Senders	3
Message size	1Kb
Burst	10000 messages

Table 4: Experiment parameters used to assess the fairness of *Coordinator* vs. other senders in the group.

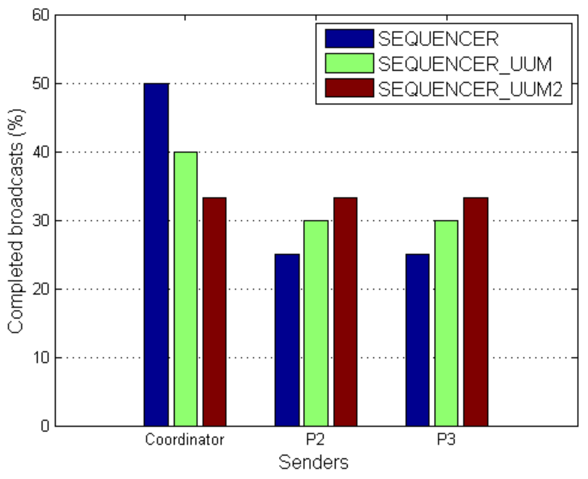

Figure 8: Fairness assessment of the SEQUENCER, SEQUENCER_UUM and SEQUENCER_UUM2. Experiments were performed with 3 processes and 3 senders. Table 4 contains the parameters used in these experiments.

From the Figure 8 we outline a set of observations. The SEQUENCER protocol that is currently implemented in

JGroups is not fair. The *Coordinator* can directly multicast the messages that it produces, whereas the processes P_2 and P_3 need to send their messages to the *Coordinator* to be multicast. This procedure induces a significant imbalance: at the end of the measurement phase, 50% of the messages that are delivered have been multicast by the *Coordinator*. The processes P_2 and P_3 have each multicast only 25% of the delivered messages.

The SEQUENCER_UUM protocol improves the fairness by introducing the request-reply cycle of a sequence number for each message before multicast. All processes in the group, including the *Coordinator*, must perform this procedure before broadcasting a message. While the request-reply improves the fairness, the senders P_2 and P_3 must request a sequence number from *Coordinator* for each message they need to multicast. At the end of the measurement phase, 40% of the delivered message delivered have been multicast by the *Coordinator*.

SEQUENCER_UUM2 ensures fairness between all the processes in the group. The results show that all the processes have equally multicast about the same number of message. This is due to the fairness mechanism described in Section 3 where each of the senders requests a range of sequence numbers based on the number of accumulated messages in the buffer of the sender.

5.6 CPU Usage

The last performance metric to evaluate is the CPU Usage of JGroups process using, by turn, the SEQUENCER, SEQUENCER_UUM and SEQUENCER_UUM2 protocols. During the experiment, the CPU Usage of the JGroups process was logged every 0.1 seconds. The collected data was added up and averaged. This procedure has been performed for two parties: *Coordinator* and the other processes. The setup parameters are presented in the Table 5.

Physical nodes	5
Processes	5
Senders	5
Message size	100Kb
Burst	10000 messages

Table 5: Experiment parameters used during the CPU Usage measurements.

The experiment in Figure 9 plots the CPU usage measured in a system with 5 processes, each on a separate physical node. The X axis represents the protocols used in each case. The Y axis represents the CPU consumption (in %).
From the Figure 9 the first remark made is that among the three protocols, the SEQUENCER variant induces the highest CPU consumption. In case of the SEQUENCER, the *Coordinator* used nearly twice the CPU usage compared with the other sending processes in all the performed experiements. This is due to the multicast of all messages forwarded by the other senders. Another relevant factor that contributes to high CPU usage by the senders is the marshalling of messages. When an application wants to pass its memory objects across a network to another host or persist it to storage, the in-memory representation must be converted to a suitable format. This process is called marshalling and the revert operation is called demarshalling. The *Coordinator* encapsulates a forward message in its own message

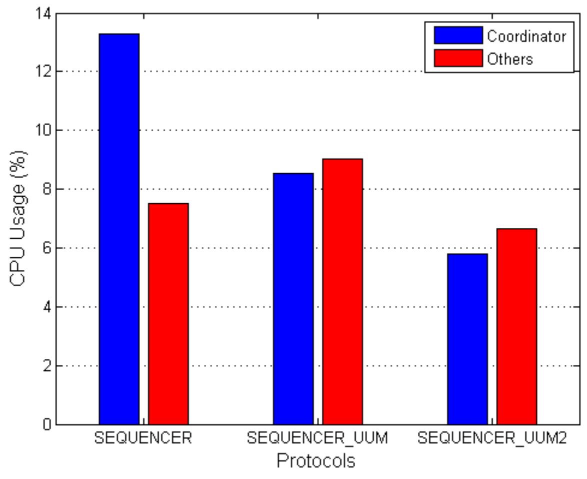

Figure 9: CPU usage during high load n-to-n multicast of the SEQUENCER, SEQUENCER_UUM and SEQUENCER_UUM2 protocols. Table 5 contains the parameters used during the experiments.

before multicast.

In the SEQUENCER_UUM case, the CPU usage of the *Coordinator* is reduced to an average of 8.52% while the rest of the senders average to 9%. The improvement is because the *Coordinator* does not multicast the messages received from the senders. Now, each sender utilizes the request-reply cycle before broadcasting its messages.

The SEQUENCER_UUM2 reduces the CPU usage even further. In this environment, the *Coordinator* utilizes 5.8% and the senders use 6.7% of the CPU time. This is reduction in CPU usage is due to the elimination of the request-reply cycle for each message that the sender need to broadcast. Having less requests and replies made to the *Coordinator*, the CPU usage is reduced on all the nodes in the group.

6. DISCUSSION

In this paper, we have examined an approach for providing scalable group communication for systems that need high availability by using reliable, scalable multicast communication. When the availability of these systems becomes a critical feature, one general approach is state machine replication which utilizes variants of leader based protocols. Typically, these protocols are also leader-centric. Since the leader is the resource with the highest utilization, it becomes a bottleneck for the set. The throughput and latency values are limited by the leader's resources (e.g., CPU and network bandwidth), although resources are still available at other machines. Performance optimizations at this resource offer the highest payoff.

To overcome this shortcoming, we implemented two leader-centric protocols: SEQUENCER_UUM and SEQUENCER _UUM2. Both are based on the Unicast-Unicast-Multicas t variant. When TO-multicast a set of messages, the SEQUENCER_UMM performs the request-reply cycle for each message m in the set. The SEQUENCER_UUM2 optimizes this process and requests a range of sequence numbers from the *Coordinator*. We compared these two protocols with the SEQUENCER protocol. SEQUENCER implements a Unic ast-Multicast variant and is currently the only TO primitive in the JGroups stack.

The purpose of the comparison was to measure utilization of the *Coordinator* and conclude when the leader becomes a bottleneck in the system. Thus this was the most important part of performance evaluation. Using perfomance analysis we modeled the *Coordinator* as queueing system using a M/M/1 queue. Using measurements we confirm the results of the analysis. The utilization of the *SEQUENCER* approaches 100% and it becomes a bottleneck in the system.

SEQUENCER_UUM2 distributes the load between all the members of the group, achieves fast recovery from failure and improves the throughput of the network. Also, both SE QUENCER_UUM and SEQUENCER_UUM2 use resources efficiently. The system utilizes minimum of resources to perform a designated task. During high load, SEQUENCER_ UUM2 will achieve a better utilization of resources and will avoid becoming a bottleneck in the system.

7. FUTURE WORK

The research undertaken as part of this study has raised several interesting points that should be considered as future work. Here, we address these points, discussing why undertaking them would be beneficial and, were is the case, provide some key challenges that one would need to oversee.

Future work should include reproducing the tests used in this paper on a cluster larger than 10 (physical) nodes. It is important to observe the scalability of SEQUENCE R_UUM2 on a larger scale. Another interesting research challenge is to evaluate how the variants Unicast-Multicast and Unicast-Unicast-Multicast with multiple coordinators change the performance of the system. This implies using a moving sequencer, where each coordinator is running each of Fixed Sequecer variants. When the *Coordinator* reaches a certain utilization limit, creates another *Coordinator* to reply to the new incoming requests made by the sending processes in the group. The initial coordinator will only TO-multicast messages left in the queue. This approach generates a number of challenges: notify the sender to stop unicasting messages to the old *Coordinator*; notify all the processes in the view about the new coordinator and transfer the sequence number from old *Coordinator* to the new *Coordinator*.

8. REFERENCES

[1] M. Biely, Z. Milosevic, N. Santos, and A. Schiper. S-paxos: Offloading the leader for high throughput state machine replication. In *Reliable Distributed Systems (SRDS), 2012 IEEE 31st Symposium on*, pages 111–120, 2012.

[2] K. Birman, A. Schiper, and P. Stephenson. Lightweight causal and atomic group multicast. *ACM Trans. Comput. Syst.*, 9(3):272–314, Aug. 1991.

[3] C. Bormann, J. Ott, H.-C. Gehrcke, T. Kerschat, N. Seifert, and N. Seifert. Mtp-2: Towards achieving the s.e.r.o. properties for multicast transport. In *In International Conference on Computer Communications Networks*, 1994.

[4] J.-M. Chang and N. F. Maxemchuk. Reliable broadcast protocols. *ACM Trans. Comput. Syst.*, 2(3):251–273, Aug. 1984.

[5] X. Défago, A. Schiper, and P. Urbán. Total order broadcast and multicast algorithms: Taxonomy and survey. *ACM Comput. Surv.*, 36(4):372–421, Dec. 2004.

[6] P. Ezhilchelvan, R. Macedo, and S. Shrivastava. Newtop: a fault-tolerant group communication protocol. In *Distributed Computing Systems, 1995., Proceedings of the 15th International Conference on*, pages 296–306, 1995.

[7] R. Guerraoui, R. R. Levy, B. Pochon, and V. Quéma. Throughput optimal total order broadcast for cluster environments. *ACM Trans. Comput. Syst.*, 28(2):5:1–5:32, July 2010.

[8] R. Jain. *The art of computer system performance analysis: techniques for experimental design, measurement, simulation and modeling.* New York: John Willey, 1991.

[9] M. Kapritsos and F. P. Junqueira. Scalable agreement: Toward ordering as a service. In *Proceedings of the Sixth International Conference on Hot Topics in System Dependability*, HotDep'10, pages 1–8, Berkeley, CA, USA, 2010. USENIX Association.

[10] J. Kim and C. Kim. A total ordering protocol using a dynamic token-passing scheme. *Distributed Systems Engineering*, 4(2):87, 1997.

[11] Y. Mao, F. P. Junqueira, and K. Marzullo. Mencius: Building efficient replicated state machines for wans. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*, OSDI'08, pages 369–384, Berkeley, CA, USA, 2008. USENIX Association.

[12] S. Navaratnam, S. Chanson, and G. Neufeld. Reliable group communication in distributed systems. In *Distributed Computing Systems, 1988., 8th International Conference on*, pages 439–446, 1988.

[13] L. Rodrigues, H. Fonseca, and P. Verissimo. Totally ordered multicast in large-scale systems. In *Distributed Computing Systems, 1996., Proceedings of the 16th International Conference on*, pages 503–510, 1996.

[14] K. M. S. Armstrong, A. Freier. *Multicast transport protocol.* RFC 1301 IETF, 1992.

[15] L. Sales, H. Teofilo, J. D'Orleans, N. Mendonca, R. Barbosa, and F. Trinta. Performance impact analysis of two generic group communication apis. In *Computer Software and Applications Conference, 2009. COMPSAC '09. 33rd Annual IEEE International*, volume 2, pages 148–153, 2009.

[16] F. B. Schneider. Implementing fault-tolerant services using the state machine approach: A tutorial. *ACM Comput. Surv.*, 22(4):299–319, Dec. 1990.

[17] P. Vicente and L. Rodrigues. An indulgent uniform total order algorithm with optimistic delivery. In *Reliable Distributed Systems, 2002. Proceedings. 21st IEEE Symposium on*, pages 92–101, 2002.