Multi-Paxos: An Implementation and Evaluation

Hao Du[*] *David J. St. Hilaire*[*]

Abstract

We implemented a fully functional sequential Multi-Paxos system and a prototype parallel Multi-Paxos system. The throughput of the system under various settings of the Paxos cluster size, network latency, and window size (of the parallel Paxos) were evaluated, giving us insight into how and when these factors affect the performance. Additonally, despite existing methods for electing distinguished proposer, we proposed and evaluated a different simple yet effective approach to determine the distinguished proposer.

1 Introduction

Paxos is a distributed system consensus algorithm that was described in Lamport's paper, *The Part-Time Parliament*. It was our goal to implement and evaluate a real implementation of a Paxos based system because while Paxos is a well-defined and understood algorithm, creating an implementation for the real world is not straightforward. Through this project, we desired to garner a better understanding of Paxos as a whole and to understand some of the issues and limitations facing real implementations. In the process of implementing Paxos based systems, we considered two open issues facing Paxos in practice: liveness and throughput. Because of the FLP impossibility proof [2], Paxos cannot guarantee liveness and safety at the same time, so it guarantees safety and gives suggestions on how to achieve liveness through a distinguished proposer. We considered distinguished proposer implementations, developed our own, and evaluated it to ensure that it performs adequately in practice. The issue with throughput is that any useful system needs to be able to efficiently implement multiple iterations of Paxos to implement any kind of state. To understand the throughput issue better, we built a full sequential Paxos system and a prototype parallel Paxos system.

Before continuing, it may be useful to refresh a few of the main ideas of the Paxos algorithm to ensure consistent terminology. A Paxos system is made up of a collection of nodes that each has to 1 to 3 different roles. These roles are Proposer, Acceptor, and Learner. When attempting to make proposal, a Proposer begins in the Prepare phase where it sends a Prepare message to the Acceptor nodes. This message contains the Paxos iteration number and the proposal number. The Acceptor can respond with a rejection or with a Promise. A Promise tells the Proposer that the Acceptor will not accept a Proposal with a lower proposal number. A Promise also contains the last proposal value that the Acceptor accepted. If the proposer receives a Promise from a majority of the Acceptors, it can move to the Propose phase. In this phase, the Proposer sends a Proposal message containing the Paxos iteration number, the proposal number, and the proposal value (this is either the value returned in the Prepare phase or a value of the Proposer's choosing if no value was obtained in the Prepare phase) to the Acceptors. If the Acceptors have not promised to only accept higher proposal numbers, they can send an Accept message to the Proposer. If the Proposer receives an Accept message from a majority of the Acceptors, it will know that the value has been decided for that iteration of Paxos.

2 Implementation

We designed our system with the original intent of running it on the Seattle platform. Since we were limited on the number of machines that we could acquire, we decided to implement each Paxos node as a combined Proposer, Acceptor, and Learner. We also decided to use UDP instead of TCP because originally we were not sure how many connections we would be allowed.

2.1 How the system works a high level

In our implementation, the Paxos system is used as a storage system. Clients have the ability to read the value of a variable or to write a new value to a variable. However, before a client can do this, it must find the distinguished proposer. Since the client may not know which Paxos node is the distinguished proposer, it will send a session request to some arbitrary Paxos node that it knows. If this Paxos node is the distinguished proposer, it will respond with a session accept message containing a client id and sequence number for the client. The sequence number allows the distinguished proposer to differentiate between multiple requests that appear the same and a single request that is received multiple times. If the Paxos node is not the distinguished proposer, it will respond with a session reject message with contains the IP address and port of the Paxos node that it believes is the current distinguished proposer.

Once a client has successfully established a session, it can start sending Paxos requests to the distinguished proposer. The distinguished proposer will respond to let

[*]Dept. of Computer Science and Engineering, Univ. of Washington.

the client know that the request has been received and is being processed. Once the request has been successfully completed, the client will receive a message with the response. If a node loses distinguished proposer status, it will notify the clients by telling that their Paxos requests have failed. The clients can then establish a new session with the new distinguished proposer and send it their requests.

2.2 Original Multi-Paxos: Sequential

The Paxos algorithm describes how to come to a consensus within a distributed system. However to be useful, a distributed system needs to be able to implement multiple sessions of Paxos that sequentially tie together to form the state of the system. We will follow Google's terminology and refer to this as Multi-Paxos [1].

Our plan to create a Multi-Paxos system was to implement completely separate sequential Paxos iterations. All Paxos nodes would defer to the distinguished proposer, who would make all proposals. A distinguished proposer would progress from one Paxos iteration to the next Paxos iteration in sequential order. While the distinguished proposer had uncompleted client requests, it would execute the Prepare phase on the Paxos iteration it believed to be the next empty iteration. In the Propose phase if less than a majority of acceptors responded in the Prepare phase that no value has been proposed, the distinguished proposer would propose the value given by the acceptors. Otherwise, it would propose the next client request in its queue as the value. Once it had successfully obtained accept messages from a majority of the acceptor nodes, it would act as the distinguished learner and broadcast to all the nodes the decided value for that iteration.

Since our Paxos system acted as the data storage system itself, the distinguished proposer would respond with the state after implementing the client requests. Since the distinguished proposer does not need to know the decided state of earlier iterations before completing later iterations (for example if a different node was the distinguished proposer for that iteration and the current distinguished proposer had some holes in its knowledge), it may not respond to client requests directly after completing them. Rather the distinguished proposer would learn the decided values for the holes in its knowledge, while processing new requests. It is safe to continue new iterations while learning old ones because every iteration is a completely separate instance of Paxos. As holes were filled, the node would implement all decided values up to next knowledge hole. Once the holes were filled, the master would know the correct state for each iteration and could send responses to the client.

2.2.1 Distinguished Proposer

Paxos as a system provides fault tolerance and safety. However, it does not guarantee liveness because more than one Paxos node may attempt to make a proposal at the same time. Lamport suggested that a Paxos system implement the idea of a distinguished proposer (or president) to solve this issue [3]. All proposer nodes will defer proposals to the distinguished proposer to minimize contention in the Paxos system. Once a single distinguished proposer has been selected, the system is guaranteed to make progress. However selecting a single distinguished proposer is an instance of the FLP problem.

In his original paper, one suggestion that Lamport made was a fixed order for the distinguished proposer that was based on the nodes "name" [3]. Another option is to elect the distinguished proposer via a Paxos iteration. One issue with the first suggestion is that if the node with the best "name has a very high latency or fails frequently, there often be situations where there is no consensus on who is the distinguished proposer. The second suggestion fixes this problem because over time a more stable node is more likely to be the distinguished proposer. However, until nodes learn the most current values, they will not know who the current distinguished proposer is.

Thus we proposed a slightly different distinguished proposer election than directly through Paxos to simplify the logic and to speed up the consensus time. During normal operations, the node to make the last successful proposal is considered the distinguished proposer. In our system, one node is both the distinguished proposer and the distinguished learner (the distinguished learner is responsible for learning the decided values for the Paxos iterations). As the distinguished proposer makes successful proposals, it broadcasts the decision to all other nodes. If a node realizes that it is missing information about a Paxos iteration, it queries the distinguished proposer for information.

When every node starts, it contains a list of all the other nodes in the system. It considers the first node in the list to be the distinguished proposer at this time. Every node periodically sends a message to the node it considers the distinguished proposer asking who it considers to be the distinguished proposer. If the response is a different node than the responding node, the questioner updates whom it considers the distinguished proposer to be this new node. If the distinguished proposer does not respond for a specific period of time, it considers the distinguished proposer to be failed. At this point, it will attempt to deal with client requests itself instead of forwarding them to the distinguished proposer.

If multiple Paxos nodes are attempting to become the distinguished proposer, there is a good chance that there will be conflict before one or more nodes successfully

complete a proposal. Following the Paxos algorithm guarantees that there is no danger of violating safety; however Paxos nodes will receive reject messages because another node used a higher proposal number. Upon receiving a reject message, the Paxos will do an additively increasing (for each rejected attempt) random backoff. If after the backoff time has expired it has not learned of a new distinguished proposer, it will increase its proposal number to be higher than the number that caused its rejection. Since only one of the nodes attempting to be the distinguished proposer will not receive a rejection message (because it was using the highest proposal number), it should be the only one attempting to make a proposal while the other nodes wait for their backoff timer to expire. If this node fails or is unable to communicate with some Paxos nodes, the random component of the backoff should space out when the other nodes retry, making it easier for one node to successfully complete a proposal and inform the other nodes that it is the distinguished proposer. It is important to increase the backoff with each rejection to ensure that a node has time to finish a proposal (if network latency varies or Paxos nodes become overloaded, the time to finish a proposal will also varies).

2.3 Multi-Paxos Take Two: Parallel

After analyzing our results for the sequential Paxos, we noted that some of the bottleneck was the extra overhead of implementing the storage system within the Paxos node. We knew that real life Paxos systems such as Google's [1] used a parallel Paxos implementation to increase the throughput. Thus we hypothesized that if we removed the excess from the implementation so that it did not have to perform the extra processing, then our bottleneck would become the delay due to the round trip time. Our desire was to see how much of a throughput increase could be gained by using a parallel implementation with large window sizes.

We considered two different options for parallelizing our Paxos system. The first option would require our distinguished proposer to run multiple consecutive Paxos iterations at once. This would require approximately the same amount of processing per iteration but would reduce any down time due to the RTT.

The second option was to implement an optimization discussed in Lamport's paper [3]. In this type of Multi-Paxos system, the distinguished proposer does a single Prepare phase and then uses the same proposal number for a series of Proposal phases. To implement this we would need a global proposal number for every iteration. At this point if the distinguished proposer can guarantee that there are no values for a set of iterations (typically this would be all the future iterations that no one has proposed on), it can perform the Prepare phase once and then simultaneously perform multiple instances of the Propose phase with the same proposal number on these iterations. As long as another node does not attempt the Prepare phase with a higher proposal number (if it is lower, the Prepare attempt will be rejected), the distinguished proposer can continue to use the same proposal number for any number of future proposals. However it another node successfully completes the Prepare phase with a higher proposal number, all proposals that the current distinguished proposer makes will be rejected. The new distinguished proposer will need to perform the Prepare stage on enough Paxos iterations until it finds a consecutive set of iterations equal to the size of the window used for the consecutive proposals before it can perform multiple Proposal phases without the Prepare phase. Therefore it is necessary for the distinguished proposer to use a known window size for the multiple Proposal phase; otherwise no other node could ever safely drop the Prepare phase.

The second option greatly decreases the number of messages required for a successful proposal and provides an easier way to look at the affect of the RTT on the successful proposal. Because of these advantages and because both implementations would require rewriting large portions of the code to ensure correctness in the eyes of the client, we decided the implement the second option. We did not implement a full Paxos system for this; rather we created a prototype based on our original code base. Because it was not necessary for our evaluations and we did not have enough time, we did not implement the required checks to deal with changing distinguished proposers described above. This allowed us to focus on the evaluation of window sizes and latencies instead. Assuming that this distinguished proposer change never happens, our prototype correctly implements the Paxos protocol and guarantees safety.

3 Evaluation

3.1 Correctness Test

For a Paxos system to be correct means that no matter how severe the environment becomes (node crashes, variable network latency) there will never be conflict between nodes about learned(decided) values.

We apply run-time conflict detection and off-line checksums of the run time logs on our implementation of the sequential Paxos system, verifying the correctness of our implementation. The online conflict detection is triggered whenever a learner learns a new value on a certain iteration. On Iteration i, if a learner l learns a value v_i' which is different from the value v_i the learner has already learned (if l has learned a value for Iteration i), inconsistency happens and the learner throws an exception. The offline checksum detector is separately run based on the permanent logs that every learner writes to the disk.

It reads the learned values of all iterations up to a common maximum learned iteration number from their logs. Then sorting and concatenating the values into a file, it computes the checksum of the file. The checksum is required to be identical among nodes.

We ran our Paxos system, randomly killing and starting nodes. It was observed that the nodes dynamically updated their states and filled the holes in their knowledge by requesting state updates from the distinguished proposer or "prepare" to learn the values if they were the distinguished proposer. Our system implementation is correct since we observe that (1) run-time assertions never occur; (2) the off-line checksum remain consistent.

3.2 Sequential Paxos

We evalutated the throughput versus the number of Paxos nodes using our implementation of the sequential Paxos system. Throughput is defined as number of Paxos requests the system can finish during a unit time period (one second). The experiment was run over the lab's high quality LAN.

Given N (from 1 to 10) nodes in the Paxos system, two clients simultaneously send 100 requests to the system, waiting until all responses are received. The total run time is the maximum run time of both clients. The following shows an instance of the data in the format $(\#nodes, totaltime(seconds))$. (1, 13.2), (2, 22.1), (3, 29.8), (4, 38.2), (5, 46.2), (6, 54.3), (7, 62.6), (8, 71.2), (9, 79.3), (10, 87.5). It was observed that the total time needed to finish 200 requests increased linearly with the number of nodes.

We ran each case five times to obtain the mean and standard deviation. Figure 1 shows the throughput verses number of nodes (the green stars annotates the corresponding standard deviation). The throughput decreases roughly according to a $\frac{1}{x}$ curve as the number of nodes increases.

The number of Paxos messages that the distinguished proposer needs to pass and process increases linearly with the existing of number of nodes, which explains that the time-cost increases linearly and the throughput decreases according to $\frac{1}{x}$.

3.2.1 Distinguished Proposer

While we knew that our algorithm for distinguished proposer works in theory, we wanted to ensure that it converges quickly in practice. To help with this evaluation, we implemented our Paxos nodes to wait to attempt becoming distinguished proposer when the old distinguished proposer stops responding until they receive a client session request. To test how long the conflict occurs on average, we calculated the time that it took our Paxos system with 10 nodes and 1 node (other than the distinguished proposer) failed to successfully return 10

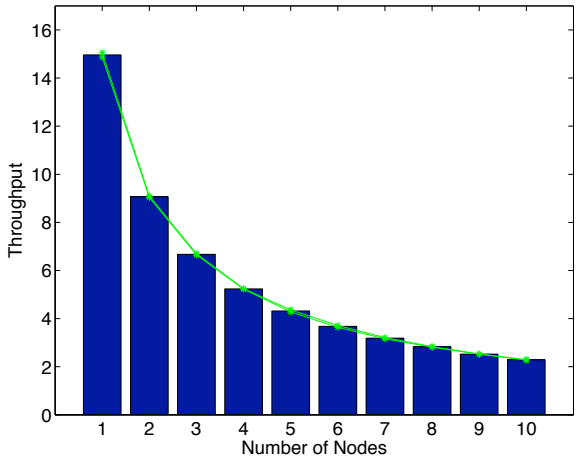

Figure 1: Scalability of our system. The overall time needed to finish a certain amount of requests (200) increase linearly with the increment of number of Paxos nodes, such that our throughput decrease according to a $\frac{1}{x}$ curve.

requests each to 10 clients. Each client was programmed to use a different Paxos node as its access point to the cluster. We then ran a number of tests with all nodes running but the distinguished proposer. After waiting to ensure that all nodes knew that the distinguished proposer was not responding, we simultaneously started the 10 clients. Since every client contacted a different node for a session, each node in the cluster attempts to make a proposal and become the distinguished proposer.

Until a node believes that another node has become the distinguished proposer, it will continue to keep its session with its client and continue proposing (backing off if rejected). Once it learns that there is a distinguished proposer, it will redirect its client to the distinguished proposer. Thus every node must either learn who the distinguished proposer is or become the distinguished proposer itself for the client to successfully complete 10 requests. Thus the difference between this time and the control time is the time it takes for all Paxos nodes to converge to the knowledge of the distinguished proposer. It is okay if a node temporarily believes that an old distinguished proposer is the current distinguished proposer because all nodes periodically send messages to their believed distinguished proposer to see if it is up. If the node is up, it will send the information on who it believes to be distinguished proposer (which could be itself).

In Figure 2, we see a distribution of the additional time required to fulfill the 100 requests. With high probability, the system had a stable distinguished proposer within a short time (less than 2 seconds). On occassion, the convergence time required a full 10 seconds, which we believe to be well within a reasonable time.

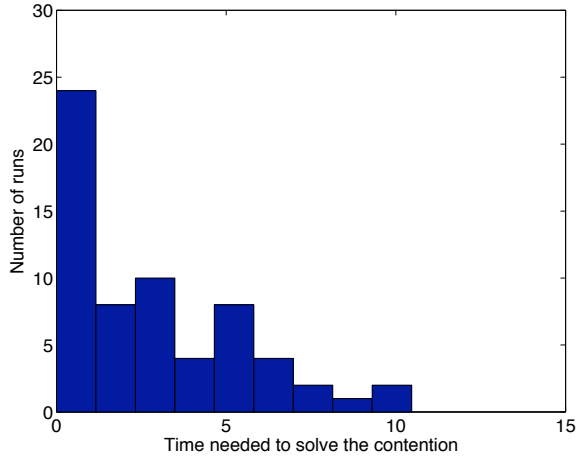

Figure 2: The histogram shows the performance of our leader election strategy. For the 63 runs of contention among 9 Paxos nodes (where the distinguished proposer died), all the contentions were solved within 11 seconds. A large number of them were solved within 2 seconds.

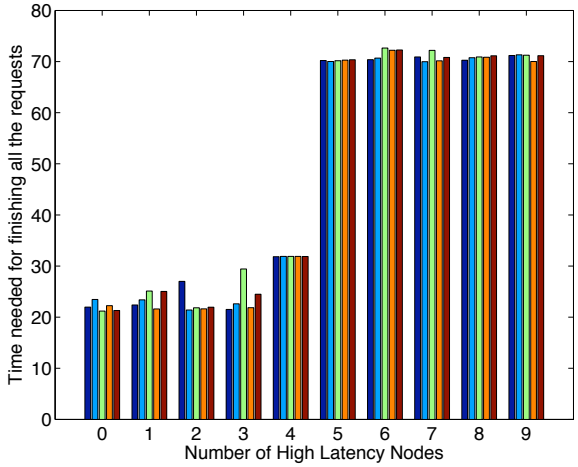

Figure 3: Run time versus the number of high latency nodes. In a 10 node Paxos system, if a majority (greater or equal to 6) nodes are of a low network delay (50ms) to the distinguished proposer, the run time is low, while the run time dramatically increases as soon as a high latency node becomes part of the majority.

3.3 Network Latency Matters

The system throughput is related to network latency of the participating nodes. However, due to the nature of the Paxos system, a simple observation is that if a majority of the nodes had a low latency connection to the leader, a minority of high latency nodes will not alter the throughput. Our theory is that if the bottle neck of the performance of a Paxos system stems from network latency (though we will show in the next section that by appropriately setting the window size in a Parallel Paxos system, the network latency's effect on the performance will be lessened), the throughput is determined by the latency of the node whose latency is after the node with median latency (when sorting latency in ascending order). In more formal terms, assume N nodes. Let l_i be the network latency of Node i, and $\{m_i\}$ be sorted network latency in ascending order of $\{\forall i : l_i\}$, the throughput T would be,

$$T = f(m_{floor(N/2)+1}) \quad (1)$$

We observed this pattern by simulating additional network latency in the LAN. For the test, 10 Paxos nodes were set up. Excluding the distinguished proposer, we divided the remaining 9 nodes into two groups, high latency and low latency. The nodes in the low latency group possessed an additional 50ms network delay, while the nodes in the high latency group possessed an additional 500ms network delay. The client sent 100 requests to the distinguished proposer. Figure 3 shows the time required to complete the 100 requests versus the number of high latency nodes. The graph supports our theory.

It may be noted that time required slightly increased at 4 high latency nodes. We observed that this occurs because in our implementation the distinguished proposer also sends every message to itself. Since it is also processing messages from the client, its reponse time is slower than the other 5 low latency nodes, since they are usually idle. Therefore the distinguished proposer's response time is the deciding factor because it is the node above the median latency node, which also supports our theory in the general sense (not just network latency makes this effect).

3.4 Multi-Paxos: Parallel

We originally implemented our parallel Multi-Paxos with the thought that we would see large gains with larger windows. However we were instantly thwarted and found that for all window sizes we achieved similar results. No amount of tweaking our prototype delivered better results. After digging through the logs, we were surprised to find that the larger window sizes were not achieving better results because, at normal network latency, the distinguished proposer was receiving messages as fast as it could process and respond to them even with a window size of 1. Thus we decided it would be important to see how much latency was required for the different window sizes. After introducing latency into our network, we observed the following results.

Figure 4 shows the relationships of various window sizes and the overall throughput given different network latency. At normal network latency (0 additional simulated latency), all window sizes perform similarly. As the latency increase, the window size becomes more impor-

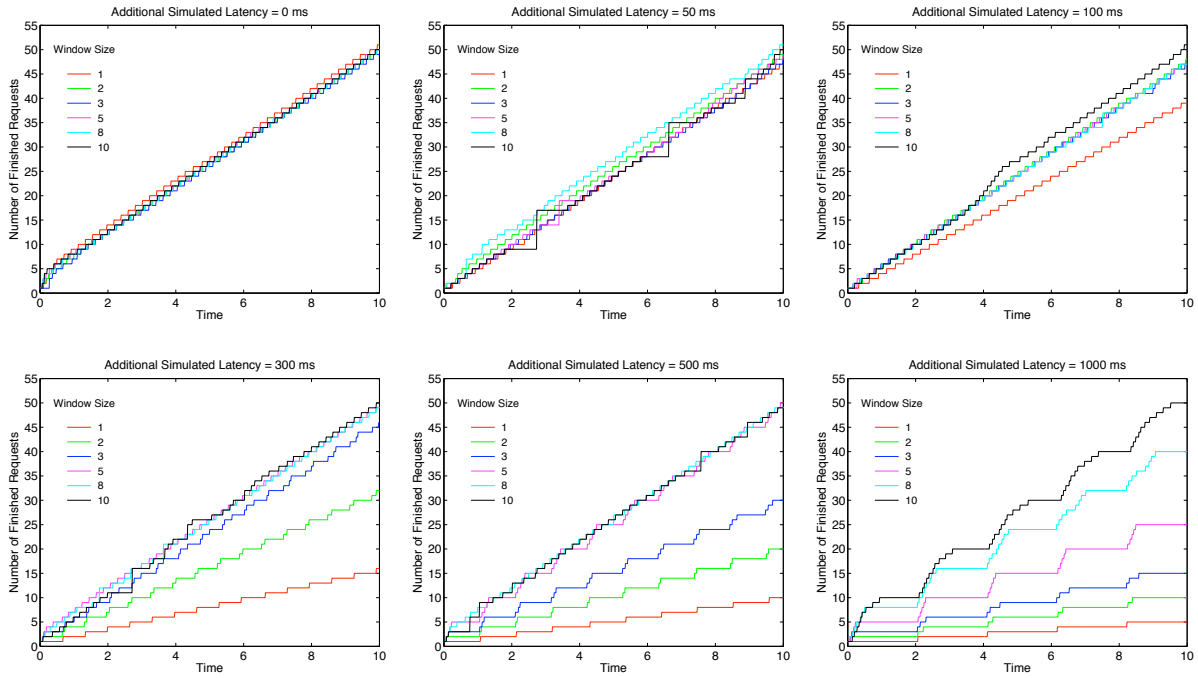

Figure 5: Detailed profiles illustrating the relationship among window sizes and network latency. In a network with higher latency, parallel Paxos, by increasing its window size, is able to retain the throughput performance obtained in a low latency network. Once the idle time period caused by the network latency is saturated, increasing the window size further will not result in any more performance gains.

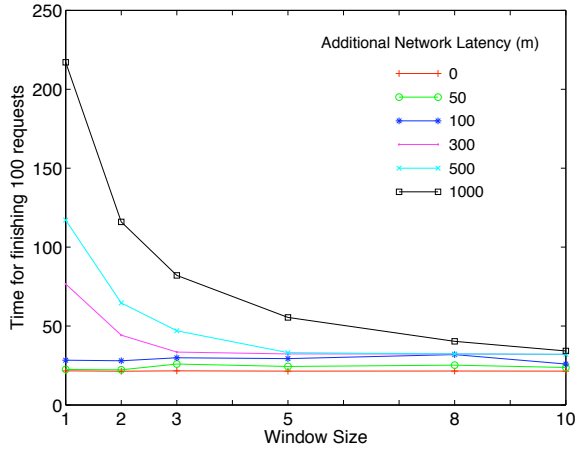

Figure 4: The effects of various window sizes to the throughput given different additional simulated network latencies. When the network latency is small compared to the time needed for processing a Paxos messages, window size does not really matter and parallel Paxos performs similar as sequential Paxos. When the network latency is large, sequential Paxos (or parallel Paxos with small window sizes) will be idle waiting for responses, while parallel Paxos with larger window sizes will continue to make progress.

tant. Of interesting note is the fact that the window size of 10 does not become more effective until there is an entire additional second of network latency beyond normal latency.

Figure 5 displays the number of successful requests over time under different variations of window sizes and additional simluated latencies. It can be observed that, when the network latency is large and the window size is small, there are significant amounts of time in which the distinguished proposer (as well as other nodes) are idle. Increasing the window size helps to fill the idle time and improves the overall throughput. On the other hand, given a defined average network latency, increasing the window size to extremely large does not make sense, because each node (especially the distinguished proposer) needs time to process the Paxos messages. Larger window size than the system can tolerate will lead to fulfilling the message buffer queue very quickly and could result in lost messages. Additionally, the large the window size, the longer the search the distinguished proposer must make before it can drop the Prepare phase and only implement the Propose phase.

4 Conclusion and Future Work

We implemented a sequential Paxos system and demonstrated that its througput decreases according to a $\frac{1}{x}$ func-

tion as the number of nodes increases (scalability). We verified that, if the bottle neck is due to network latency, the overall performance is limited by the slowest node in the fastest majority. We proposed and implemented a method for the electing distinguished proposer and demonstrated that with high probablity it will converge in a short time. Further, we partially implemented a parallel paxos system and tested its throughput versus various window sizes and network latency, showing that a larger window size does help to maintain throughput on higher latency networks.

In the version we implemented for the evaluation purposes for this paper, all the paxos requests are reads and writes to small sized varibles in a replicated storage system. Since paxos requests are costly, which normally requires $O(n)$ (n is the number of nodes) messages being sent and receive, it makes sense to combine unrelated requests from different clients and make them into a one big request. In that case, the packet size would become more important, and it would be interesting to take into consideration not only network latency but also network bandwidth. Also we would like to see all these evaluations in a WAN setting, where there is more variation in routing and packet loss.

References

[1] T. D. Chandra, R. Griesemer, and J. Redstone. Paxos made live: an engineering perspective. In *Proceedings of the 26th annual ACM symposium on Principles of distributed computing*, pages 398–407, 2007.

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

[3] L. Lamport. The part-time parliament. In *ACM Transactions on Computer Systems (TOCS)*, volume 16, pages 133–169. ACM, 1998.