Paxos for System Builders: An Overview

Yair Amir and Jonathan Kirsch
Johns Hopkins University
{yairamir, jak}@cs.jhu.edu

1 Introduction

State machine replication (SMR) [8, 14] is a well-known technique for building distributed services requiring high performance and high availability. The Paxos protocol [9, 10], developed by Leslie Lamport, is perhaps the most widely-known SMR protocol and has received a great deal of attention in the literature. Although Paxos was known in the 1980s to some and published in 1998, it is difficult to understand how the protocol works from the original specification. Further, the original specification had a theoretical flavor and omitted many important practical details, including how failures are detected and what type of leader election algorithm is used.

As system builders, we believe that filling in these missing details is critical if one wishes to build a real system that uses Paxos as a replication engine. This white paper describes our experience over the last several years in trying to answer several fundamental questions about Paxos: How robust is it to processor and network failures? What liveness properties does it guarantee? What level of performance can one expect from it? Our research has aimed to clearly and completely specify Paxos such that system builders can understand it and implement it.

We are not the first to attempt to clarify Paxos or specify it more precisely [2, 4, 11–13]. These previous works are valuable because they give insight into the correctness (i.e., safety) of Paxos. However, the works (with the exception of [4]) are mostly theoretical in nature, and none specifies all of the details necessary to translate the algorithmic specification into an implementation (e.g., flow control, message recovery, failure detection, etc.). One of the main contributions of our work is a complete specification of protocol pseudocode for our interpretation of Paxos (which we call Paxos for System Builders). Due to space limitations, this white paper only briefly presents Paxos for System Builders (see Section 2); we refer the interested reader to the complete specification [7].

We comment that the systems-related details missing from existing Paxos specifications are explicitly addressed by existing SMR protocols that operate above a group communication system (GCS) (e.g., [1, 6]). We believe the only way to understand the tradeoffs of using Paxos compared to these GCS-based systems is to completely specify Paxos such that (1) its safety and liveness properties can be stated and (2) it can be implemented and evaluated.

Our experience has shown that there is much more to developing a Paxos-based replication engine than simply specifying a "correct" protocol. While our work takes a system builder perspective, we also bring to light important theoretical differences, related to liveness, that arise from how one specifies the details of Paxos; specifically, our focus is on the choice of leader election algorithm, as discussed in Section 3. Our analysis reveals that claims about the robustness of Paxos only make sense in the context of a complete specification of the Paxos algorithm and all its components. We present experimental results of our own implementation of Paxos for System Builders in Section 4. This white paper is an overview; a more complete description is available in a technical report [7].

2 Protocol Overview

As a state machine replication protocol, Paxos assigns a global, persistent, total order to client updates. A server executes an update after it has executed all previous updates in the global order. During normal-case operation, one server acts as the *leader*. The leader assigns each update a sequence number, and then sends a PROPOSAL message, proposing the assignment, to the other servers. The non-leader servers respond to the proposal by sending an ACCEPT message to all servers, which acknowledges that they have accepted the proposed ordering. A server *globally orders* an update when it collects the PROPOSAL in which the update is contained and $\lfloor N/2 \rfloor$ matching ACCEPT messages, where N is the total number of servers in the system. Normal-case operation is summarized in Figure 1.

In the protocol described above, the leader's PROPOSAL messsage is used as an implicit ACCEPT message for the purpose of ordering. In this case, Paxos requires two sequential disk writes: the leader must sync to disk before sending the PROPOSAL, and each non-leader must sync be-

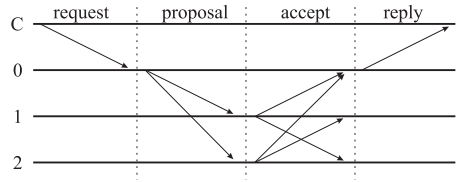

Figure 1: Paxos normal-case operation. Client C sends an update to the leader (Server 0). The leader sends a PROPOSAL containing the update to the other servers, which respond with an ACCEPT message. The client receives a reply after the update has been executed.

fore sending its ACCEPT. One possible variation is that the leader can send an ACCEPT message as well, in which case it could sync to disk in parallel with the other servers. If disk writes are expensive with respect to the overall latency of the ordering, then this results in a reduction of the ordering latency at the cost of one additional incoming message per non-leader (and $N-1$ additional sends by the leader). In either case, the initiator of an update syncs it to disk before sending it to the leader to ensure that the update is consistently identified across crashes.

Servers attempt to elect a new leader if insufficient progress is being made – that is, if a timer expires without the server having executed a new update. Paxos for System Builders uses a leader election protocol similar to the one used by the BFT protocol [3], adapted for use in benign environments. When a server's timer expires, it sends a VIEW-CHANGE message to the other servers. A server completes the leader election protocol when it collects VIEW-CHANGE messages from a majority of servers for the view it is attempting to install.

3 Liveness: Theory and Practice

The liveness of a protocol reflects its ability to make forward progress. Given sufficient network stability, Paxos allows any majority of servers to order new updates, regardless of past failures. What degree of network stability is "sufficient"? Answering this question is not straightforward because it is difficult to define what exactly "the Paxos protocol" is. Many of the important details, some of which play a large role in determining the liveness of the overall protocol, were not originally specified. In this section we discuss the liveness of both the leader election protocol and the normal-case operation of Paxos.

Leader Election: Our work on Paxos for System Builders revealed that Paxos is only as live as its leader election protocol. Although many different leader election protocols can be used without impacting the safety of the system, the choice of leader election has a significant impact on overall system liveness. When we compared the network stability requirements of several different Paxos specifications, we observed that there is a scale of network stability requirements for the overall systems, resulting directly from the choice of leader election protocol. Each leader election protocol requires a different level of stability to remain on a single leader (which is needed to guarantee liveness).

The network stability requirement of the leader election protocol used in Paxos for System Builders can be stated formally as:

DEFINITION 3.1 STABLE MAJORITY SET: *There exists a set of processes, S, with $|S| > \lfloor N/2 \rfloor$, that are eventually alive and connected to each other, and which can eventually communicate with each other with some (unknown) bounded message delay.*

The leader election failure detector specified in [13] requires a stronger degree of stability from the network to remain on a single leader. In the protocol, each server maintains a list of the servers it believes to be alive, and the leader is chosen as the server with the highest identifier. A Paxos system using this failure detector requires a stable majority set whose members do not receive messages from particular unstable servers that repeatedly crash and recover or partition (i.e., those with higher identifiers).

The leader election failure detector specified in [2] requires strictly less stability than that of [13], requiring the stable majority set to be isolated only from particular unstable servers that repeatedly partition (servers that repeatedly crash and recover are eventually not considered as potential leaders). The protocol used in Paxos for System Builders is not vulnerable to disruption by any unstable servers. However, in certain cases, Paxos for System Builders requires a slightly longer time to settle on a leader in the stable majority set than [2], since our elections are not based on which servers are believed to be alive.

We emphasize that our comparison of leader election protocols is not exhaustive, and the purpose of the preceding discussion is not to endorse a particular protocol. Rather, our goal is to highlight the importance of specifying Paxos in its entirety so that its performance and availability can be evaluated and compared to other solutions.

Normal-case Operation: In principle, Paxos can continue to order new updates as long as the leader can communicate with a majority of servers, where the members of the majority can switch very rapidly (although note that the leader election protocol might not tolerate such rapid fluctuation). However, there is a difference between the theoretical network stability requirement for *ordering* and the practical network stability requirement for *execution*. In order to use Paxos as a replication engine, servers must be able to execute new updates (i.e., to globally order new updates with no holes). There seems to be little value in having servers order updates very quickly if execution must be delayed due to gaps in the global sequencing. To execute at full speed, a server must either be continuously connected to a majority

of servers (and the leader) so that it can order each update in sequence, or reconciliation must occur sufficiently quickly so that it is as if the server were continuously connected.

To the best of our knowledge, previous Paxos specifications did not make this important distinction. Our implementation uses a window mechanism to ensure that a majority of servers can execute new updates. The leader will only send a PROPOSAL with sequence number i if it has executed all updates through sequence number $i - W$ (where W is the size of the window), and a non-leader server will only respond to a PROPOSAL with sequence number i if it has executed all updates through $i-W$. Thus, there exists a majority of servers that have executed all updates within two windows of the latest proposal. We note that GCS-based state machine replication protocols such as COReL [6] and Congruity [1] do not face this issue because they deliver updates only when they are ready to be executed.

System Liveness: We now present the liveness properties provided by Paxos for System Builders:

Paxos-L1 (Progress): If there exists a stable majority set of servers, then if a server in the set initiates an update, some member of the set eventually executes the update.

Paxos-L2 (Eventual Replication): If server *s* executes an update and there exists a set of servers containing *s* and *r*, and a time after which the set does not experience any communication or process failures, then *r* eventually executes the update.

To gain a deeper understanding of the implications of Paxos-L1, we first compare it to the level of network stability required by many GCS-based replication systems (e.g., COReL [6] and Congruity [1]). In order for the membership algorithm of the GCS to terminate, the system requires a stable component of servers, which, following [5], we define as follows:

DEFINITION 3.2 STABLE COMPONENT: *There exists a set of processes that are eventually alive and connected to each other and for which all the channels to them from all other processes (that are not in the stable component) are down.*

Notice that the stable majority set (required by Paxos for System Builders) allows servers in the set to receive messages from servers outside of the set, whereas the stable component requires an isolated set of servers. Thus, our specification of Paxos requires less network stability than GCS-based replication protocols. Paxos allows servers outside the majority to come and go without impacting overall system liveness. We believe GCS-based protocols can most likely be made to achieve Paxos-L1 by passing information from the application level to the group comunication level, indicating when new membership should be permitted (i.e., after some progress has been made).

The size of the stable component required for the GCS-based replication protocol to make progress differs depending on whether or not dynamic voting is employed in establishing a quorum. If static quorums are used (as in COReL), then, like Paxos, the stable component must contain a majority of the total number of servers in the system. If dynamic linear voting is employed (as in Congruity), then progress may be able to continue with less than a majority of the system.

Note that leader election can be implemented on top of a GCS membership algorithm; the resulting system will require the same level of network stability as the underlying group membership algorithm to guarantee liveness. This point – the difference in stability required for membership compared to the stability required for leader election – is the key liveness difference between Paxos and GCS-based replication systems.

Finally, we compare Paxos-L1 to the following alternative progress requirement:

DEFINITION 3.3 STRONG L1 (MAJORITY SET): *If there exists a time after which there is always a set of running servers S, where $|S|$ is at least $(\lfloor N/2 \rfloor + 1)$, then if a server in the set initiates an update, some member of the set eventually executes the update.*

Strong L1 requires that progress be made even in the face of a (rapidly) shifting majority. We believe that no Paxos-like algorithm will be able to meet this requirement. If the majority shifts too quickly, then it may never be stable long enough to complete the leader election protocol.

4 Experimental Results

We implemented Paxos for System Builders, and source code is available for download at http://www.dsn.jhu.edu. In this section we evaluate the performance of our implementation. As mentioned in Section 3, it is difficult to define exactly what "Paxos" is. This problem is even more prominent when one tries to evaluate its performance. Should an evaluation of Paxos allow the use of multicast (e.g., IP multicast or smart overlay multicast), or should links be point-to-point? Should an implementation use aggregation that can increase throughput? Should it include group-based flow control? Is such an algorithm still "Paxos?" What type of reliability mechanisms should be used? How are servers that were crashed or partitioned away brought up to date?

Our implementation of Paxos for System Builders represents what we believe "Paxos" should be. It includes systems-related details we believe are necessary for achieving practical state machine replication. Our implementation

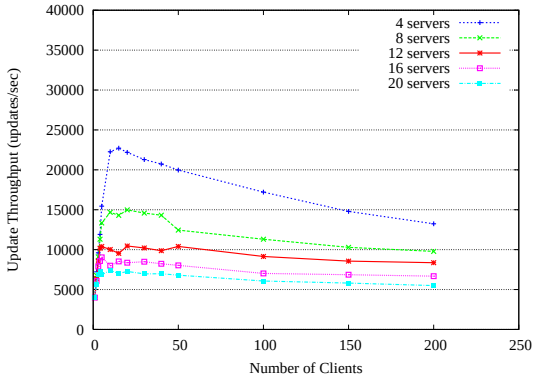

Figure 2: Update Throughput, No Disk Writes

Figure 3: Update Throughput, Synchronous Disk Writes

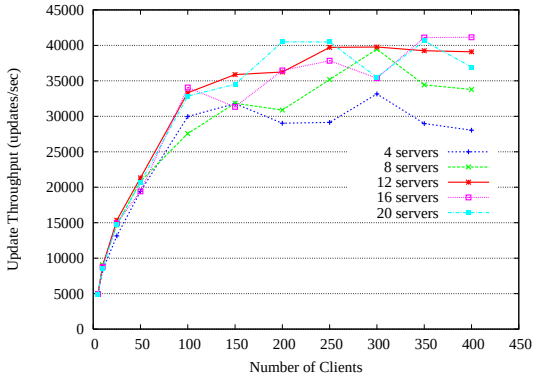

Figure 4: Update Throughput, Aggregation, No Disk Writes

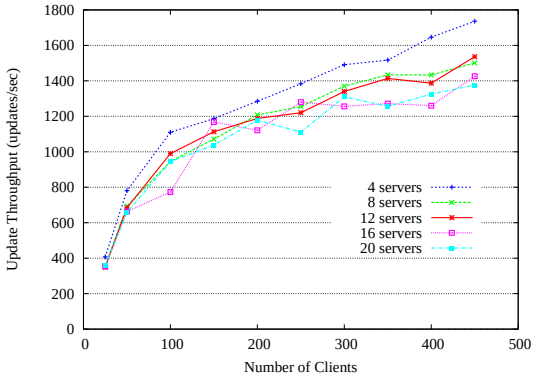

Figure 5: Update Throughput, Aggregation, Synchronous Disk Writes

employs group-based flow control; the leader maintains a loose membership of which servers it believes to be alive, and it slows down to ensure that all of these servers can execute updates. We present results both with and without aggregation so that its (significant) impact can be evaluated. Due to space limitations, we omit results that show the impact of other systems-related issues, such as the use of IP multicast (used in the following experiments). The interested reader is referred to [7] for a more complete performance evaluation.

Network Setup: We ran our tests on a cluster of twenty 3.2 GHz 64-bit, dual processor Intel Xeon computers, connected via a Gigabit switch. We tested our implementation on configurations ranging from 4 to 20 servers, varying the number of clients initiating write updates. Each update is 200 bytes long, representative of an SQL statement. Clients are spread as evenly across the servers as possible. Each client has at most one outstanding update at a time.

Memory Tests (No Disk Writes): Although Paxos is not resilient to crashes when disk writes are not used, evaluating the system when all operations are performed in memory shows the type of performance that can be achieved strictly when considering the messaging and processing overhead associated with normal-case operation.

Figure 2 shows the throughput achieved in configurations ranging from 4 to 20 servers. In all configurations, we observe a steady increase in throughput until the system reaches its saturation point (i.e., when the leader becomes CPU-limited), at which point throughput levels off. We achieve a maximum throughput of 22,716 updates per second when using 4 servers, with a plateau around 7000 updates per second when using 20 servers.

Synchronous Disk Writes: Paxos requires all servers to write to disk on each update; in addition, the initiator of an update must write to disk before sending its update to the leader so that the update is uniquely identified across crashes. Figure 3 shows the throughput when using synchronous disk writes. The implementation achieves between 25 and 35 updates per second in all configurations tested. The throughput is limited by the speed at which a single server can sync to disk. Hence, running Paxos "natively" is very expensive when crash resiliency is required.

Aggregation: Without disk writes, we aggregate in several ways. First, the leader packs multiple updates into a single PROPOSAL message. Second, non-leader servers sends a single ACCEPT message for several PROPOSALS, greatly reducing the number of messages that must be processed (although each ACCEPT is now larger, since it con-

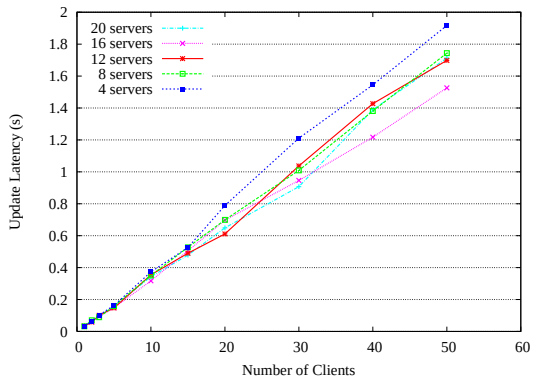

Figure 6: Update Latency, No Aggregation, Synchronous Disk Writes

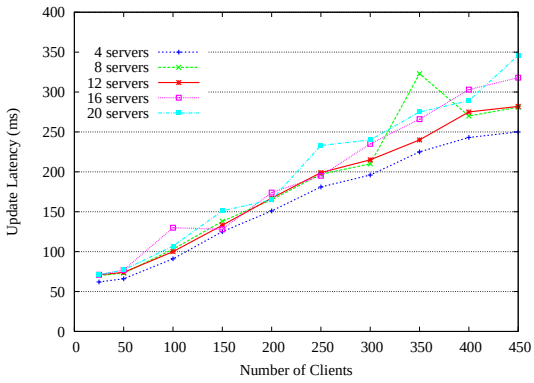

Figure 7: Update Latency, Aggregation, Synchronous Disk Writes

tains a list of the sequence numbers that it covers).

As seen in Figure 4, aggregation significantly changes the trend of the throughput graph. Instead of throughput degrading as the number of servers increases, it increases, reaching a maximum of over 40,000 updates per second. Reducing the number of ACCEPT messages being sent greatly reduces the overhead associated with adding more servers. It also allows more of the CPU to be devoted to processing new UPDATE and PROPOSAL messages, rather than extra acknowledgements.

When using synchronous disk writes, we sync several PROPOSAL messages to disk at once, while sending back a single ACCEPT message. We also sync several updates from local clients at once, amortizing the cost of syncing upon initiation over several updates. The results are shown in Figure 5. Comparing Figures 3 and 5, we can see that aggregation dramatically increases the maximum throughput for all configurations of servers (from roughly 35 updates per second to about 1500 updates per second).

Figures 6 and 7 show the update latency when using synchronous disk writes, with and without aggregation. When no aggregation is used, Paxos pays a high cost to provide crash resiliency, reaching a latency of one second at around 30 clients. However, aggregation dramatically reduces the latency, amortizing the cost of syncs across many updates.

In summary: Our work shows that the small technical details, which are usually looked at as engineering considerations, actually have large liveness implications and can dramatically impact performance. These details are critical components for building a high-performance, highly-available Paxos-based replication engine.

References

[1] Y. Amir. *Replication Using Group Communication Over a Partitioned Network.* PhD thesis, The Hebrew University of Jerusalem, Jerusalem, Israel, 1995.

[2] R. Boichat, P. Dutta, S. Frlund, and R. Guerraoui. Deconstructing Paxos. *SIGACT News*, 34(1):47–67, 2003.

[3] M. Castro and B. Liskov. Practical byzantine fault tolerance. In *Proceedings of the 3rd Symposium on Operating Systems Design and Implementation (OSDI '99)*, pages 173–186. USENIX Association, Co-sponsored by IEEE TCOS and ACM SIGOPS, 1999.

[4] 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 (PODC '07)*, pages 398–407, New York, NY, USA, 2007. ACM Press.

[5] G. Chockler, I. Keidar, and R. Vitenberg. Group communication specifications: a comprehensive study. *ACM Computing Surveys*, 33(4):427–469, 2001.

[6] I. Keidar. A highly available paradigm for consistent object replication. Master's thesis, Institute of Computer Science, The Hebrew University of Jerusalem, Jerusalem, Israel, 1994.

[7] J. Kirsch and Y. Amir. Paxos for system builders. Technical Report CNDS-2008-2, Johns Hopkins University, www.dsn.jhu.edu, 2008.

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

[9] L. Lamport. The part-time parliament. *ACM Transactions on Computer Systems*, 16(2):133–169, May 1998.

[10] L. Lamport. Paxos made simple. *SIGACTN: SIGACT News (ACM Special Interest Group on Automata and Computability Theory)*, 32:18–25, 2001.

[11] B. Lampson. The ABCD's of Paxos. In *Proceedings of the 20th ACM Symposium on Principles of Distributed Computing (PODC '01)*, page 13. ACM Press, 2001.

[12] H. C. Li, A. Clement, A. S. Aiyer, and L. Alvisi. The Paxos register. *Proceedings of the 26th IEEE International Symposium on Reliable Distributed Systems (SRDS '07)*, 2007.

[13] R. D. Prisco, B. Lampson, and N. Lynch. Revisiting the Paxos algorithm. *Theor. Comput. Sci.*, 243(1-2):35–91, 2000.

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