Layered Paxos: A Hierarchical Approach to Consensus

Andrew Nagawiecki
Department of Computer Science
Rensselaer Polytechnic Institute
Troy, United States
andrew.nagawiecki@gmail.com

Stacy Patterson
Department of Computer Science
Rensselaer Polytechnic Institute
Troy, United States
sep@cs.rpi.edu

***Abstract*—We present a new consensus algorithm for use in distributed systems. The algorithm, Layered Paxos, is designed for hierarchical systems where processes can be grouped into disjoint components. The underlying communication network is assumed to be two-fold with quick communication between processes in the same component and high-latency communication between processes in different components. Layered Paxos uses the Paxos algorithm as a building block with the addition of a few mechanisms to achieve hierarchical consensus and aims to increase the overall consensus throughput of a system that follows this two-fold communication paradigm. We present a theoretical proof that Layered Paxos satisfies the basic safety and liveness properties of Paxos in addition to some related to the hierarchical nature of the system. Finally, we present the experimental results of the comparison between Layered Paxos and Paxos in a hierarchical system, in which Layered Paxos was able to achieve up to 5x the throughput of Paxos.**

I. Introduction

With the rise in data creation and cloud computing, it is no surprise that reliable replication of large sets of data has become a growing problem for which to provide more robust solutions. For years, distributed systems have made use of consensus algorithms to achieve this replication where data needs to be constantly available despite replica crashes. As modern-day systems grow larger and scale globally, the machines running these consensus algorithms are often split into clusters spread out over various geographical locations. This is seen when looking at systems consisting of data centers of many machines with multiple data centers across the world. The underlying communication network of these systems is usually two-fold. The first is a fast, local communication among servers in the same cluster. The second is a high-latency communication between servers in different clusters due to the spatial distance between them.

One of the main consensus algorithms used in these systems to achieve replication is Paxos [1][2]. However, Paxos requires multiple rounds of messaging between a majority of servers in the system in order to reach consensus on a single value. In the globally scaled systems described above, these messaging rounds become costly due to the high latency communication between data centers. As globally scaled systems provide service to a large number of clients, it is not desirable for a client's request to be processed slowly.

In this paper, we propose Layered Paxos, which is tailored towards these types of distributed systems. Within each data center or cluster of servers, a local instance of Paxos is run to queue up client requests. One server from each cluster elected as a delegate then participates in a global instance of Paxos. These delegates propose and reach consensus, using Paxos, on batched sequences of client requests which are then replicated in the global system. This hierarchical pattern makes use of the fast network between servers in the same cluster to quickly queue up a large sequence of client requests while reducing the high latency effect of the global network to increase the overall throughput of the system.

Layered Paxos guarantees a totally ordered global log across all processes. The ordering of values queued within each component is maintained in the global log; however, there is no ordering imposed on values queued in different clusters. Layered Paxos will make progress given that a quorum of clusters are active and able to queue up values within their cluster. In addition, a quorum of delegates must be stable long enough with enough messages eventually delivered between them.

Finally, we present an experimental evaluation of Layered Paxos. The throughput of Paxos and Layered Paxos are compared in a hierarchical system with a varying number of components. We also define two types of request latency depending on the type of application the target system is running. These two types of request latency are compared between Paxos and Layered Paxos to discuss what must be taken into consideration when using Layered Paxos in any system. We end by analyzing the effects of delegate crashes on the execution of Layered Paxos.

Some attempts have already been made to improve Paxos in these network situations. In D-Paxos [3], processes are grouped into components. Processes within each component run Paxos to queue up a batch of client requests. One process from each component is elected to be the delegate of the component who proposes a batch of client requests to the other delegates. Our proposed solution takes inspiration from D-Paxos; however, it requires less from the system model than D-Paxos requires such as Partial Synchrony [4] and a failure detector of eventually perfect class [5].

In C-Raft [6], a similar structure to D-Paxos is used;

however, the underlying algorithm used for replication is Raft [7]. Raft is a replication algorithm similar to Paxos and was designed for the purpose of understandability. As such, there are a few extra overhead mechanisms required by Raft that are not required by Paxos. Our proposed solution aims to follow the same structure of C-Raft but use Paxos as the underlying replication algorithm. This reduces the number of overhead mechanisms required and simplifies our proposed solution.

In Institutionalized Paxos [8], processes are grouped into clusters similar to D-Paxos. However, the main contribution in this work is the protocol for membership changes between these clusters. Each cluster has its own state machine that is replicated through Paxos, but the same state is not replicated across the whole system of clusters. Our proposed solution allows for multiple groupings of processes to replicate the same state across the whole system, regardless of which grouping a process belongs to.

The rest of this paper consists of the following. Section II states the system model as well as the safety and liveness properties of Layered Paxos. In Section III, we describe more properly the specifications of the Layered Paxos algorithm. In Section IV, we analyze the algorithm specification in order to prove that it satisfies the safety and liveness properties. Section V discusses experimental results of Layered Paxos compared to Paxos. Finally, we conclude the paper in Section VI.

II. System Model

A. Definitions

Our system consists of n processes that are grouped into m disjoint components. We define a *component* as a set of processes that does not intersect with any other defined component. Within each component, a single process is elected by a quorum within the component to serve as a *delegate*, which serves a special role in consensus. In Section III, we define the election procedure that components must use to elect their respective delegate. Fig. 1 depicts three components each with three processes with the orange representing the process chosen to serve as a delegate. Messaging within the system is asynchronous, and messages may be lost or duplicated but never corrupted. We assume that each process has a unique ID, and a process may send a message to any other process if it has knowledge of the said process ID. Processes may crash and recover with information they have stored in stable storage. A process is said to be *active* if it is not currently crashed. A component is *active* if and only if a quorum of its processes are currently active and there exists an active delegate process.

Since our proposed solution is a layered algorithm, we find it useful to define two distinct layers used in the algorithm. The local layer involves running Paxos between all processes within a single component. Thus, there is a single local layer of Paxos run for each component in the system. The goal of the local layer is to quickly queue up client requests received by any process in said component. Below are definitions used when discussing the local layer of consensus.

- **Local Paxos Instance:** The behavior of Paxos that is run between processes within the same component in order to decide upon a Local Log of values.
- **Local Log:** The log replicated among all processes within the same component. Processes from different components may have differing Local Logs.
- **Local Acceptor:** A process acting as an acceptor in the Local Paxos Instance.
- **Local Proposal:** A proposal that is made in the Local Paxos Instance.

Above the local consensus layers lies a global layer, which involves running a slightly modified version of Paxos between the elected delegates from each component. In this layer, each delegate proposes batched client requests of size k that have been queued up in its respective component's local layer. Once a client request has been chosen at both the local and global layer, it is considered to be chosen by the whole system. Below are definitions used when discussing the global layer of consensus.

- **Delegate:** A single process from a component recognized by a quorum of processes in the component to serve in the Global Paxos Instance.
- **Global Paxos Instance:** The behavior of Paxos that is run between the each component's delegate in order to decide upon a Global Log of values.
- **Global Log:** The log replicated among all processes in the system (regardless of component membership). A "slot" in the Global Log stores a contiguous sequence of values from any component's Local Log proposed at the Global Paxos Instance. The Global Log maintains the true state of the system.
- **Global Acceptor:** A process acting as an acceptor in the Global Paxos Instance.
- **Global Proposal:** A proposal that is made in the Global Paxos Instance.

In Fig 2. we show a high-level depiction of the general flow of Layered Paxos.

B. Safety Properties

The underlying goal in our consensus problem is to replicate a totally-ordered Global Log at each site. In order to achieve this Global Log, we make use of a Local Log within each component. The Local Log is comprised of Local Proposals made from each site. To ensure a totally-ordered Global Log,

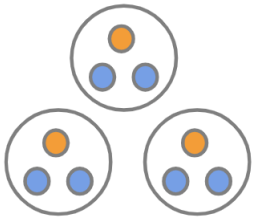


Fig. 1: Example of three components each with three processes.

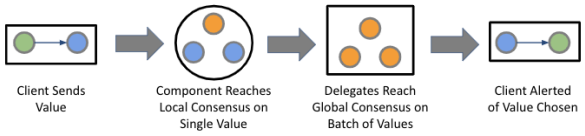


Fig. 2: High level flow of Layered Paxos.

our proposed solution satisfies the following safety properties, which will be proven in Section IV.

- **LS1:** Only a value proposed in the Local Paxos Instance may be chosen within the same component it was proposed.
- **LS2:** Only a single value is chosen for any slot in the Local Log.
- **LS3:** A process never commits any value to a Local Log slot unless it has actually been chosen for that slot in the Local Log.
- **GS1:** Only a value proposed in the Global Paxos Instance may be chosen within the Global Paxos Instance.
- **GS2:** Only a single sequence of values is chosen for any slot in the Global Log.
- **GS3:** A process never commits any sequence of values to a Global Log slot unless it has actually been chosen for that slot in the Global Log.
- **GS4:** Only a contiguous sequence of values chosen in a Local Log in any Local Paxos Instance may be chosen in the Global Paxos Instance.
- **GS5:** A value chosen in a component's Local Log may not be chosen more than once in the Global Log.

C. Liveness Properties

From the FLP theorem [9], we know that it is impossible for processes to reliably reach consensus in an asynchronous system with potential crash failures without requiring additional liveness conditions. We find it useful to define *Local Progress* as components being able to reach consensus on values in their respective Local Logs and *Global Progress* as delegates being able to reach consensus on values in the Global Log. In order to satisfy the safety properties defined above, Layered Paxos requires the following liveness conditions in order to successfully make progress.

- **LL1:** Local Progress will be made in a component given that a quorum of its processes are active and enough messages within the component are eventually delivered.
- **GL1:** Global Progress will be made given that a quorum of components are active and enough messages between delegates are eventually delivered.
- **GL2:** Global Progress will be made given that a quorum of delegates are stable for long enough.
- **GL3:** Global Progress will be made given that a quorum of components are able to make Local Progress for long enough.

III. Algorithm Description

A. General Algorithm Procedure

Initially, each delegate is pre-decided, and that information is shared with all processes in the system. Local Paxos Instances begin execution allowing processes in each component to fill up a Local Log of values proposed and accepted by processes in the same component. The behavior of the Local Paxos Instance consists of processes in the same component executing a single instance of the Synod algorithm per Local Log slot. At any point, a delegate may make a Global Proposal containing a contiguous sequence of values from its component's Local Log starting one slot above the last Local Log slot from the delegate's component committed in the Global Log. The Global Paxos Instance behavior consists of the delegates executing a modified instance of the Synod algorithm for each Global Log slot. The modifications of which are further described below. Throughout execution of the Global Paxos Instance, the following rules must be followed:

- A delegate must propose its desired sequence of values for the first empty slot in the Global Log.
- If a process receives any Global Paxos message but is not the component's current delegate, it should respond to the sender with its knowledge of the current delegate of its component.
- If the proposer delegate receives a message back that it contacted the incorrect process in another component, it should update its knowledge of the delegate to the response and try sending the global message again to the updated delegate process.

Phase 1 of the Global Paxos Instance behaves the same as that of Synod with the following modifications, which can be seen in Fig. 3:

- Upon receiving a GlobalPrepare message from another delegate, the delegate should "forward" this message to all processes in its component, given that the delegate has not already responded to a higher numbered proposal number.
- Any non-delegate process, upon receiving a forward GlobalPrepare message from its component's delegate, should record the proposal number into stable storage and send a positive response back to the delegate who sent the message.
- Once the delegate has received a positive response from a quorum of processes in its component, the delegate may respond to the global proposer with a GlobalPromise message.
- If the proposer delegate does not receive a response from a majority of other delegates, it should retry its proposal.

Phase 2 of the Global Paxos Instance behaves the same as that of Synod with the following modifications, which can be seen in Fig. 3:

- Upon receiving a GlobalAccept message from another delegate, the delegate should "forward" this message to all processes in its component.

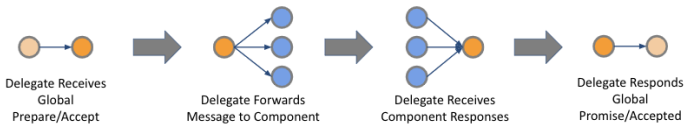


Fig. 3: The general flow of delegate forwarded messages.

- Any non-delegate process, upon receiving a forwarded GlobalAccept from its component's delegate, should record the message contents in stable storage and send a positive response back to its delegate.
- Once the delegate has received a positive response from a majority of processes in its component, the delegate may respond to the global proposer with a GlobalAccepted message.
- Once the proposer delegate has received a GlobalAccepted message from a majority of all delegates, the value is chosen and it sends a GlobalLearn to all delegates.

When a process receives a GlobalLearn message, it sends a ForwardLearn message to every process in its respective component. When a process receives a ForwardLearn message, it commits the accepted sequence to its Global Log. At this point, the sequence of Local Log values committed to the Global Log become a part of the true state of the system.

B. Dealing with Crashes

The above procedure assumes that each delegate is active indefinitely. However, when a delegate process does crash, progress will be partially halted if a new one is not elected for the component, as the component will be no longer active. Thus, when enough delegate processes crash, the whole algorithm will be completely halted and cease to make any Global Progress due to liveness condition GL1. To deal with delegate crashes, we adopt the leader election algorithm used by Raft with a few modifications.

The leader election adopted from Raft here is actually a simplified version of the original. The election process still consists of three roles: leaders, analogous to delegates, candidates, and followers. All processes keep track of the highest term they have seen thus far. Each increment of the term counter corresponds to a single election execution. Thus, the term counter kept on each processes corresponds to the most recent election they have knowledge of or have participated in thus far. Candidates increment their term counter, vote for themselves, and send a RequestVote message to all processes within their component with their current term counter. If a receiving process has a term counter less what was received and has not yet voted in that term, it replies positively to the candidate. Upon receiving an acknowledgment from a quorum of processes in the component, the candidate escalates itself to the delegate and begins sending heartbeat messages to all processes in its component. At this point the process can act immediately in the Global Paxos Instance.

If a process fails to receive a heartbeat message within its election timeout, the process should transition to the candidate role and launch the delegate re-election protocol as described above. As proposed in Raft, the election timeout should be randomly generated at each process whenever a timeout is reached. This timeout value should be greater than the expected round-trip latency of messages within the same component and less than the expected time between process crashes.

C. Algorithm Pseudocode

To fully specify the Layered Paxos algorithm in a coded fashion, we provide the following pseudocode describing the new message protocols for Layered Paxos. In Fig 4. we show the state that all processes must store in addition to rules differing types of processes must follow in an execution of Layered Paxos. In Fig 5. we describe the set of message protocols required when electing a new delegate after a crash and those required for updating knowledge of the new delegate across the whole system. In Fig. 6 we set forth the Global Proposal protocol executed by a delegate in order to have a sequence of Local Log values committed to the Global Log. Finally, in Fig. 7 we state how processes are to respond to Global Paxos messages upon their receipt.

IV. Algorithm Analysis

In this section, we will prove the following theorems.

Theorem 4.1: Layered Paxos satisfies LS1, LS2, and LS3.

Theorem 4.2: Layered Paxos satisfies GS1, GS2, GS3, GS4, and GS5.

State on all Processes

- LocalMaxProp[]
- LocalAccNum[]
- LocalAccVal[]
- LocalLog[]
- GlobalMaxProp[]
- GlobalAccNum[]
- GlobalAccVal[]
- GlobalLog[]
- CurrentTerm
- VotedFor
- ComponentID
- SelfID
- CurrentDelegates[]
- LastCommittedSlot[]

Rules for All Processes

- Participate in Component's LPI
  - Propose Values from clients for LocalLog[]
  - Accept Values for LocalLog[]
  - Commit Values to LocaLog[]
- Respond to Global Paxos Instance Messages
- If election timeout has been reached without receiving a heartbeat message, transition to Candidate Role

Rules for Candidate Processes

- On Transition to Candidate Role
  - Increment CurrentTerm
  - VotedFor = SelfID
  - Reset election timer
  - Send RequestVote(CurrentTerm, SelfID) to all process IDs in component
- If receive positive votes from a majority of processes in component, transition to Delegate Role
- If Heartbeat received with term > CurrentTerm, revert to a normal process
- If election timeout, restart election as if just transitioned to Candidate Role

Rules for Delegate Processes

- On Transition to Delegate Role, send GlobalAlert(ComponentID, SelfID) to all process IDs in CurrentDelegates[]
- Periodically send Heartbeat(CurrentTerm) to all processes in component
- Propose sequence of LocalLog[] slots in the Global Paxos Instance by initiating the GlobalPropose() protocol
  - desiredGlobalSlot must be the first uncommitted slot in the delegate's GlobalLog[]
  - proposalSequence must be LocalLog[] values starting at LastCommittedSlot[CompID]

Fig. 4: State all processes must store in addition to rules the differing types of processes must follow in an execution of Layered Paxos.

GlobalPropose

Args

- proposalSequence
- desiredGlobalSlot

1. proposalNumber = generate proposal number
2. Send GlobalPrepare(proposalNumber, desiredGlobalSlot) to all process IDs in CurrentDelegates[]
3. If receive GlobalPromise from majority of delegates, continue, else retry proposal
4. validSequence = GetValidSeq(propSeq, PromiseResponses)
5. Send GlobalAccept(proposalNumber, validSequence, desiredGlobalSlot) to all process IDs in CurrentDelegates[]
6. If receive GlobalAccepted from majority of delegates, continue, else retry proposal
7. Send GlobalLearn(desiredGlobalSlot, validSequence) to all process IDs in CurrentDelegates[]

GetValidSeq

Args

- proposalSequence
- accNums
- accVals

1. If accVal == NULL for all accVals
   - return propSeq
2. Else
   - return argmax(accVals)

Fig. 5: Message protocols that are required in order to elect a new delegate and update knowledge of the new delegate across the system.

GlobalPropose

Args

- proposalSequence
- desiredGlobalSlot

1. proposalNumber = generate proposal number
2. Send GlobalPrepare(proposalNumber, desiredGlobalSlot) to all process IDs in CurrentDelegates[]
3. If receive GlobalPromise from majority of delegates, continue, else retry proposal
4. validSequence = GetValidSeq(propSeq, PromiseResponses)
5. Send GlobalAccept(proposalNumber, validSequence, desiredGlobalSlot) to all process IDs in CurrentDelegates[]
6. If receive GlobalAccepted from majority of delegates, continue, else retry proposal
7. Send GlobalLearn(desiredGlobalSlot, validSequence) to all process IDs in CurrentDelegates[]

GetValidSeq

Args

- proposalSequence
- accNums
- accVals

1. If accVal == NULL for all accVals
   - return propSeq
2. Else
   - return argmax(accVals)

Fig. 6: Global Proposal protocol executed by a delegate in order to have a sequence of Local Log values committed to the Global Log.

The general outline for our proof will be as follows. We first find it helpful to define multiple invariants to use in our proof. We then prove that Layered Paxos holds these invariants throughout its execution. Finally, we show that these invariants prove Theorem 4.1 and Theorem 4.2.

We define the following invariants regarding the Local Paxos Instance.

- **LI1:** A local acceptor can accept a local proposal numbered n iff it has not responded to a local prepare request having a number greater than n.
- **LI2:** For a set S of a majority of local acceptors, either no value has been accepted by any local proposal less than n, or the value v has been chosen and is the only thing that can be proposed at the Local Paxos Instance.

We also define the following invariants regarding the Global Paxos Instance.

- **GI1:** A global acceptor can accept a global proposal numbered n iff it has not responded to a global prepare request having a number greater than n.
- **GI2:** For a set S of a majority of global acceptors, either no value has been accepted by any global proposal less than n, or the value v has been globally chosen and is the only thing that can be proposed at the Global Paxos Instance.
- **GI3:** There is at most one process in each component who is recognized as a delegate by a majority of processes in its component.

GlobalPrepare

Args

- proposalNumber
- desiredGlobalSlot

Receiver Implementation

1. componentDelegate = currentDelegates[ComponentID]
2. If componentDelegate != SelfID, send UpdateDelegate(ComponentID, componentDelegate) to sender and exit
3. Send ForwardPrepare(proposalNumber, desiredGlobalSlot, CurrentTerm) to all processes in component (including self)
4. If receive response from majority of processes in component
   - maxProp = max(recvMaxProps)
5. If proposalNumber > maxProp:
   - maxAccNum = max(recvAccNums)
   - maxAccVal = max(recvAccVals)
   - Reply GlobalPromise(maxAccNum, maxAccVal) to sender
6. Else respond NACK(maxProp) to sender

ForwardPrepare

Args

- proposalNumber
- desiredGlobalSlot
- term

Receiver Implementation

1. If term < CurrentTerm, reply NACK(CurrentTerm)
2. Let:
   - maxProp = GlobalMaxProp[desiredGlobalSlot]
   - accNum = GlobalAccNum[desiredGlobalSlot]
   - accVal = GlobalAccVal[desiredGlobalSlot]
3. If proposalNumber > maxProp, GlobalMaxProp[desiredGlobalSlot] = proposalNumber
4. Reply (maxProp, accNum, accVal)

GlobalAccept

Args

- proposalNumber
- validSequence
- desiredGlobalSlot

Receiver Implementation

1. componentDelegate = currentDelegates[ComponentID]
2. If componentDelegate != SelfID, send UpdateDelegate(ComponentID, componentDelegate) to sender and exit
3. Send ForwardAccept(CurrentTerm, proposalNumber, validSequence, desiredGlobalSlot) to all processes in component
4. If received ACK from majority of processes in component, respond GlobalAccepted()
5. Else, send NACK(max(recvMaxProps))

ForwardAccept

Args

- term
- proposalNumber
- validSequence
- desiredGlobalSlot

Receiver Implementation

1. If term < CurrentTerm, reply NACK(CurrentTerm)
2. If proposalNumber >= GlobalMaxProp[desiredGlobalSlot]:
   - GlobalAccNum[desiredGlobalSlot] = proposalNumber
   - GlobalAccVal[desiredGlobalSlot] = validSequence
   - Reply ACK(proposalNumber)
3. Else reply NACK(GlobalMaxProp[desiredGlobalSlot])

GlobalLearn

Args

- desiredGlobalSlot
- commitSequence

Receiver Implementation

1. componentDelegate = currentDelegates[ComponentID]
2. If componentDelegate != SelfID, send UpdateDelegate(ComponentID, componentDelegate) to sender
3. Send ForwardLearn(desiredGlobalSlot, commitSequence) to component processes

ForwardLearn

Args

- desiredGlobalSlot
- commitSequence

Receiver Implementation

1. LastComittedSlot[commitSeq.CompID] = commitSeq.LastLocalSlot
2. GlobalLog[desiredGlobalSlot] = commitSequence

Fig. 7: Message protocols sent and received by delegates at the Global Paxos Instance to reach consensus on a set of values.

- **GI4:** The current delegate for a component must receive acknowledgement from a majority of processes in its component whenever acting as a Global Acceptor.
- **GI5:** A process can only make a Global Proposal if it is a contiguous sequence of values chosen in its component's Local Log.
- **GI6:** A Global Proposal can only contain a sequence of values if no value from that sequence has already been chosen for a slot in the Global Log.

With our invariants defined, the next step is to prove that they all hold within an execution of Layered Paxos.

Lemma 4.3: LI1 and LI2 hold in Layered Paxos.

Proof: As these invariants only refer to the Local Paxos Instance, we only need to look there for the proof. The Local Paxos Instance is consecutive Synod instances run between a subset of all the processes in the system following a quorum based on the number of processes participating in the Local Paxos Instance. As there are no modifications to Synod in this case, we know these two invariants hold from analysis done in the Paxos paper. Thus, Lemma 4.3 holds. ■

Lemma 4.4: GI1 and GI2 hold in Layered Paxos.

Proof: As these invariants only refer to the Global Paxos Instance, we only need to look there for the proof. The Global Paxos Instance is slightly modified consecutive Synod instances run between all delegates in the system. Since there are no relaxed modifications of Synod in this case with respect to how values are proposed or accepted, we know these two invariants hold from analysis done in the Paxos paper. Thus, Lemma 4.4 holds. ■

Lemma 4.5: GI3 holds in Layered Paxos.

Proof: Delegates are elected through a Raft-like leader election. As proven in Raft, this leader election results in either no or one process becoming a leader recognized by a majority of processes in the component. There are two cases to consider – the previous delegate has crashed or the previous delegate has not crashed. If the previous delegate has crashed, then after the election, there will be either no or one process who won the election, both of which are allowed by GI3. If the previous delegate has not crashed, after the election, there may be two processes who believe themselves to be the delegate. However, only the newly elected delegate will be recognized by a majority of processes in the component due to the previous delegate's lower term number. If the new delegate is not recognized by a majority of the component, then it could not have been elected delegate in the first place. Thus, in either case, we will have at most one process recognized as the delegate by a majority of processes in the component and Lemma 4.5 holds. ■

Lemma 4.6: GI4 holds in Layered Paxos.

Proof: When a delegate acts as a Global Acceptor – receives either a Global Prepare or Global Accept message – it must first forward this message to all processes in its component. The delegate may not respond to this message until it hears a positive response from a majority of processes in it component. If the delegate does not receive a majority of responses, it may not respond to the Global Paxos message it received. Thus, it cannot act as a Global Acceptor from the point of view of the Global Paxos Instance and Lemma 4.6 holds. ■

Lemma 4.7: GI5 holds in Layered Paxos.

Proof: Delegate processes may only make a Global Proposal if the value proposed is a contiguous sequence of values from the Local Log. As delegate processes are the only ones who are able to make Global Proposals, the only Global Proposals made will be a contiguous sequence of values from a Local Log. Thus, Lemma 4.7 holds. ■

Lemma 4.8: GI6 holds in Layered Paxos.

Proof: When making a Global Proposal, the sequence of Local Log slots proposed must start with the first slot – to the delegate's knowledge – that has not been committed in the Global Log. It is also required that a delegate proposes for the first, uncommitted Global Log slot it has knowledge of. When making a Global Proposal for Global Log slot i, a delegate can fall into one of two cases – slot i has had no sequence committed or the delegate did not learn that slot i is already committed. In the first case, slot i is the true first, uncommitted slot in the Global Log, which means the delegate has knowledge of every value committed in the Global Log before slot i, otherwise the delegate would be forced to propose for some Global Log slot $j < i$. Since the delegate has knowledge of all committed values in the Global Log, the start of the sequence proposed cannot have been committed in the Global Log. If it was, the delegate would have knowledge of it and would not propose it in the sequence. In the second case, it is possible that the delegate's proposed sequence contains a value already committed in the Global Log. However, a sequence has already been committed to Global Log slot i, so the delegate's sequence will not be chosen nor committed to Global Log slot i. In both cases that a delegate can propose in, no sequence containing a value already committed in the Global Log will be committed again. Thus, Lemma 4.8 holds. ■

Now that it has been proven that all the invariants hold in Layered Paxos, all that is left is to show that these invariants satisfy our safety properties. We first show that Lemma 4.3 proves Theorem 4.1.

Proof: Lemma 4.3 proves that Layered Paxos maintains LI1 and LI2. In the Paxos paper, it is proven that two similar invariants satisfy the same three safety properties in Paxos. Thus, without loss of generality, it follows that an algorithm maintaining LI1 and LI2 satisfies LS1, LS2, and LS3. Since Lemma 4.3 shows that Layered Paxos maintains LI1 and LI2 and these two invariants satisfy LS1, LS2, and LS3, Theorem 4.1 is proven. ■

Finally, we show that Lemmas 4.4, 4.5, 4.6, 4.7, and 4.8 prove Theorem 4.2.

Proof: First, we show that GI1, GI2, GI3, and GI4 satisfy GS1, GS2, and GS3. Similarly to how Theorem 4.1 was proven, if GI1 and GI2 are maintained, then GS1, GS2, and GS3 will be satisfied as long as the Global Paxos Instance effectively behaves the same way Paxos does. Within the Global Paxos Instance, no modifications have been relaxed in

terms of proposing values to be committed. The other scenario to assert is that no relaxations have been made in terms of accepting values. From GI3 and GI4, there is at most one process recognized as a delegate by a majority of processes in a component and a delegate must receive acknowledgment from a majority of processes in its component when acting as a Global Acceptor. Thus, it follows that no more than one process from a component can act as a Global Acceptor at any given time. Since no requirements on proposing or accepting have been relaxed and the number of Global Acceptors remains constant at any given time, GI1, GI2, GI3, and GI4 satisfy GS1, GS2, and GS3.

Next, we show that GI5 and GI6 satisfy GS4 and GS5 respectively. From GI5, the only value that can be proposed at the Global Paxos Instance is a contiguous sequence of values from a component's Local Log. If this is all that can be proposed, it trivially follows that this is all that can be chosen for a Global Log slot as only proposed values can be chosen. From GI6, a Global Proposal can only contain values such that no value has already been committed to the Global Log. Again, it trivially follows that no value can be committed twice in the Global Log if it is never proposed twice in a scenario where it may be chosen. Thus, GI5 and GI6 satisfy GS4 and GS5 respectively.

Since we have shown that GI1, GI2, GI3, GI4, GI5, and GI6 satisfy GS1, GS2, GS3, GS4, and GS5, it follows that Lemmas 4.4, 4.5, 4.6, 4.7, and 4.8 prove Theorem 4.2. ■

We have now proven Theorems 4.1 and 4.2 through Lemmas 4.3, 4.4, 4.5, 4.6, 4.7, and 4.8. Thus, our analysis of the correctness of Layered Paxos in satisfying our safety properties is complete.

V. Experiments

A. Experimental Setup

To test the performance of Layered Paxos in a hierarchical system against Paxos, we performed experiments in the Amazon Web Services (AWS) environment. In our setup, we allocate each component of processes within different regions including the United States, Canada, and Europe to simulate a high-latency, wide area network. In particular, the six regions used were us-east-1, us-east-2, us-west-1, us-west-2, ca-central-1, and eu-central-2. Each process used a t2.micro EC-2 instance within its component's AWS region running on the Ubuntu 20.04 LTS operating system. Since the latency between servers in the same AWS region is roughly 1 to 5 milliseconds and that between AWS regions is somewhere between 200 to 300 milliseconds, we feel this is an accurate representation of the target systems for Layered Paxos.

For our implementation, we developed both Paxos and Layered Paxos applications in Python 3.8.3. All messages passed between processes utilized UDP protocol to simulate the unreliable messaging described in the System Model. We set the heartbeat timeout for our leader election mechanism to a randomly-generated time between 300 and 500 milliseconds as described in the Raft algorithm. To simulate client requests, a single process from each component was designated as a proposer. This process continuously proposed a new value to its component when its previous proposal had been chosen or rejected. Both Local and Global proposers were allowed to retry a proposal at most three times before resetting the proposal process, potentially updating the log slot proposed for with knowledge gained during the previous proposal process.

Two main metrics of performance were measured to judge the performance between Paxos and Layered Paxos – throughput and request latency. We define *throughput* as the number of client requests committed to the true state of the system every second – the Global Log in Layered Paxos and the single, replicated log in Paxos. In Paxos, we define *request latency* as the time a client request takes to be committed to the log. Since Layered Paxos has two distinct log layers, we find it useful to define two types of request latency. *Front-end request latency* is defined as the time a client request takes to be committed in the Local Log with *back-end request latency* as the time a client request takes to be committed to the Global Log. In order to measure throughput and all types of request latency, three events are logged – when a client request is proposed at the Local Paxos Instance, committed to the Local Log, and committed to the Global Log. In addition to comparing these metrics between Paxos and Layered Paxos, we test how these metrics are affected when a delegate is crashed and the component is forced to elect a new one.

B. Throughput Results

We ran the implementation described above for both Paxos and Layered Paxos to see how the throughput of each varies as a constant number of processes is split evenly among a varied number of components. In our tests, we took 12 processes and split them evenly across 2, 4, and 6 components. Each test was run for 100 seconds before results were compiled and averaged. Since each component had a single proposer

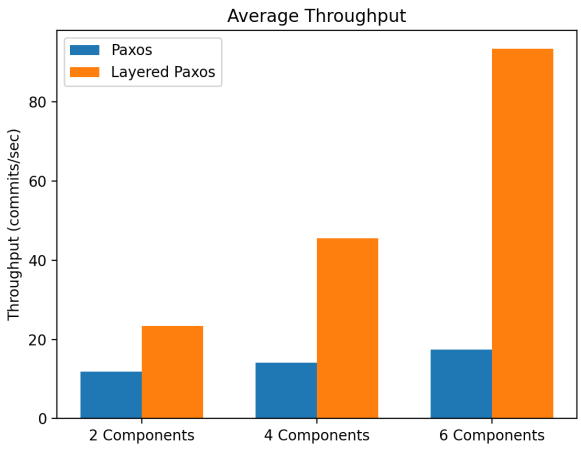


Fig. 8: Average throughput for Paxos versus Layered Paxos with each component in a different AWS region. 12 processes are split evenly between 2, 4, and 6 components for a total of 6, 3, and 2 processes in each component respectively.

designated, the number of proposers in the system increased as the number of components increased.

The results of our throughput test are depicted in Fig. 8. To avoid many concurrent proposals between components in Layered Paxos, delegate processes would sleep for a randomly-generated time between 50 and 150 ms before proposing a batch size of at least 15 entries. As seen below, Layered Paxos outperforms Paxos in all three component configurations. The increase in throughput ranges from roughly 2x with 2 components, to 3x with 4 components, to almost 5x with 6 components. In all scenarios, Layered Paxos was able to make use of the fast intra-component communication while mitigating the effects of the high-latency inter-component communication.

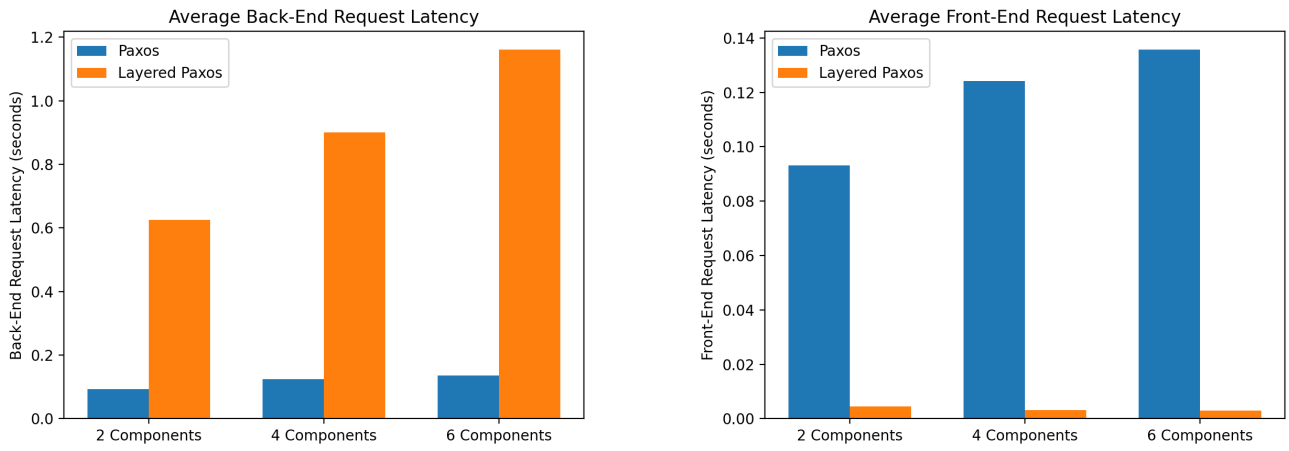
C. Request Latency Results

In order to compare the request latency defined for Paxos and the two types defined for Layered Paxos, we ran tests identical in setup to those described above to measure the throughput of each. Some systems exist such that a client's request may not be serviced for reason's other than network connectivity or system failure. In other words, a client's request may not be able to be serviced due to some other request that has already been serviced such as in an airline reservation system. In the case of Layered Paxos, if there are two such mutually exclusive client requests chosen in different Local Logs, the request that is committed first to the Global Log is the one that is serviced – the other will be rejected by the system's application layer. Thus, it is not until a client request is committed to the Global Log that a confirmation could be sent back to the client. In these types of systems, back-end request latency would be the metric of choice.

The results of the tests comparing back-end request latency between Layered Paxos and Paxos are depicted in Fig. 9a. As seen in the figure, the drawback of a higher throughput in every configuration seen in Layered Paxos over Paxos is a correspondingly larger back-end request latency. However, this trend makes sense. In Paxos, client requests are serviced as soon as the their proposal is committed in the single replicated log. On the other hand for back-end request latency in Layered Paxos, once a client request is committed to the Local Log, it must then wait for at most 14 other client requests to be committed before the batch is proposed and chosen at the Global Paxos Instance. Thus, it follows that the average back-end request latency in Layered Paxos would be multiple times that of Paxos plus the time of a single Global Paxos Instance. When the number of components is scaled up, there is a possibility for more concurrent Global Proposals at once, which would again increase the back-end request latency in Layered Paxos.

In comparison to the back-end based systems described above, other systems exist such that there is no case where two client requests will be mutually exclusive. In systems where this is the case, the metric of consideration would be front-end request latency. As soon as a client request is committed to a component's Local Log, given that liveness conditions hold for long enough, the client's request will eventually be committed to the Global Log and become a part of the system state. As such, a client could be sent a confirmation of their request being accepted upon it being chosen by the Local Paxos Instance it was sent to.

The results of the tests comparing front-end request latency between Layered Paxos and Paxos are shown in Fig. 9b. As seen in the figure, the average front-end request latency is much less for Layered Paxos than Paxos. Again, this follows as each component in Layered Paxos shares a fast communication network, so consensus in the Local Layer is always fast. In Paxos, consensus must take place across multiple components



(a) Average back-end request latency

(b) Average front-end request latency

Fig. 9: Request Latency compared between Paxos and Layered Paxos with each component in a different AWS region. 12 processes are split evenly between 2, 4, and 6 components for a total of 6, 3, and 2 processes in each component respectively.

which have a slower network, so reaching consensus on a client request of course takes longer the more components there are. In contrast to back-end request latency in Layered Paxos, front-end request latency is not affected by the number of components, but instead affected by the number of sites within each component. This is attributed to the fact that more sites need to be contacted within a component to reach consensus as more are added to said component.

With the results from above, multiple things needs to be considered when choosing the number of sites and components running Layered Paxos. If the target system is one that falls under the back-end request latency criteria, a balance between throughput and client response time will need to be taken into account when determining the number of components. If the target system falls under the front-end request latency criteria, the number of components may be scaled to increase throughput without affecting client response time.

D. Delegate Crash Results

In order to test the effect of delegate crashes on Layered Paxos, we ran a test involving 2 components with 6 processes each for 80 seconds. We allowed all processes to run for roughly 47 seconds before crashing the delegate of one of the components. In Fig. 10a, we plot the throughput of the system versus the experiment time. The delegate is crashed at 47 seconds where the red dashed line is seen on the graph. Following the delegate crash, the throughput of the system drops to zero for roughly 3.5 seconds before a new delegate is elected and the throughput spikes. After the spike, the throughput settles to the steady state it had been before the crash. In Fig. 10b, the back-end request latency is plotted versus the experiment time. Complementary to what is seen in Fig. 10a, Fig. 10b shows a large spike in request latency when the delegate crashes and the throughput drops to 0. After a new delegate is elected roughly 3.5 seconds later, the request latency drops back down to the steady state value it had been previously. Thus, by implementing a Raft-like leader election, Layered Paxos can withstand delegate failure and recover without affecting the system too much granted that delegate crashes do not happen very often.

VI. Conclusion

In this paper, we presented Layered Paxos, which aims to increase the throughput of consensus in globally scaled systems. Layered Paxos is designed for use in systems with a two-fold communication network consisting of disjoint components sharing a fast communication network all connected by a high latency communication network. We defined safety properties generally desired in consensus and prove that Layered Paxos satisfies them. In addition, we defined liveness conditions required for the algorithm to make progress. Finally, experimental tests were run in AWS to simulate Layered Paxos in these hierarchical systems. It was shown that Layered Paxos can achieve up to 5x the throughput of Paxos. We also show that Layered Paxos can tolerate delegate crashes using a Raft-like leader election without an overly adverse effect on the system. In the future, we plan to explore extending Layered Paxos with optimized versions of Paxos, like Fast Paxos [10] or other such variants, as a building block.

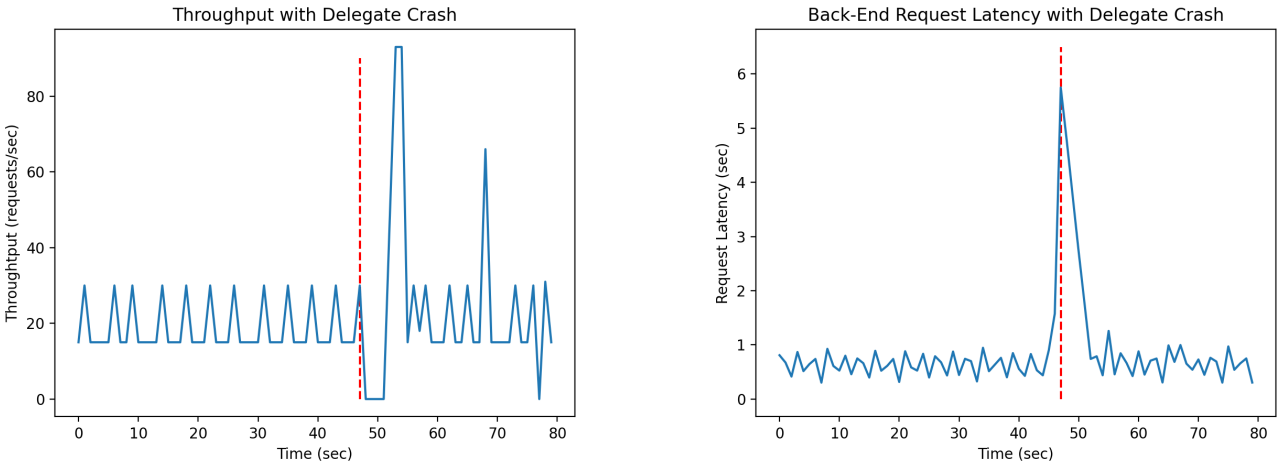
References

[1] Lamport, Leslie. "Paxos made simple." ACM Sigact News 32, no. 4 (2001): 18-25.

[2] Lamport, Leslie. "The part-time parliament." In Concurrency: the Works of Leslie Lamport, pp. 277-317. 2019.

[3] Liu, Fagui, and Yingyi Yang. "D-Paxos: building hierarchical replicated state machine for cloud environments." IEICE TRANSACTIONS on Information and Systems 99, no. 6 (2016): 1485-1501.

[4] Dwork, Cynthia, Nancy Lynch, and Larry Stockmeyer. "Consensus in the presence of partial synchrony." Journal of the ACM (JACM) 35, no. 2 (1988): 288-323.



(a) Throughput vs. Time

(b) Back-End Request Latency vs. Time

Fig. 10: Delegate crash experiment run with 2 components each with 6 processes. Delegate of one component is crashed at 47 seconds denoted by the red dashed line in each plot.

[5] Chandra, Tushar Deepak, and Sam Toueg. "Unreliable failure detectors for reliable distributed systems." Journal of the ACM (JACM) 43, no. 2 (1996): 225-267.
[6] Castiglia, Timothy, Colin Goldberg, and Stacy Patterson. "A Hierarchical Model for Fast Distributed Consensus in Dynamic Networks." arXiv preprint arXiv:2004.06215 (2020).
[7] Ongaro, Diego, and John Ousterhout. "In search of an understandable consensus algorithm." In 2014 USENIX Annual Technical Conference (USENIXATC 14), pp. 305-319. 2014.
[8] Sanderson, David, and Jeremy Pitt. "Institutionalised Paxos Consensus." In ECAI, pp. 714-719. 2012.
[9] Fischer, Michael J., Nancy A. Lynch, and Michael S. Paterson. "Impossibility of distributed consensus with one faulty process." Journal of the ACM (JACM) 32, no. 2 (1985): 374-382.
[10] Lamport, Leslie. "Fast paxos." Distributed Computing 19, no. 2 (2006): 79-103.