University of Texas at Austin

Project Report

CS380L Advanced Operating Systems

Porting Fast Paxos

Authors:
Ankit Goyal
Cheng Fu
Gurbinder Gill

Supervisor:
Dr. Emmett Witchel

December 4, 2014

Abstract

Paxos is a protocol for solving consensus in an asynchronous environment that admits crash failures. *Consensus* is a process of agreeing on one result among group of participants. *Classic Paxos*[1] proceeds over several rounds to decide on a sequence of commands. In this project, we port an existing implementation[2] of a variant of *classic paxos* called *Fast Paxos*[3]. *Fast Paxos* reduces the number of messages between client request and response by 2 [1].

1 Introduction

The problem of agreeing on a sequence of operations or values proposed by different processes is known as the *Distributed Consensus* problem. From *FLP* result[4], it is impossible to solve consensus in an asynchronous distributed system even if a single process fails by permanently stopping or if a distributed system suffers a *Partial Failure*, in which processes may stop and recover later. Paxos is an algorithm that gets around this problem by making sure that the system doesn't violate the safety requirements during periods when system behaves asynchronously and is certain to make progress (liveliness) if the system behaves partially synchronously for periods long enough to satisfy the progress requirements.

Classic Paxos and Fast Paxos are most widely studied algorithmic solutions to the problem of distributed consensus. Fast Paxos has smaller theoretical message latency, therefore is faster, but Classic Paxos is more resilient and hence can tolerate more failures. In this project we ported the Fast Paxos algorithm on the simulator to study its behavior in different scenarios.

2 Protocol Overview

2.1 Paxos Roles

In our implementation of Fast Paxos, each server can take following roles:

1. **Proposers** receive request from clients, associate it with the next instance id (slot number) and broadcast accept $< Req, InstanceID, Ballot >$ message to all acceptors

2. **Acceptors**, in the **fast round**, upon receiving accept message from proposer, they apply that request and broadcast learn message to all learners. In the **classic round**, upon receiving a prepare message from leader, they apply it to log and reply with a promise message; upon receiving anyval message from leader, they apply it to log.

[1] in case of no conflicts

3. **Leader**, one of the proposers assumes the role of leader. In the **classic round**, it broadcasts prepare message to all acceptors and wait for a quorum number of promise messages, and then broadcasts anyval message to acceptors. In the **fast round**, it detects the potential conflicts and resolves it by initiating a classic round on given Instance ID

4. **Learners** learn the accepted values from acceptors and check the quorum condition, once reached, they deliver the accepted value in the order of Instance ID

It must be pointed out that the role of the leader in Fast Paxos is totally different from that in Classic Paxos. In Classic Paxos, the leader is responsible for serializing commands in global order by assigning a unique Instance ID (namely timestamp or slot number) when proposing a new value. While in Fast Paxos, the leader is responsible for proposing values when a conflict occurs, that is, when two proposers are trying to propose different values for the same Instance ID and no one reaches the quorum. Leader is checking the progress of the protocol periodically and doing arbitration if it sees a conflict. To be able to detect the conflicts, the leader must also be a learner.

2.2 Message Flows

The normal-case communication pattern in Paxos protocol is proposer $\rightarrow$ leader $\rightarrow$ acceptors $\rightarrow$ learners. In Fast Paxos, a proposer sends its proposal to acceptors directly. bypassing the leader and saving one message delay. So the message flow is proposer $\rightarrow$ acceptors $\rightarrow$ learners.

2.3 Normal Operation

2.3.1 Phase 1a

As shown in **Figure 1**, Leader on start up and on regular timeouts, sends a phase 1a batch message $PREPARE < ballot, iids >$ to acceptors for N (configurable) $iids(instance\_id)$. Ballot number used in this message is bigger than any ballot number used by any other proposer.On receiving this batch message, acceptors do the following:

1. if record is not found in the acceptor log, it needs to be created with the received ballot number.

2. if record exists, it updates the ballot number in the record found in the log to the larger ballot number (received or already present in the record).

After processing this message, it sends back the $PROMISE < iid, ballot, value >$ to the leader, where value could be either $NULL$ or some accepted value for that iid.

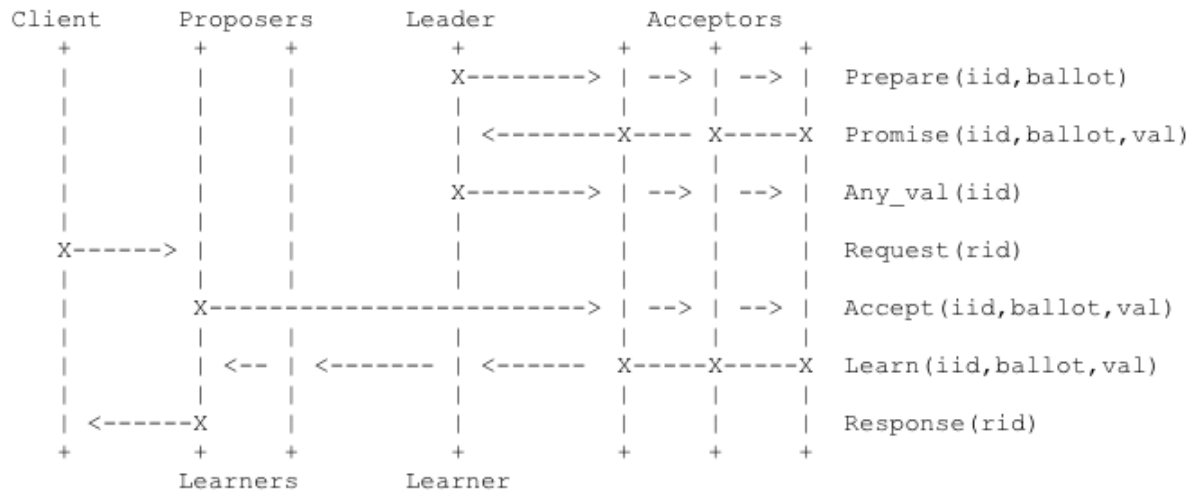

Figure 1: Normal operation of Fast Paxos Protocol

2.3.2 Phase 1b

Leader on receiving phase 1b batch message $PROMISE < iid, ballot, value >$ from acceptors do the following:

1. it checks for the quorum condition for a given iid, if true it broadcasts the $ANY\_VAL < iid >$ batch message. This pre-executes the $Phase1$ of Classic Paxos and sets stage for Fast Paxos.

2. if the quorum condition is not reached, on timeout it resends the $PREPARE$ batch message.

2.3.3 Phase 2a

Acceptors on receiving $ANY\_VAL < iid >$ message do the following:

1. they update their persistent logs and set any_enabled for each received iid to true, which signifies that they can accept values from proposers directly (fast round).

2. they wait for proposers to propose values.

Proposers on receiving request from client, do the following:

1. they assign their ballot number($fixedballot$) and send an $ACCEPT < iid, ballot, value >$ message to all acceptors directly(fast round).

2.3.4 Phase 2b

On receiving accept message, acceptors do the following:

1. case 1: if any enabled, they accept the message by updating their log.
2. case 2: if any is not enabled and the accept is sent by the leader, they accept the value if the ballot number is greater than the one in the record.
3. in all other cases, the message is ignored (could be due to conflicts, retransmission or delayed message).

After updating the log, acceptors broadcast the $LEARN < iid, value, ballot >$. Learners (including leader and proposers) on receiving the $LEARN$ message, do the following:

1. they update their in-memory log to check for duplicates and quorum condition.
2. On satisfying the quorum condition, they execute the request in the order of *iid*, deliver it (update the proposer state) and send back the response to the client.

2.4 Choosing Quorum:

For Fast Paxos, we need a bigger quorum than Classic Paxos because for a single round multiple values can be proposed for a given instance. For the leader to resolve the conflict and guarantee the correctness it needs bigger quorum size. Here, correctness means that once leader picks a value for an instance, no other value can reach the quorum condition on any learners. To tolerate f failures, Fast Paxos needs at least $3f + 1$ servers and the quorum size is $2f + 1$.

2.5 Resolving conflicts:

As shown in **Figure 2**, conflicts may happen if all of the following events happen:

1. All acceptors received ANY_VAL message for a given *iid*
2. More than two proposer send $ACCEPT$ message with different value for that *iid*
3. None of those $LEARN$ messages reach the quorum condition, which depends on the arrival order of $ACCEPT$ messages on acceptors' side (acceptor only accepts the value in the first $ACCEPT$ message received).

The leader detects the progress of the protocol periodically using a timeout mechanism, if nothing gets delivered from learner,

1. It broadcasts a $PREPARE < ballot, currrent\_iid >$, where current_iid is the next iid to deliver by learner, and *ballot* is a higher ballot number to indicate its conflict resolution purpose.

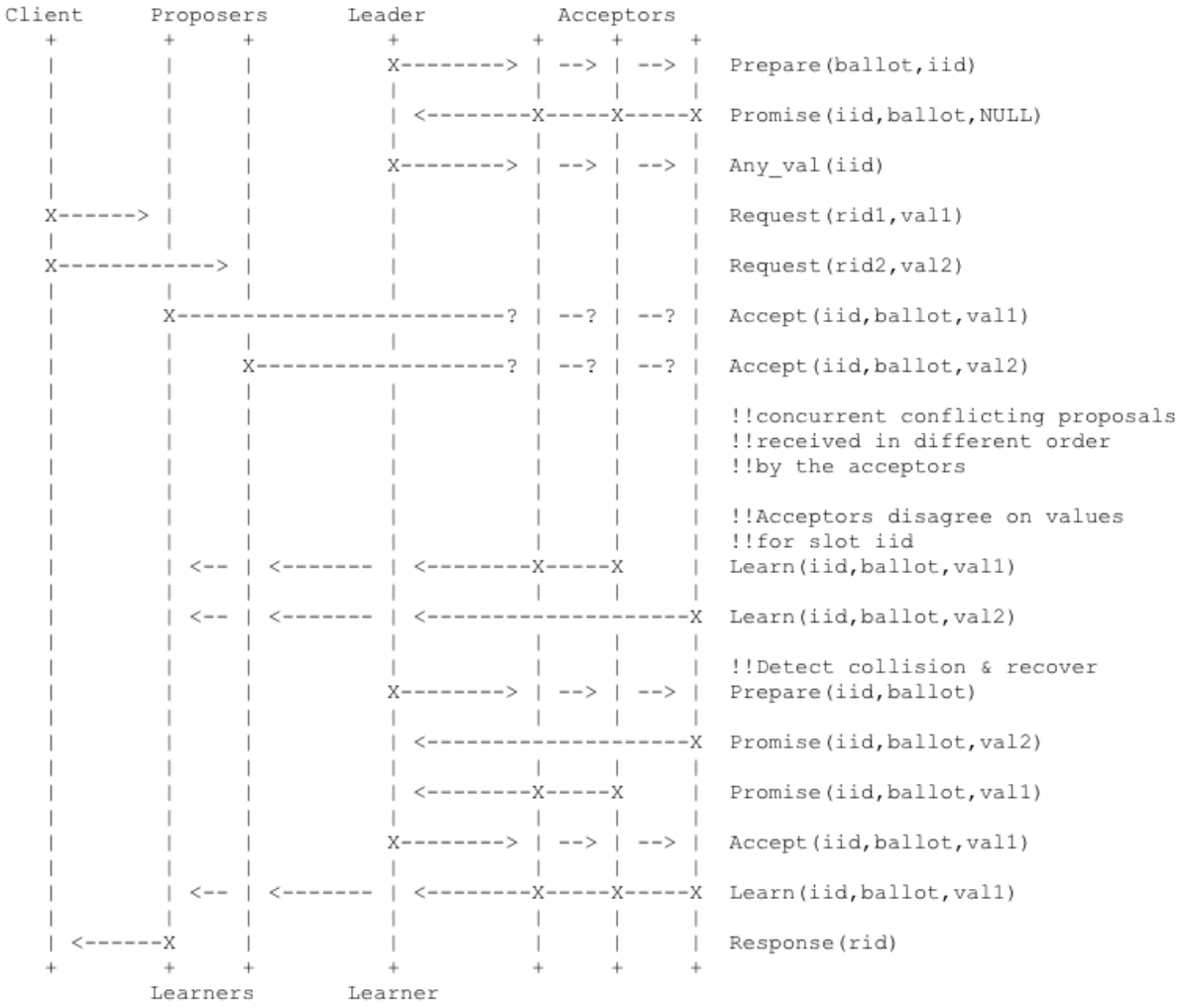

Figure 2: Conflict Resolution of Fast Paxos Protocol

2. Acceptors upon receiving $PREPARE < ballot, currrent\_iid >$, reply with $PROMISE < iid, value\_ballot, accepted\_value >$.

3. The leader upon receiving the $PROMISE < iid, value\_ballot, accepted\_value >$, updates its in-memory records and check the quorum condition; if quorum condition is true, it picks the value with majority votes in quorum, if none such value is found, it can pick the value with the highest *value_ballot*; then it broadcasts the $ACCEPT < iid, ballot, chosen\_value >$ to acceptors

4. Acceptors on receiving this $ACCEPT < iid, ballot, chosen\_value >$ mes-

sage from the leader; the condition of case 2 in 2.3.4 will be true and the value is accepted and learn message will be broadcasted as normal.

5. Eventually one of the proposers will get notified that its proposal is granted, other proposers (losers) will retry with next *iid* by broadcasting $ACCEPT < iid + 1, ballot, value >$ message to acceptors.

2.6 Retransmissions:

There are several retransmissions in the protocol to ensure liveness and tolerate message losses:

2.6.1 Leader $PREPARE$ retransmission:

Leader maintains a state of all the expected $PROMISE$ messages for all $PREPARE$ messages. On timeout, leader rebroadcasts the $PREPARE$ messages for those $PROMISE$ messages for which the quorum condition is not satisfied.

2.6.2 Learners $LSYNC$ message:

Learner maintains a state of all the expected $LEARN$ messages, and sends $LSYNC$ message for all *iids* for which quorum condition is not satisfied.

Acceptor on receiving this message will reply with $LEARN < iid$, $accepted\_value$, $ballot >$ for each iid.

2.6.3 Proposers $ACCEPT$ retransmission:

Proposer maintains the state of the current value it is trying to propose and retransmit it on timeout. The timeout will be cancelled if the value gets delivered.

2.6.4 Client $REQUEST$ retransmission:

Each proposer can only handle one client at a time, so if a proposer receives requests from another client while it is processing a request, it simply drops the message. Client on timeout, retransmits the $REQUEST$.

3 Implementation Differences

1. In *Libfastpaxos*, each role is a separate process. Each proposer spawns a learner thread and main thread acts as proposer. One of the proposers (pre-defined) acts as a leader and each acceptor is a process of its own. In our implementation, there's only one entity called `paxserver` that acts as a leader, learner, acceptor and proposer.

2. They use real time to do timeouts using `libevent` library whereas simulator has a different notion of time and we use counters in each instance to do timeouts.

3. They use Berkley DB as a persistent log in acceptors whereas we use in-memory `paxlog` as our acceptor log.

4. In *Libfastpaxos*, client and proposer are running as two threads in same process and using shared variables for synchronization. Hence, they cannot have multiple clients connecting to same proposer, and each proposer has a predefined FIFO request queue. In our implementation, client is independent from the paxos server system and communicates using message passing. Request may duplicate over different proposers. In proposer's deliver callback method, we use request id and client id instead of proposer's id to detect if proposer needs to re-propose its request with a higher instance id.

4 Timeouts

4.1 Different types of timeout

The progress of the protocol depends on the several timeouts being used by the protocol. There are four timeouts being used:

1. **Leader Phase 1 timeout:** Leader uses this timeout to send the phase 1a ($PREPARE$) batch messages to pre-execute the phase 1 of classic paxos and guarantee the progress of phase 1 in case of message loss.

2. **Leader Phase 2 timeout:** Leader uses this timeout to detect the progress of phase 2. If no progress has been made, it assumes that the conflicts happened for the current iid and it initiates the conflict resolving algorithm (section 2.4).

3. **Proposer Timeout:** Proposer uses this timeout to retransmit the $ACCEPT$ message if it doesn't see a progress for current iid (i.e., if no value has been accepted for the current iid).

4. **Learner Timeout:** Learner uses this timeout to sync its state with acceptors to learn the accepted values for iids that it might have missed.

4.2 Caveats:

According to the original implementation of `libfastpaxos` for the protocol to make progress,

1. the message delays should not be of the same order for phase1 and phase2 timeouts.(delay $<<$ timeout interval)

2. when running in simulator, assumption(1) above requires the number of messages that can be processed by each server per tick (say k messages) should be proportional to effective message delay, as determined by shuffle, delay and server number. The total number of messages per instance is $O(N^2)$, where N is the number of paxos servers. For each server, if we do a 100 shuffle, it will introduce a $O(N)$ tick message delay. To make the protocol work, we set this k to be (N).

5 Limitations and Corner Cases:

1. Our paxos server system implementation does not count client as a learner, so client has no information about the granted *iids* and next iid to try. That's the reason for client proposing value through proposers and not directly to acceptors.

2. If a client times out and tries a different proposer but the previous request of client succeeds and the new proposer fails to receive the update for that request, it will re-propose the value with new *iid*. One possible solution for this is for proposer to sync with acceptors to learn about the different requests for this client. Since we maintain a finite size log, we couldn't implement this within given time.

3. `Libfastpaxos` doesn't implement view change algorithm. It assumes that leader is always alive and it is unique. There's no leader election protocol.

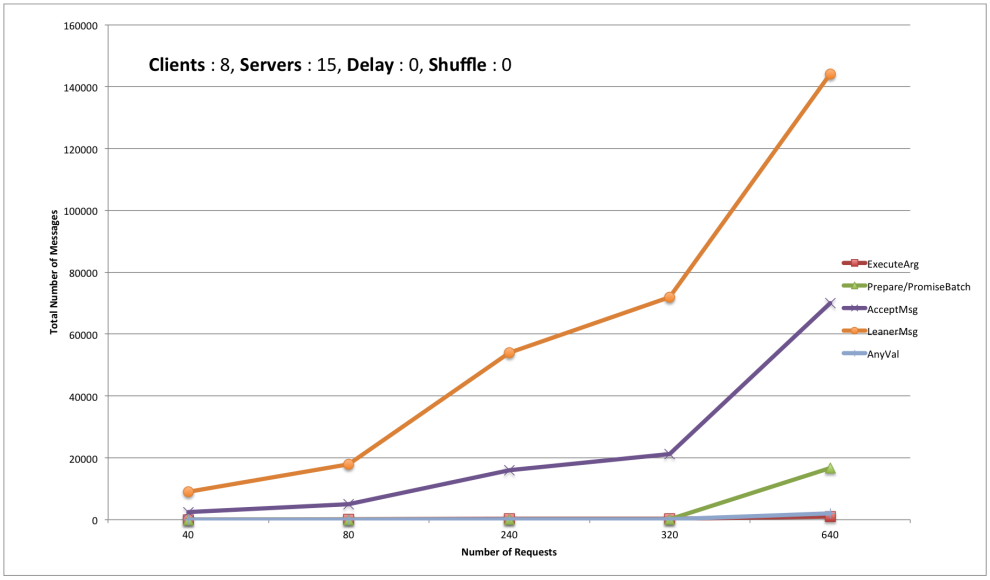

Figure 3: Total number of various messages as the number of requests increase.

6 Test and Results:

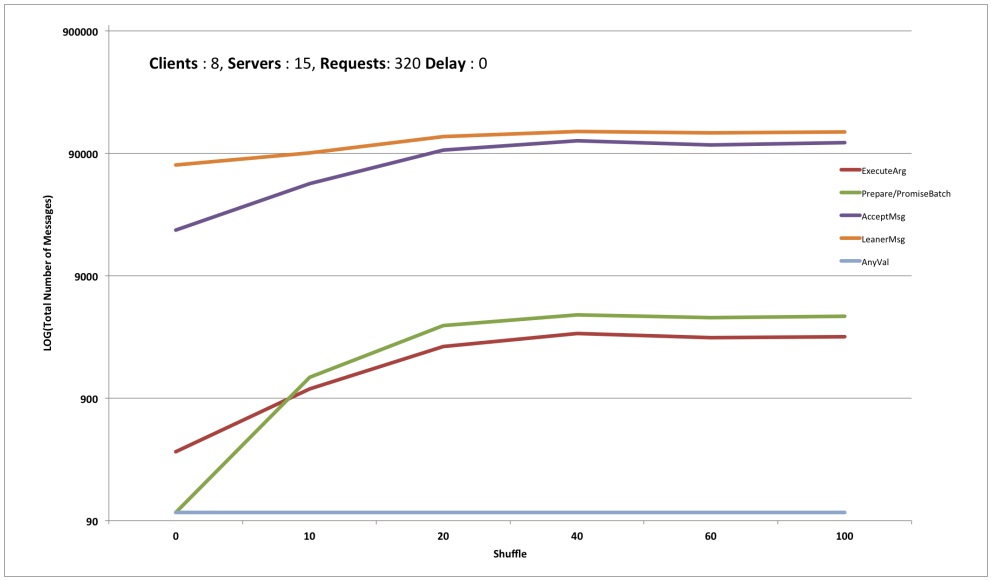

Figure 4: (Logarithmic scale) Total number of various messages as the shuffle increases.

6.1 Base Case: `2 clients, 3 servers, no delay or shuffle, requests 4`

This is the bare minimum configuration since there are no delays or shuffle, there will be no conflicts (conflict condition 3 of 2.4 will not be true).

Figure 7(in appendix), shows the response being sent to clients from servers (proposers). It shows that all 6 requests from different clients are executed in a fixed global order. Figure 8(in appendix) shows that the logs are same on all 3 servers (Learners).

6.2 Conflict Resolution

Figure 5 shows that as the shuffle increases, the contention for a slot increases leading to higher number of conflicts. It can also be observed in figure 4 that the number of $PREPARE$, $PROMISE$ and $ACCEPT$ messages increase with shuffle as leader resolves conflicts by sending $PREPARE$ and $ACCEPT$ messages.

6.3 Message Latency:

It can be seen from Figure 6 that as the delay increases the average number of messages passed for a single request also increases since there are more collisions and retransmissions due to various timeouts.

6.4 leader paused and unpaused:

Figure 9(in appendix) shows the case where leader is paused and then unpaused. When leader is paused, all live servers (proposers) keep retrying the accept message and as soon as leader comes back online, the protocol makes progress.

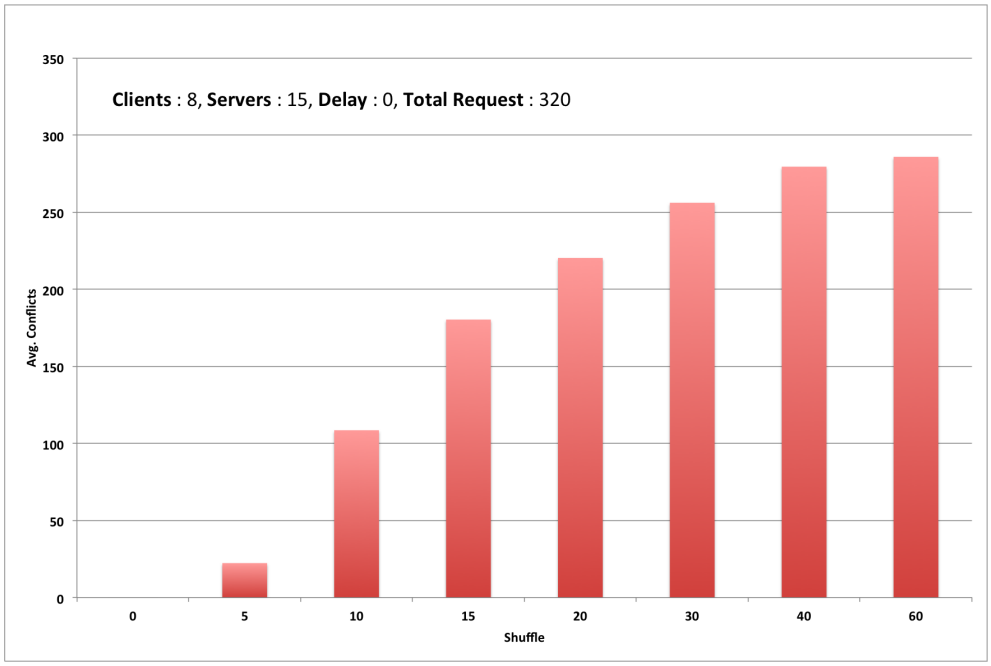

Figure 5: Avg. number of conflicts increases as shuffle increases.

Figure 5 shows that as shuffle increases average number of conflicts increase drastically from 0 to 20 and then it increases gradually. To contain the randomness of shuffle, all values are average of runs over 5 different seeds.

Figure 3 shows that the number of *Phase*1 batch messages grows considerably slower than other messages in the system. This is where Fast Paxos benefits by pre-executing the Phase1 as compared to Classic Paxos.

7 Challenges

1. The `libfastpaxos` implementation has no documentation or comments in the code, so it took us long time to understand the code.

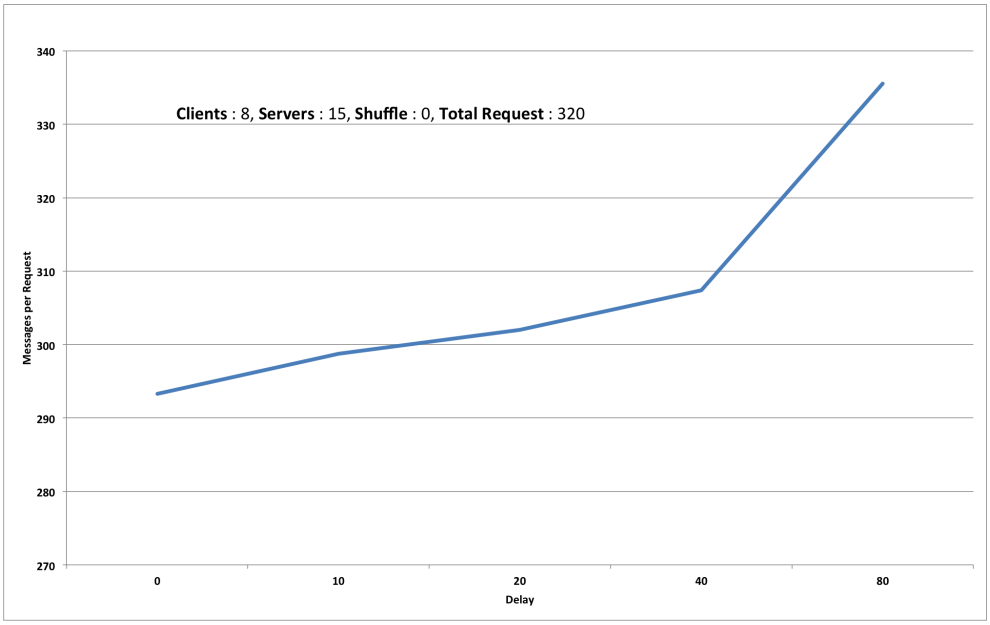

Figure 6: Avg. number of messages per request as delay increases.

2. The implementation is not consistent with the terminology used or implementation given in the Fast Paxos paper. Use of ballot and instance id (slot number) somewhat corresponds to Paxos Made Moderately Complex[5] by Robbert V. Rannese but the roles used in implementation (learners, proposers, etc.) are logically different from the ones used in Rannese's paper(scout, commander, leaders, etc.).

3. `libeventpaxos` used `libevent` to schedule timeouts which is based on real time. It took us time to relate the libevent timeouts to time (ticks) in simulator.

4. The relationship between different timeouts is non-trivial and would cause the program to hang.

8 Future Work:

1. **Smart Learners:** `Libfastpaxos` is using the fixed timeout intervals to detect the conflicts which is not optimal and can be optimized by making learners report conflicts to the leader as soon as they detect the conflict condition (multiple values being learned for same iid and quorum is not possible for any of the values).

2. Currently when a client connects to a proposer that is already working on another request, the proposer drop the client's request without respond-

ing anything. Client eventually timeout and retries a random proposer. This could be optimized by buffering the client requests at the proposer or sending back a fail message so that client can continue with another proposer without waiting for the timeout.

Time spent on the project ≈ 60 hours each

References

[1] Lamport, Leslie (May 1998). "The Part-Time Parliament" *ACM Transactions on Computer Systems 16 (2): 133–169*

[2] Marco Primi and Daniele Sciascia. "libfastpaxos" *http://libpaxos.sourceforge.net/paxos_projects.php\#libfastpaxos*

[3] Lamport, Leslie (July 2005). "Fast Paxos"

[4] Fischer, Michael J; Nancy A. Lynch; Michael S. Paterson (April 1985) "Impossibility of distributed consensus with one faulty process". *Journal of the ACM 32*

[5] Robbert van Renesse (March 2011). "Paxos Made Moderately Complex"

A Appendix

```
% ./pax -f --shuffle 0 -c 2 -s 3 -r 4 -l DEBUG --delay 0 | grep response
Sending response back for client: 5 for rid: 1 iid: 0
Sending response back for client: 5 for rid: 1 iid: 0
Sending response back for client: 5 for rid: 2 iid: 1
Sending response back for client: 5 for rid: 3 iid: 2
Sending response back for client: 4 for rid: 1 iid: 3
Sending response back for client: 5 for rid: 4 iid: 4
Sending response back for client: 4 for rid: 2 iid: 5
Sending response back for client: 4 for rid: 3 iid: 6
Sending response back for client: 4 for rid: 4 iid: 7
```

Figure 7: Client's view for base case

```
All BK Match, groovy
SERVERS
        PAXSERVER S01 Snd:   1840 Rcv:   3599
        PR Start:      0 Succ:      0
        BK Start:      0 Succ:      0
        Syncwrite:      0
        Max paxlog:      0
        PAXSERVER S02 Snd:   1802 Rcv:   3590
        PR Start:      0 Succ:      0
        BK Start:      0 Succ:      0
        Syncwrite:      0
        Max paxlog:      0
        PAXSERVER S03 Snd:  12470 Rcv:   8924
        PR Start:      0 Succ:      0
        BK Start:      0 Succ:      0
        Syncwrite:      0
        Max paxlog:      0
CLIENTS
        PAXCLIENT C04 Snd:      5 Rcv:      4 Start:     4 Succ:     4
        PAXCLIENT C05 Snd:      4 Rcv:      4 Start:     4 Succ:     4
```

Figure 8: Paxos internal execution statistics

```
 % ./pax -s 3 -c 3 -r 4  --sched "{0, 3, pause}, {100, 3, unpause}" -f -l DEBUG | grep re-sending
S01 Instance: 0 timed-out ( 1 times), re-sending accept, cid 5 rid 1
S02 Instance: 0 timed-out ( 1 times), re-sending accept, cid 6 rid 1
S01 Instance: 0 timed-out ( 2 times), re-sending accept, cid 5 rid 1
S02 Instance: 0 timed-out ( 2 times), re-sending accept, cid 6 rid 1
S01 Instance: 0 timed-out ( 3 times), re-sending accept, cid 5 rid 1
S02 Instance: 0 timed-out ( 3 times), re-sending accept, cid 6 rid 1
S01 Instance: 0 timed-out ( 4 times), re-sending accept, cid 5 rid 1
S02 Instance: 0 timed-out ( 4 times), re-sending accept, cid 6 rid 1
S01 Instance: 0 timed-out ( 5 times), re-sending accept, cid 5 rid 1
```

Figure 9: Part of the log showing conflicts being resolved by Leader