Paxos based directory updates for geo-replicated cloud storage

1.Introduction

With the recent trend of moving services to the cloud, it’s not surprising that with the services, data too has moved to the cloud. Cloud applications can be classified into various classes based on their purposes. Some, such as interactive web applications face stringent requirements on latency and availability. An example of such an application would be social media applications such as Facebook or Twitter. When we post updates/tweets, we expect the application to be more responsive than absolutely correct in its general ordering of news feed.

Applications like banking or online trading whilst being sensitive to latency and availability too, have a stricter requirement in consistency. Typically these are the online transactional applications. Outside of some special applications, we expect our transactions to be absolutely right even if it takes a few seconds longer to process.

The user bases of these applications are geographically distributed and the applications are expected to scale to hundreds of thousands of such users. In response to these challenges, a number of systems that replicate data across geographically distributed data-centers have emerged in recent years.

Geo-replication has the double benefit of:

1. Placing data closer to the user base to minimize access latency
2. Adding a redundancy later for availability in case of failures – alleviating the single point of failure problem

Geo-replication comes with the added cost of the replication process itself which could be asynchronous or synchronous (and hence on the critical path of read or write operations). The consistency (or staleness) requirements coupled with the latency requirements have a role to play in the choice of either. Thus we are left with the problem of customizing replication policies in cloud datastores to application workloads.

Some datastores like Cassandra and Dynamo are based on consistent hashing which limits their flexibility in placing data in replicas. Hashes however are fast to compute and the lookup time to determine the data location is just the compute time of the hash. Other datastores like COPS(?) assume that all data is replicated everywhere, which may be prohibitively expensive for large applications.

To motivate the need for flexibility in data placement, consider an interactive web application that involves reads and writes by geographically distributed users while requiring consistent updates on the replicated/distributed replicas. An example of such an application would be Facebook timelines or collaborative editing. This requires us to carefully choose the number of replicas maintained, which datacenters contain what data and the underlying consistency parameters (e.g., quorum size in quorum based systems). The requirement of placing data in the right datacenter is key to achieveing low read and write latencies in the cloud datastore.

This placement requires some sort of directory scheme to manage the current location of data. A directory scheme as the name suggests involves a collection of directories (typically one co-located with every replica) which serves as a lookup table for objects to determine their current locations. As the directory lookup now becomes a key step in object fetching, it is important to keep these directories updated. Thus, the directory is subject to the standard set of CRUD operations to reflect the current spread of data.

The importance of having consistent directory updates is to ensure correctness. If we have different directories holding different states for objects, one of them by definition is bound to be wrong. To illustrate this, consider the following example:



Object migration between replicas

In this example, we see that Object-1 has been migrated from one replica to another, but the updates made to the directories about the new location of the object are inconsistent. Dir-1 still believes that the object resides in Replica-1 and when the client asks the directory for the object’s location, it replies with the wrong replica. When the client tries to search for the object at this location, it will fail.

One way to address this problem would be to invalidate (make un-resolvable) the object’s location when under migration, and consider the migration complete only when all directories have been consistently updated about the object’s new location.

Our system entertains requests for migrations of objects (updates of object location state) and asynchronously manages this process. The actual object movement itself is considered blackboxed by the system.

* 1. Goals of the Thesis

As a part of this thesis, we implement a system which manages directory updates in a consistent fashion. Consistency is a widely studied topic and there exist many protocols such as Quorum and Paxos. There is contemporary research work regarding key and object placement strategies for consistent, high-availability, low latency applications (DTunes, SpanStore?). This thesis seeks to deal with the practical aspects of the proposed flexible key and object placement schemes by implementing the systems.

There have been a lot of papers explaining the workings of the Paxos protocol, and research works that propose optimizations and improvements to the same. However, we found that the number of actual implementations of the protocol were scarce. That also explains the lack of data and results about the performance of the Paxos protocol in a distributed WAN setting. There are large scale systems such as Spanner than use the protocol but there have been no official reports about the performance of the protocol. Our instrumentation of the consensus protocol is aimed at verifying common theories about convergence times of the Paxos protocol.

* 1. Thesis Contribution

We present our implementation of the consistent directory updates system built around a Paxos core. We have built a service around an open sourced Java based implementation of the Paxos protocol called JPaxos. Our system manages the coordination between the clients seeking to migrate or fetch object state, the Paxos cluster, agents which would manage the object movement and the directories consistently. The system exposes a CRUD-like API service for developers building services requiring consistent directory updates.

Alongside the implementation of this system, we instrument and deploy it in a simulated WAN setting (using the PRObE testbed) to observe the behavior of Paxos. The instrumentation results will be used to validate expectations of consensus system behavior we expect in different WAN settings.

* 1. Organization of the Thesis

The rest of the thesis is organized as follows: Chapter 2 presents an overview of the Paxos protocol and the JPaxos implementation including details about the salient features of the implementation and deviations and optimizations of the implementation from the standard protocol. Chapter 3 details the design and implementation of the directory updates system, challenges involved, decisions made and a description of the logging service. Chapter 4 presents our evaluation goals and methodology, a brief description about the PRObE public testbed used to run the experiments and then details the results along with discussions about the same. Chapter 5 concludes the thesis.

2. Background

2.1 Paxos

Paxos is an algorithm for implementing fault-tolerant distributed systems. At the heart of the Paxos algorithm/protocol is a consensus algorithm – how do we get multiple processes that are each trying to assert/propose a value to agree upon and stick with a single value?

The safety requirements of such a consensus algorithm required to achieve consistency are as follows:

* Only a single value that has been proposed may be chosen
* Processes learn about values if and only if they have been chosen

Note here that the safety requirements do not specify any liveness/convergence requirements. That is, all we’re focusing on here is correctness, not practical concerns such as progress. There are 3 classes of “agents” that take part in the protocol:

* Proposers – They propose values to be chosen
* Acceptors – They choose to or not to accept proposed values
* Learners – They learn the final, single proposed value that was accepted by the acceptors (not all, just a majority, see below)
* There are no strict requirements on the mappings between the given processes and these roles

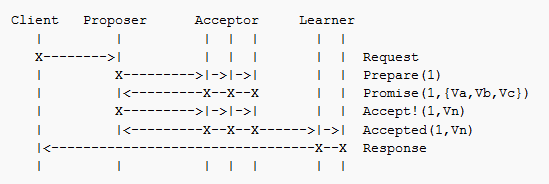
A value proposed by a proposer can be considered accepted once a majority of acceptors have accepted it. The cornerstone of the algorithm lies in determining how and which value must be accepted. From a bird’s eye perspective, the acceptors control the proposers and their proposed values – so the working of the algorithm is driven by acceptors forcing the proposers to propose acceptable values, whilst the design of the algorithm revolves around setting down rules for how to accept values. The design considerations for accepting values are as follows (revised as new requirements emerge):

* 1. An acceptor must accept the first proposal it receives – we must begin somewhere
* Only a single value must be accepted => we’ll turn this around and instead put the responsibility on the proposers and say – only that value may be proposed repeatedly
* As a proposer, I can see what values have been accepted while proposing, but I cannot predict what values might be accepted in the future. To this end, I somehow seek to control the future acceptances by extracting promises from acceptors regarding the nature of the same
  + Proposals now have a proposal number. To avoid confusions, different proposals must have different numbers, a global ordering of sort – the implementation left open ended. A suggestion would be to just have proposers choose the numbers from non-overlapping sequences and store the last used number in stable storage.
  + Promise to me, the proposer that you, the acceptor will not accept a proposal with a number lower than mine
  + If you have already accepted a proposal, let me know.
* Due to this extracted promise, we need to change acceptance rule 1 to: 1a. Acceptors can and must only accept proposals that do not violate promises it has made => accept proposals which have numbers > numbers of proposals to which promises have been made
* 2. If a proposal with value ‘v’ is chosen, then every higher numbered proposal that is chosen by any acceptor has value ‘v’ – this follows from the requirement that only a single value be chosen in a round of Paxos.
* This is where the implementation of the algorithm is driven backwards – to ensure that no proposal with a value other than ‘v’ with a proposal number higher than the highest accepted proposal number (with value ‘v’) is accepted, the acceptors force the proposer to only issue proposals with value ‘v’. Hence:   
  2a. If a proposal with value ‘v’ is chosen, then every higher-numbered proposal issued by any proposer has value ‘v’.
* Now we relax constraint 2a by moving to a majority instead of every acceptor. Hence:   
  2b. For any proposal numbered ‘n’ with value ‘v’ issued, there exists a set ‘S’ consisting of a majority of acceptors such that either:
  + a) no acceptor in S has accepted any proposal numbered less than ‘n’, or
  + b) ‘v’ is the value of the highest numbered proposal among all proposals numbered less than ‘n’ accepted by acceptors in ‘S’

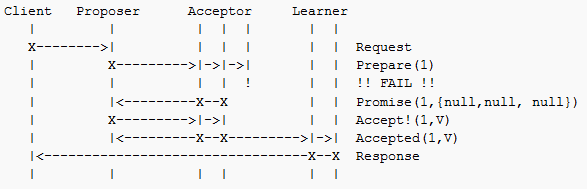
Putting all this together, the algorithm for a single ‘round’ of Paxos sums up to such:

Phase 1.   
(a) A proposer selects a globally exclusive proposal number ‘n’ and sends a prepare request to a majority of acceptors (it could be all acceptors in the implementation) – this is called a ‘prepare’ request.  
(b) If an acceptor receives a ‘prepare’ request with number ‘n’ greater than any ‘prepare’ request to which it has already responded, it responds to the request with a promise not to accept any more proposals with number less than ‘n’, and the number ‘n’ and value ‘v’ of the highest number proposal it has accepted (if any).

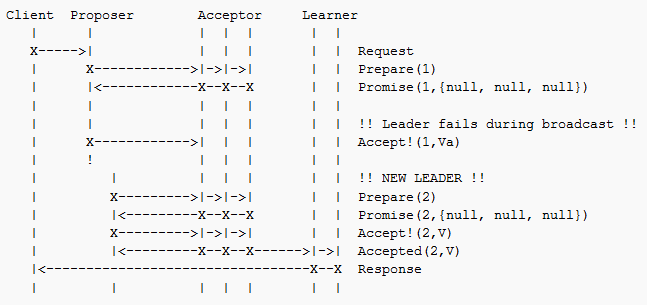
Phase 2.  
(a) If the proposer receives a response to its prepare request numbered ‘n’ from a majority of acceptors, then it sends an ‘accept’ request to each of those acceptors for a proposal numbered ‘n’ with either the value of the highest numbered proposal it received from the acceptors in response to its prepare request, or if no such value exists, then any value of its choosing.  
(b) If an acceptor receives an accept request for a proposal numbered ‘n’ >= highest prepare request number it has responded to, then it accepts the proposal.



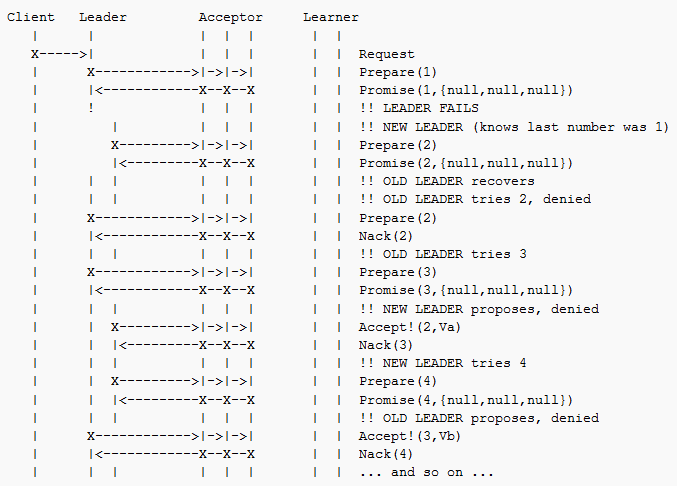
Paxos: round successful



Paxos: A single failed acceptor, majority still ensures progress.



Paxos: Failed proposer



Paxos: Livelock without distinguished proposer

There are a few things here to note:

There is no direct correlation between Phases 1 and 2 in terms of a Phase 1 being sufficient for Phase 2. That is, a proposer ‘P1’ could elicit a response to its prepare request but it might end up racing with another proposer ‘P2’ in that acceptors could end up rejecting P1’s accept requests after accepting its prepare requests because P2 is racing P1 and keeps issuing prepare requests with numbers succeeding P1’s prepare requests.

A decision is implicitly reached when a majority of the acceptors accept the same value ‘v’ – because using induction and the property of there being at least one common acceptor in the intersection of 2 majorities of acceptors, we can show that the acceptors will force any future proposers into re-proposing the same accepted value.

There is no limit on the number of proposals that can be made – proposers can abandon proposals mid-flight and reissue proposals of higher numbers if they want.

There is no guarantee of convergence – the protocol is correct, but may never converge.

To learn a chosen value, the learners must find out that a majority set of acceptors have accepted a single value. There are multiple ways to do this, the most straightforward of which would be to have every acceptor ack acceptances it makes to every learner.

A few optimizations to the protocol:

* The first obvious optimization would be some step to alleviate the non-convergence problem. We could have a “distinguished proposer”, a leader who would be the only one trying to issue proposals, circumventing the race problem.
* Similarly, we could have a distinguished learner, or a set of them to reduce the number of acks that the acceptors would have to send out once they accept a value.

2.2 JPaxos

2.2.1 Brief Introduction

JPaxos is a full-fledged, high-performance Java implementation of state machine replication based on Paxos. It is an open-source contribution maintained at github.com/JPaxos. The focus of the implementation is on:

* Batching and parallelizing rounds of Paxos to maximize performance
* Different crash recovery mechanisms for systems with and without stable storage
* Scalability with processing parallelism

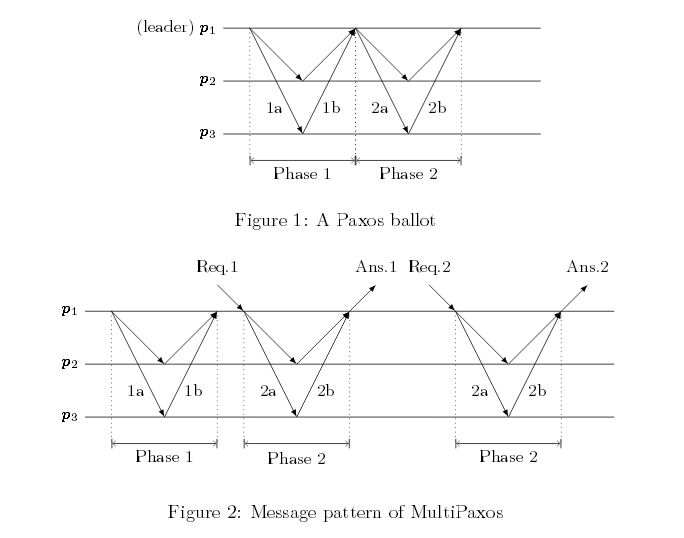
To deal with replicas/processes who are members of the Paxos protocol crashing, JPaxos uses 2 crash models:

* Crash stop: If a member of the protocol crashes, it cannot recover from the crash.
* Crash recovery: A member can resume execution of the algorithm after crashing. There are multiple implementations of this model to trade off critical path execution time with state storage in stable vs. volatile storage. This consequently leads to recovery from different periods before the crash.

2.2.2 MultiPaxos

Now we present a brief discussion about MultiPaxos state machine replication.

The Paxos protocol by itself only describes the necessary conditions to achieve consensus for a single round of execution. To extend this to a sequence of rounds of consensus in deterministic order is closely related to the atomic broadcast problem, as both share the same core problem of ordering a sequence of values.



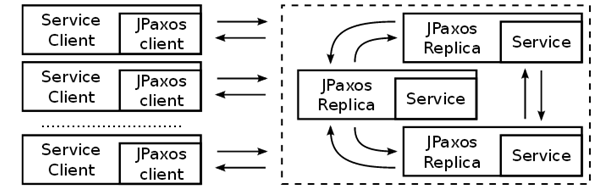
The message patterns of single (optimized) and multi Paxos.

Though it would possible to use a sequence of independent Paxos instances with absolute ordering to implement MultiPaxos, it would be inefficient. MultiPaxos achieves better performance by “merging” execution phases of several instances. In MultiPaxos the system advances through a series of views, which play a similar role as ballots in single instance Paxos.

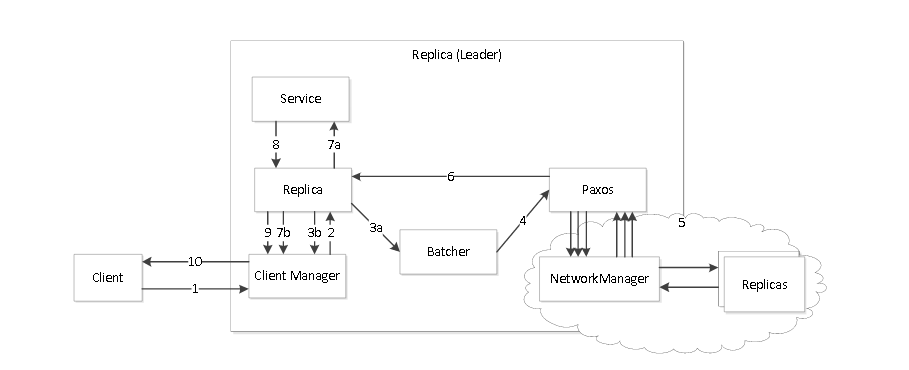
JPaxos uses a concept of a view. A view plays a similar role as a ballot of a single instance of Paxos. The leader of each view is determined by a rotating coordinator scheme, that is, the leader of view ‘v’ is the process v mod n where n is the number of replicas in the cluster. Once a process ‘p’ is elected leader (by some external leader oracle module), it advances to the next view number ‘v’ such that p is the coordinator for that view (v mod n = p), and ‘v’ is higher than any view previously observed by p. Process p then executes Phase 1 for all instances that according to the local knowledge of p were not yet decided by sending a <Prepare, v, i> message where ‘i’ is the number of the first instance that p thinks is undecided. The acceptors answer with a message containing the Phase 1b message for every instance of consensus higher than i. The acceptors send the last value they accepted (or null if they accepted no value). Once Phase 1 is complete, the proposer can then execute Phase 2 for every instance >= i. In state machine replication, when the leader receives new requests from clients, it executes only Phase 2 of a new instance.

2.2.3 JPaxos architecture

JPaxos consists of two main modules: Replica and Client. The Replica module executes the service as a replicated state machine, while the Client module is a library that is used by client applications to access the service.



A service replicated in three replicas accessed by several clients. Arrows indicate communication flows.



Request handling, when client is connected directly to the leader. 1) Client sends request, 2) request read and forwarded to Replica modules, 3a) request added to batch queue (new request) or 3b) send cached answer (repeated request), 4) propose request as part of a batch, 5) order batch using MultiPaxos, 6) after being ordered, batch is given to Replica for execution, 7a) Replica executes request in service if the request is new, or 7b) answers with cached reply if request is repeated, 8), 9) and 10) answer is sent back to the client.

A Paxos log is a data structure that is actually replicated consistently among replicas. Each replica has its own copy of the log. The log reflects a series of rounds of Paxos with relevant information for each such as whether the round has been decided yet, what value was decided upon, when it was decided and the replicas that formed the majority for the decision. One of the details of MultiPaxos which is open to implementation that we will discuss here is the generation of unique request identifiers. Each request must have a unique identifier in order to be distinguished from the others. A commonly used method is by using a <clientID, sequenceNumber> pair. While sequences can be monotonically maintained within a system, the main challenge arises with allocating unique clientIDs. JPaxos makes replicas responsible for granting IDs to clients. When the client first establishes a connection to the replica, a unique ID will be supplied using 2 policies:

* # of replicas: A modulo scheme based on number of replicas. Replica i of 0…n will grant “k mod n = i” integers as IDs.
* Time based: IDs granted are of the form (t+localId), (t+localId+n), (t+localId+2n) where t is the time the Replica was started, localId the identifier of the replica and n the number of replicas in the protocol.

MultiPaxos requires both a leader election oracle and a mechanism to assign to each process an infinite number of exclusive proposal numbers, both of which are left as implementation details. JPaxos uses view numbers to implement both leader election and to generate proposal numbers. The ordering protocol is organized as a sequence of views with increasing numbers. Processes keep track of their current view v (which would ideally be the same across processes unless one of them is recovering from a crash) and tag all their messages with the number of the view when they were sent. Messages from lower views are ignored and receiving a messages from a higher view forces the process to advance to the higher view immediately. As discussed before, the process v mod n is pre-assigned to be the leader of view v. Proposal numbers are generated by using the view number and adding a sequence number internal to that view. The leader of view v uses <v,i> as a proposal number where I is the sequence number generated by the leader. Ordering among proposals is defined first by view number then, in the case of a tie, by the sequence number. Leader election is implemented by advancing view whenever the leader of the current view is suspected to have failed. When a process suspects the leader, it tries to become leader by advancing to the next view where it can be leader and sends out Prepare messages to everyone. If multiple processes suspect a leader crash and race to become leader, the one with the highest view number will win. Failure detection happens using a simple heartbeat based scheme – the leader sends out Alive messages as a heartbeat to inform other replicas that it’s still up. A configurable number of missed Alive messages in a configurable time interval will result in the process suspecting the leader of having crashed.

2.2.4 Optimizations

JPaxos uses several optimizations to reduce the number of messages sent. Many of these are possible because in JPaxos every process is at once Propose, Acceptor and Learner – so all processes share the exact same replicated log (outside of crashes).

Sending to self: In traditional Paxos, the Proposer has to send the Phase 1a and Phase 2a messages to all Accptors. As the leader plays both roles, it can suppress the message to itself, and directly updates its state. A similar optimization can be applied when the Acceptor sends 2b to all Learners.

Merging Phase 2a and 2b of the leader: In Phase 2, the leader has to send, as the Proposer a Phase 2a message to all, and immediately after, as an Acceptor, a Phase 2b message also to all. Since the leader implicitly accepts its own 2a message, it only sends out a 2a message to all other processes, which is understood to be a combined 2a + 2b message. This reduces the number of messages sent by leader in Phase 2 by 50%.

Minimizing count of messages carrying the value: Most descriptions of Paxos state that both the Propose and Accept messages carry the value being agreed upon, which in our case are client requests. The size of the requests is contingent on the service being implemented and could get large. JPaxos ensures that the message is only sent once per round by omitting it from the Accept messages and relying on the Propose message of the leader to distribute the value to all replicas (which then store it in their local storage). To preserve correctness, the protocol of the Acceptor must be modified slightly: if the Acceptor receives an Accept before the corresponding Propose, it must wait before sending its own Accept.

2.2.5 Catchup

MultiPaxos must guarantee that all learners eventually learn the decision of every instance. This is important because a gap in the sequence of requests to be executed will block the process from executing future requests because of the properties of a state machine replicated log.

The leader keeps retransmitting the Propose message until it receives a majority of Accepts, but in a lossy network this does not guarantee that all processes will receive enough messages to decide.

For these situations, JPaxos includes a catch-up mechanism based on the following observation: if the leader is correct (heartbeats) and a process knows of an instance that started some time ago but has not yet been decided, then it’s likely that the value has already been decided and the process should contact some other process to learn the decision. There are 2 possible catch-up schemes that could be used:

* Log-based: The replica only copies the missing decisions from other replicas and then executes them locally.
* State-based: The replica copies the missing decisions and the resulting state from other replicas and just directly applies them as a patch without moving through the state machine.

The more intricate details of triggers to start and stop catchup, and the different catchup and recovery algorithms are not discussed here.

2.2.6 Recovery

Recovery is the process of a crashed replica coming back up and rejoining the Paxos cluster, taking part in the decision process of new requests after making up for any requests that it might have missed while it was down. When reasoning about crash-recovery, it is usual to assume that processes have access to volatile and stable storage. Any data stored in volatile memory is lost during a crash, while data on stable storage is preserved.

There are 3 recovery models that JPaxos uses to support different types of recovery (excluding the trivial CrashStop recovery model wherein crashed replicas simply cannot recover). They differ in how they log checkpoints to recover upon a crash.

Find below a brief description of each recovery process:

Crash Recovery with stable storage:

This algorithm saves enough information to stable storage so that upon recovery, it can restore its state using only information stored on local storage, and rejoin the protocol without executing any additional recovery protocol involving the other replicas. With this algorithm, processes write to stable storage often, once per instance of the ordering protocol which is a significant overhead.

Effectively what this model does is save every executed instance and the respective decided value in stable storage upon decision. Thus when a replica recovers, it can replay all requests necessary to rejoin the protocol just using the locally saved information.

Using this model enables the cluster to tolerate catastrophic failure (that is, possibly all replicas fail), but as noted comes with the additional overhead of making synchronous writes to stable storage on every decision.

Epoch-based recovery:

This algorithm only makes one synchronous write to stable storage on process startup, but the recovery phase is more complicated than the previous algorithm.

The algorithm is not described in detail here, but a high level view would be that it uses an epoch vector describing the last epoch number written to stable storage, where the epoch number is updated on every recovery after a crash. The algorithm uses this epoch vector alongside replies of epoch vectors from other replicas, and the catch-up module of JPaxos to restore lost state to the crashed replica.

This algorithm however requires a majority of replicas to be up at any given point to work, but is almost as fast as the trivial CrashStop model which has no synchronous writes to stable storage.

View-based Recovery:

Like Epoch-based recovery, View-based recovery requires a majority of processes to be up at all times, just that instead of using epoch numbers, the view number is written to stable storage on every change (while the epoch number was a number that was incremented on every recovery from a crash).

Similar to Epoch-based recovery, View-based recovery uses the collective knowledge of instances from other live replicas to restore state it has missed.

The performance of this algorithm too is almost as fast as the trivial CrashStop model, but is contingent on the number of view changes (synonymous with leader changes), as more view changes trigger more synchronous writes to stable storage.

2.3 PostgreSQL

The database used for saving all migration requests and tracking changes/updates to them is PostgreSQL. This is also the database used to log the instrumentation data from the logging framework.

PostgeSQL is a powerful, open source object-relational database system. It has been in active development for 15 years and runs on most major operating systems.

Apart from being having a very rich set of features, Postgres is a database that is very commonly used in the industry today as a relational store. It’s high performance, customizability, and most importantly open-source nature make it a top choice for this purpose.

It is for this reason that we chose to use Postgres as the database of our choice for tracking migration data. It’s very rich and performant query optimizer and interface also made it an ideal choice for logging instrumentation data to run analytics on the same.

3. Design and Implementation

3.1 Motivation

At the outset, we talked about flexible data placement schemes for interactive, latency sensitive web applications. We now briefly further motivate the problem by providing context about one of the direct applications of the implemented system – key migrations. To elucidate further about this and give an example of the necessity to adjust data locations, consider the read and write latency sensitivity dimension of the design space.

Consider a twitter user in India with majority of his/her friends and followers in India. It would make the most sense to save his tweet data in data centers in Asia close to India. Now consider the scenario of said person moving to the U.S. and making a lot of new friends in the U.S. who are interested in following him/her on Twitter. The placement of data in the Asia data center would no longer be optimal because his friends in the U.S. would see much longer latencies in retrieving his tweet data as compared to his friends in India. Also, all his new Tweets (writes) would now have to be pushed all the way to the Asian data center which also is sub-optimal. This problem can easily be made more involved by bringing in weights as a design consideration:

* Read latency weight vs. Write latency weight (prioritizing one over the other)
* # reads or writes from a given geographical region
* # of users in given geographical region
* Storage cost of the data in replicated geographical regions

Finally, we must realize that this computation will not result in a static placement of data. Once these parameters change by a large enough factor, they will necessitate a migration of data to a placement optimal to the new parameters. This makes a solid use-case for the system we’re designing. We have talked about consistent updates and the need for them, but we haven’t discussed the necessity for Paxos to implement the same. The use of Paxos alleviates the problem of single point of failure. To illustrate this, consider the following scenario:

Consider a single node migration “cluster”. That is, one node that manages the whole migration process. Clients connect to this single node to initiate migrations and that node manages the whole migration process and eventually drives migrations to completions.

What happens if this node fails? Not only can we not entertain any new migration, but all the migrations that the node had underway are also stalled at this point. So we have a single point of failure.

What if we set up a master-slave replication scheme instead? This could be implemented one of 2 ways:

* Synchronous updates – this would slow down writes because every write would now incur n (number of slaves) write costs + interconnecting link transmission costs to keep all replicas up to date.
* Asynchronous updates – this would cause correctness issues as a failed master would result in a slave that could possibly be out of date (if not all updates have been propagated). This would cause the slave to make wrong decisions regarding the migration process

3.2 The Migration Service

The migration service is the CRUD wrapper around the Directory Service. It guarantees that once a migration request has been persisted (Paxos-ly), the system will eventually drive the migration to completion. We separate the migration process into four main concerns which we shall discuss in detail:

* A directory service that implements the Paxos service interface fronted by the JPaxos code
* A migration protocol that takes care of the actual migration process
* Directories that store the current location of a given object
* Migration Agents that perform the actual process of copying the object from the source to the destination location.

3.2.1 The Directory Service

The Directory Service implements the service wrapper that JPaxos fronts. This means, apart from the actual execution of decided Paxos requests themselves, it also handles concerns like snapshotting and restoration from snapshots. The design considers that there is no single replica that can be in charge of the migration of even just a single object. The justification for this is that we can have failures and leader changes mid-migration too.

When a new object migration is received (and decided upon), a record is created in the database to represent the migration request. This record represents all the state information associated with the respective object’s migration. The fields stored in the record are:

* ObjectId: The ID of the object being migrated
* Old Replica Set: Where the object is currently being stored
* New Replica Set: Where the object will be stored after the migration
* Directory Acks: Directories that know about the new location of the object
* Migration Progress Acks: Migration agents that have ACK’d back, having completed their share of the object movement
* Creation Time: Time of creation of the record
* Completion Time: Time the migration was completed
* Last Updated Time: Time this last record was last serviced (read/updated). This is meant to be a mechanic to avoid starvation.
* Migration Started Timestamp: Time the Migration Agents were informed about the required object movement.
* Migrated: Boolean state of whether the object has been moved from its old replica set to the new one.
* Migration Complete: Boolean state of whether the whole migration process has been completed for this object.

These fields are updated as the object moves through the migration process, and are also used to make decisions about what must be done next to complete the migration for the object.

Since we are implementing a fault-tolerant service, we can see that all write operations (insert and update) made on the record must be Paxos operations. The justification for this is simple – if any operation is performed non-Paxosly, it would lead to some state being saved local to a replica or a subset of replicas not honoring the Paxos safety concerns. This would lead to that single write/update being inconsistent and correctness being an all or nothing concept would break down.

The first thing we had to consider while implementing the service was that the way the Paxos protocol as implemented recovers from a failure is by replaying batches of requests. Depending on the crash recovery model we use, the strategy of recovery changes. For our implementation we only used the EpochSS recovery model. The EpochSS recovery model replays all requests from the last saved Epoch, not just saved requests. This means that decided requests that the crashed replica played before crashing will now be replayed. The model has been designed to replicate services that have no state surviving crash. However, JPaxos gives us a sequence number for every request that has the following properties:

* It is monotonically increasing
* It is consistent amongst replicas (the same request will have the same sequence number for every replica)
* There are no gaps between 2 numbers

Using this primitive, we can design a stateful replication service. To do this ,we save the sequence number of the last executed request in the database and when we replay requests, we skip all requests up till the last executed one. One important thing to note here is that the saving of the last executed request’s sequence number and the effect of the last executed request itself (in terms of any database operations) must be one single atomic operation.

Now that we have a stateful replicated system, the next thing to discuss would be the operations that the service supports:

* Insert: creates a migration record in the database. By design, only one outstanding migration is allowed for an object. This design choice has tight correlation with the design choice of optimizing local database reads as opposed to Paxos reads in the migration protocol.
* Update operations: updates on the above listed fields of the migration record. In the directory service, updates are triggered by either the protocol process or a migration agent.
* Read: reads and serves the current migration state for the given object id
* Delete: deletes the migration record for the given object from the database
* Register operations: register directories/migration agents that are bootstrapping

To implement snapshotting, instead of restoring from a snapshot by replaying all requests contained in the snapshotted period, we snapshot the state of the database itself. This makes the implementation straightforward and in some cases we end up with fewer database operations this way. An example of such a case would be a snapshot of a single completed migration’s database record – if we maintained a traditional transaction based log, we would have about 8 database write transactions to reach completion. If on the other hand we just snapshotted the finished database state, we would achieve the same effect, in a single insert statement. When restoring from a snapshot, the restoring replica wipes its database state clean and completely restores the state received.

3.2.2 The Migration Protocol

The protocol is a process that runs co-hosted with all the replicas. The only protocol process that can make any decisions and take actions is the one co-hosted with the leader process. Reason for this being, the protocol process co-hosted with the leader makes DB local reads (as an optimization instead of making Paxos reads). When a leader change occurs, the process co-hosted with the new leader seamlessly picks up from where the old one left off. To do this, we have a polling mechanism set up between the replica and protocol processes. Since we want the protocol processes to be able to function in this seamless fashion without any communication between them on failure, the main design consideration was statelessness.



A step-by-step progress of the migration process

The protocol process is essentially a state machine. It picks up a few objects which still have been slated for migration and are in different stages of their migration and pushes them to completion. In the implementation, the process is not threaded – that is it only performs one migration at a time.

As discussed before, the progress of any object’s migration is reflected in its state in the database. The protocol process reads in the full state of the object from the database and then through a series of conditionals that represent the state machine, determines where along the process the object currently is and what the next step should be. We now present a detailed step-by-step example to illustrate the statelessness, seamlessness and working of the protocol processes:

* Consider an object “object-id:1” that currently resides in some set of replicas {A,B,C} and needs to migrated to {B,F}.
* A client process (that we refer to as the migration initiator) connects to the Paxos cluster and requests that this migration be performed.
* The leader process accepts the request and queues it for proposing.
* At this point, a failure of the leader process would not preserve the request and it would be lost. But also note, we have not responded to the client yet confirming that we have registered the migration request and it is now fault-proof.
* The leader eventually (based on parameters such as the number of outstanding proposals, poll time, batching factor) drains the client request for the migration and proposes it.
* Once the proposal is accepted by a majority of other replicas (including the leader), the leader replica proceeds to “Decide” the request.
* Now note that each replica (by implementation in JPaxos) “Decide”s requests independently of each other. That is “Accept” requests from the follower replicas are multicast to all other replicas in the protocol, and once any replica in the cluster locally sees a majority, it goes ahead and “Decide”s the request.
* Once a request has been decided, the request is then processed – that is, each individual replica gives control to the underlying implemented service to process the request byte stream.
* This is where the previously discussed Directory Service would create a migration record for the Object in the database.
* At this point, the migration request is considered resilient as it has been replicated in a majority of replicas.
* Now the leader replica responds back to the client saying that the request has been persisted in the database and will eventually completed.
* Meanwhile, the protocol process co-hosted with the leader is constantly polling the database for any outstanding migrations that need to performed.
* It finds the new migration request in the database and starts off by initiating the actual movement of the object.
* It does this by informing all registered migration agents of the object move by opening up connections to the IP/port they are listening on. It only does this for processes that have not already completed their part of the move and ACK’d back that they are done.
* The ACKing from the migration agents is an asynchronous operation. The protocol process does not block on its execution path for this move to happen. It simply informs the agents about the required move and updates a timestamp on the migration record to reflect when it last informed the migration agents about the requisite move.
* The timestamp helps handle failures of migration agents. If the protocol process informs them about the move and they fail before they can complete it, the process retries if it sees that all agents haven’t ACKed after a specified time interval from when it last updated them.
* When migration agents ACK, it is another Paxos operation (as it is an update operation on the migration record’s state)
* Once the protocol process sees that all registered agents have ACKed for the requested object move, it proceeds to the next step which is informing the registered directories about the new location of the object.
* To do this, it runs over each registered directory and communicates the new replica set for the object over a connection to the IP/port the directory has been registered with. Once the directory (synchronously) ACKs back that it has received the information, the process makes a Paxos update to reflect the updated states of directories that have been informed about the new state of the object.
* The reason for doing these updates on a per-directory basis is to keep our protocol stateless – this way, even if one protocol process fails midway, another can take over and continue where the old one left off.
* We assume that the directories store the information in stable storage, so the information they hold is resilient to failures.
* Once all directories have been informed, the protocol process then calls the migration complete and proceeds to look for more migrations to perform.
* We note that all operations performed by the protocol process are just enactments of the state machine logic. It does not locally store any state about the progress of the migration. Any progress made is replicated and stored as state in the database, thus keeping everyone updated about the progress of every migration and achieving the seamless transitions between protocol processes in failure scenarios.

TODO: Maybe a detailed state machine diagram of the protocol process?

3.2.3 Directories

Directories are processes that maintain a lookup table of mappings between objects and the set of replicas that currently hold them. When directories bootstrap, they must register with the Paxos cluster using the IP and port they are listening on. The protocol processes and any other interested clients lookup directory listings in the database to contact them.

3.2.4 Migration Agents

Migration Agents are processes that blackbox the migration process. In the implementation, they wait for a fixed time interval before ACKing back to the Paxos cluster to signify the completion of object move. This process as discussed is asynchronous – so their ACKs include details about the object which was moved. Similar to directories, when migration agents are bootstrapped, they too must register with the Paxos cluster using the IP and port they are listening on and the listing is looked up by the protocol processes to contact them.

3.2.5 Logging Framework

The above sections discussed the implemented Directory Service for fault-tolerant object migrations. Another one of our main aims was detailed instrumentation of the Paxos protocol and to observe its performance in a WAN setting. To achieve this we needed a non-invasive, detailed logging framework.

* If we logged the progress of every request as it passed through checkpoints we setup in the system, we could collate the results to understand how the time is being divided on a part-by-part per request basis. The salient required features: Per request granularity of logging: We need to able to identify individual requests so any discrepancies can be tied back to the originating request for analysis.
* Non-Invasive: The logging framework cannot interfere with the actual execution of the code itself. It must be as decoupled from the code as possible. This translates to an asynchronous, threaded setup with queues.
* Aggregated: We are tracking the progress of the request as it proceeds through multiple processes/machines in a distributed systems setting. We have the ability to track a single request across systems using its request number. We use the same to aggregate log data from multiple checkpoints across multiple machines into the log of the progress of the same request. We still maintain the source of the data – that is 2 checkpoints for the same request that are hit on 2 different machines will not overwrite each other; instead they will result in 2 numbers for the same checkpoint, for the same request under different replica numbers.
* Analyzable: Instead of complicating the post-processing of logs to derive metrics, we do the work up front before persisting the logs. We use a relational database to store our data. This gives us a rich log format to store data under. This also gives us a very rich query interface to run analytics on the logged data.
* Detailed: We wanted to log the data in as much detail as possible as we could always choose to process it at a coarser granularity. To this end, we have about 20 logging checkpoints setup per machine. However, some of these checkpoints will not be hit on all machines as they are parts of the code only the leader for the round will execute.

4. Experimental Methodology and Results

4.1 Aim

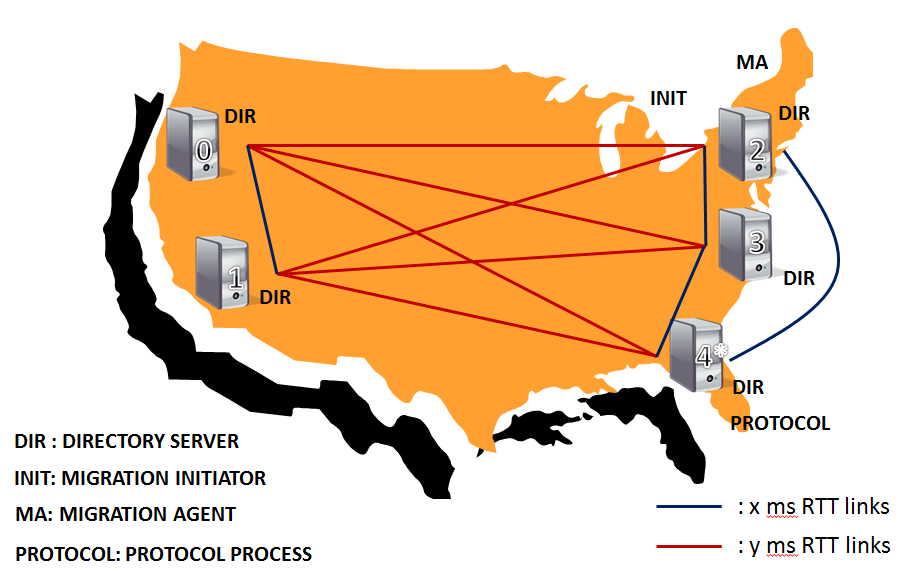
The high level aim of the experiments was to get a deeper understanding of the Paxos algorithm in an actual implementation. This effort can be broadly classified into 2 steps:

* Understanding the implementation and it’s departures from the algorithm in terms of optimizations and details generally left open to implementation.
* Instrumenting the implementation to piece-wise analyze the different component times in the algorithm.

Coupling this with different delays simulated between replicas would give us an example of an instrumented Paxos system deployed in a WAN. The Directory Protocol gives us a working state machine system built around the Paxos core to drive the experiments.

4.2 Setup

The experimental setup is as shown below:



Experimental Setup

We used a five replica cluster. The cluster was split into 2 sub-clusters. Depending on the inter and intra cluster delays this allowed us to model different geographical setups. Each replica runs its own directory server. The Migration Initiator is co-located with the Migration Agent on one of the replicas. We did not want to end up needing a lot of the machines to conduct these experiments. Also since all 3 processes are rather lightweight, there would be limited overhead/contention between them. The Protocol Process runs on every machine, but since for the purpose of these experiments we are not simulating any failures, all processes apart from the one co-located with the leader replica are of no consequence as they would be unable to take any action. Since the focus here is on latency and not throughput measurements, we set the parameters to have high polling frequencies, and low critical path latencies.

* CrashModel – EpochSS
* WindowSize – 2 (the migrations are serial for the purpose of these experiments)
* MaxBatchDelay – 0 (do not batch, push proposals instantly on arrival)

The implementation had been modified to force any Paxos client to always connect to the leader process (directly, no redirections) and the same replica (#4) is elected leader for all experiments. This forces uniformity between runs enabling us to set strong expectations for the experiments.

4.3 PRObE

The experiments were run on the NMC’s (New Mexico Consortium) PRObE test bed.

PRObE’s Kodiak cluster is an excellent, well-provisioned large scale compute resource that runs on an Inifiniband backend. At the time these experiments were being conducted on PRObE, they had upwards of 700 free nodes at any given point.

To create an experiment, a NS (network simulator) formatted file describing the topology for the experiment is expected to be supplied. This allows you to setup whole network topologies including the nodes, the bandwidth and latency of the interconnects. However, for the purposes of these experiments, a very basic NS file that just spawned the required number nodes on the default Infiniband network was used.

PRObE also gives its users the ability to snapshot images of their node’s disks. This proves to be extremely useful when it comes to bootstrapping nodes with a custom OS and software packages necessary. This enables having a cluster of nodes up and running from scratch in a very short time period. As PRObE has a limit on the maximum duration an experiment can stay active, it necessitates the need to snapshot disk images. Once the nodes were swapped in (become active), since we have an extremely fast network fabric for a primitive, we use DummyNet to shape the link delays to emulate network configurations that we want.

The Kodiak cluster’s nodes are not accessible from the internet. Management of the nodes (including access) happens through an ops node. Consequently data is also shuttled in and out of the nodes through the ops node. The ops node also controls other functions of nodes such as rebooting and reloading a new disk image amongst other things.

4.4 Experiments

The experiments are each run for 10 key migrations – each key migration involves 5 Paxos stages, namely:

* Migration Initiation
* Directory Acks – 1 per agent (5 in our implementation)
* Updating Timestamp after informing Migration Agent about object move
* Migration Agent Acks – 1 per agent (1 in our implementation)
* Updating migration record to complete

All graphs are plotted on the basis of above mentioned Paxos rounds. We ran our experiments for 3 different network configurations. Discussed below are each configuration and associated expectations and results.

4.4.1 No DummyNet

These set of experiments are running on the PRObE testbed’s Infiniband fabric directly. The link delays between nodes i.e. x = y = ~0.1ms.

Expectations:

* The Paxos latencies as measured from the enqueued phase to decide phase measured at the leader (as that is the only replica which has to power to queue and propose) should be of the order of a few milliseconds (~1 RTT). This is heavily dominated mainly by 2 factors:
  + Code execution time and threading overhead.
  + Queuing latencies between the multiple asynchronous parts of the application
* The client end to end latencies should be dominated by the service time of the request. The link latencies are almost negligible and the Paxos part of the request servicing should only come out to a few milliseconds. The service time again can be splits into 2 dominant factors:
  + Code execution time of the state machine itself
  + Database access times

Results:

  
Paxos client end to end latency



Paxos leader latency



Directory service time

While we have some deviation from the expected case, if we round-wise sum the service times and Paxos leader latencies, they line up with the Client end to end latencies.

The client end to end latencies may not be uniform because of the different natures of the Paxos clients:

* Migration Initiator and Migration Agent processes run on different machines from the leader process, and will hence experience link delays.
* The Paxos client for the other rounds is the Protocol process which is running co-hosted with the leader replica, and will only experience inter-process communication delay.

The possible mismatches could be due to reasons such as the client end to end latency including a round trip from the paxos client to the leader replica and whilst on inifniband this link latency itself is a small value, there is a finite queuing time involved in the transmission of the message within the application.

4.4.2 DummyNet with 0ms delay

DummyNet introduces a finite amount of overhead. If we configure DummyNet to emulate link delays to be 0ms, there is still a finite ping time between nodes. This time was roughly observed to be around 5ms. Plotting this case will now give us a benchmark for an overhead over our pure inifniband case.

The expectations here are the same as the no dummynet case and any observed deviation will be treated as the dummynet overhead.

Results:



Client end to end latency  


Paxos leader latency



Directory service time

Now we notice that we have a large deviation from the expected case (which would be that these graphs be very similar to the no dummynet graphs). Since there has been no change between these 2 experiments apart from dummynet being introduced to model 0ms delay, all of the deviation can be attributed to dummynet overhead.

As the Paxos leader latency times represent ~ 1 RTT of link delay, we can use the overhead in that graph to fit and verify our earlier proposed explanation to the mismatch between the round-wise sum of the Directory service times + Paxos leader latencies and the Client end to end latency:

* The client end to end latency includes a round trip from the paxos client to the leader replica.
* The earlier explanation attributed the extra time to message queuing delays in the application, but with this new overhead we are seeing due to dummynet for the Paxos leader times (~1 RTT) would now make the round trip from the client to the leader replica significant and in fact the dominant factor.
* Accounting for this, our observations line up as expected – Paxos leader latencies + Directory service times + Paxos leader latencies representing client-leader roundtrip ~= Client end to end latencies.

4.4.3 DummyNet with uniform 20ms (RTT) latency

Now we simulate the 5 nodes being at equal link delays of 20ms from each other. i.e. x = y =20ms.

Expectations:

* The convergence time of a single Paxos round measured at the leader would now be composed of 3 factors:
  + Code execution time and threading overhead.
  + Queuing latencies between the multiple asynchronous parts of the application
  + Since all replicas are 20ms (RTT) away from the leader, ~1 RTT convergence time for the Paxos round would also now be expected to kick in.
* Since in our baseline pure Infiniband only experiment we saw Paxos leader times of ~3-5ms which account for i. and ii., overall we would expect to see a 20 + 3-5ms delay.
* But we also note that in our 0ms dummynet experiment that we saw a ~20ms overhead in our Paxos leader latencies.
* So while we would ideally expect to see a 25ms Paxos leader latency graph, it would be offset by the overhead, moving up our expectations to around 45ms.
* The client end to end latencies can be split into dominant factors:
  + The client-leader RTTs for the initiation and migration agent ack rounds – ~20ms
  + The service time for the request – DB + code execution time
  + The Paxos time itself - ~20ms (1 RTT)
* Hence we expect a 20ms (for initiation and migration agent acks) + 20ms + 3-5ms (baseline Paxos) + 20ms dummynet overhead + DB + code execution time.

Results:



Client end to end latency



Paxos leader latency



Directory service time

With reasonable room for error, we note that most of our expectations were right.

4.4.4 DummyNet with 20 and 80ms delays

We simulate a setup with 3 machines on the east coast and 2 on the west coast. x = 20ms, y = 80ms.

Expectations:

* The convergence time of a single Paxos round measured at the leader would now be composed of 3 factors:
  + Code execution time and threading overhead.
  + Queuing latencies between the multiple asynchronous parts of the application
  + The convergence time for a single Paxos round would still be expected to be around 20ms as the leader replica has the ability to form a majority with the 2 other east coast replicas.
* Other expectations remaining the same as the previous case, we would still ideally expect to see a 25ms Paxos leader latency graph, it would be offset by the overhead, moving up our expectations to around 45ms.
* The client end to end latencies can be split into dominant factors:
  + The client-leader RTTs for the initiation and migration agent ack rounds – ~20ms
  + The service time for the request – DB + code execution time
  + The Paxos time itself - ~20ms (1 RTT)
* Hence we expect a 20ms (for initiation and migration agent acks) + 20ms + 3-5ms (baseline Paxos) + 20ms dummynet overhead + DB + code execution time.

Results:



Client end to end latency



Paxos leader latency



Directory service time

Our Paxos leader times as expected are still around the 45ms mark – validates our claim about the east coast majority causing consensus. Other results also match up as expected.

Something that we observe across the service time results above is that the initiation and migration agent acks times seem to be consistently higher than the other 3. This can be attributed to the following reasons:

* Database – Initiation is an insert operation. The other operations are all update operations. This leads us to believe that the DB optimizer somehow seems to favor performance of one over the other.
* Migration Agent Acks by design is a 3 step database operation due to its asynchronous nature.
  + First the migration agent acking is identified through its IP and port information.
  + Then the current migration progress for the object being migrated is looked up.
  + Finally, if this is an ack from an agent that hasn’t already acked, the entry is recorded.

5. Conclusion