ByePaxos
Learned Consensus for Total Ordering of requests on replicas

Shahid Ikram
shikram@illinois.edu

Ricky H. Hsu
rickyhh2@illinois.edu

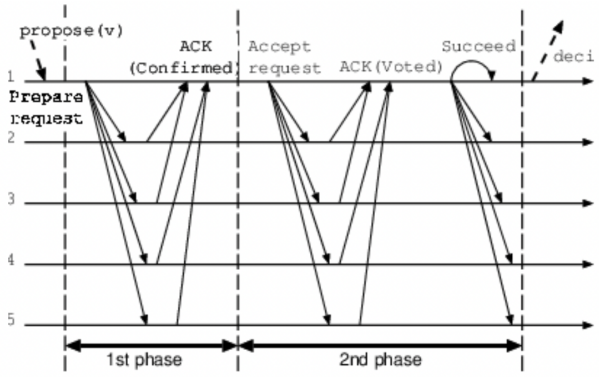
Abstract

Achieving system reliability in the presence of faulty processes calls for coordination on data values that is required for computation. In the case of a Replicated State Machine (RSM), we use protocols like Paxos, to ensure coordinated transition of states across the replica machines so that state changes are ordered in a deterministic way. The Paxos protocol incurs two round trip times in deciding a particular value across a set of nodes. Our work is a research on the possibility of eliminating the one extra round trip time that is required on deciding a total order of request execution. Similar work exists in the area that tries to tackle this problem, but approaches the multicast ordering problem at a different layer in the application stack as we shall see later on. Our solution approaches this problem at the application layer, making no assumptions of the underlying network. We explore Machine Learning (ML) based sequencing of requests, that are flooded to a set of replicas from concurrent clients. We also realised that eliminating the one round trip time would eventually render the coordinator process useless, and hence eliminate the need for a consensus protocol, as the role of the coordinator process would be embedded into the ML models that are deployed on each of the replicas. We also observe that by using ML we could train models to learn ordering specific to the application thereby making the ordering pertinent to the application using the RSM.

1 Introduction

A fundamental problem in distributed computing is achieving overall system reliability in the presence of a number of faulty processes. This often requires coordinating processes to reach consensus, or agreeing on some data value that is needed during computation. For instance, consensus protocol is used in deciding the order of execution of requests from the clients on the replica nodes. Given a condition where the clients issue a number of requests for a process/node which is maintained as a state machine, i.e, for a given request the state of the process transitions to a new state in a deterministic way, we need to ensure that all the requests are executed on all the replicas of that process in the same order. This is necessary to ensure that all the replicas are always in the same state. The way in which this is handled is by leveraging consensus protocols to arrive at a desired ordering of requests among all the processes in a replica set.

Below presented is a run of the Paxos algorithm, in deciding one value among a set of processes.



We can clearly see that a run of Paxos (agreement on one value) requires two phases to complete. The first phase is the prepare-promise phase, where the coordinator proposes a value and a quorum of nodes accept/reject the ongoing proposal. If a quorum of nodes agree on a proposal then the second phase kicks in wherein the coordinator informs all the processes in the system of the agreed-upon value.

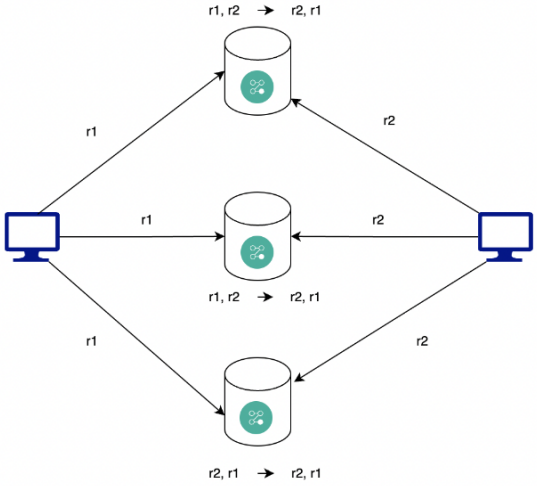
In case of an RSM, order is to be consensually agreed upon for a stream of requests. This would involve running one round of Paxos for each of the incoming requests, and reaching consensus on the ordering per request. This is could be expensive in case we have multiple operations coming at a process from concurrent clients. Moreover, for a coordinator to propose values, we would need to elect a coordinator and this election process is again a consensus problem.

This paper aims to reduce the extra overhead of 1 RTT introduced by Paxos. Although there are previous works in the domain like, FastPaxos, SpecPaxos, NoPaxos and Domino,

there hasn't been any work that leverages ML in solving the problem. In this paper we are going to present our ideologies, assumptions, challenges faced and the results seen on exploring the potential of designing ML based multicast ordering systems. In the subsequent sections of the paper, we will discuss a bit more on the background and motivation, followed by past related work in the domain and an insight into how the solution we propose is different from the existing studies, the solution architecture and algorithms designed and finally conclude by providing initial evaluation results seen as well as the future of extending the work.

2 Background and Motivation

As discussed earlier, the problem of ordering requests in an RSM requires agreeing on a sequence in which the each of the incoming request should execute across replicas. For a Paxos based sequencer, this would mean running multiple rounds of the Paxos, one for each request. With each round comprising of 2 phases, we incur a lot of overhead. After a coordinator is picked among a group, why do we need the prepare-promise phase? Can the coordinator not just decide on the sequence and inform all the other nodes? This would be an oversimplification in a distributed environment where reliability of processes is never guaranteed. Processes can fail at any time and the underlying network is an unreliable channel where message drops are not uncommon. In the scenario of a network partition, we could run into the "split-brain" condition, where there are more than one coordinator active in the system and each proposing values which would violate the "safety" of the consensus protocol. However, there is another optimised version of the Paxos protocol: MultiPaxos, that optimises the multi-round runs of Paxos. Rather than starting again with prepare requests, coordinator moves to a galloping mode where it sends successive accept messages when it hears a majority of acknowledgments for the previous accept request. Here we need minimal number of messages, but it only occurs for multiple rounds from a stable coordinator. This may be interrupted by the coordinator crashing or a network partition. MultiPaxos degenerates down to basic Paxos and hence the problem prolongs. However, if we could eliminate the coordinator process and its responsibilities of sequencing, we could gain significantly on the overhead in the network layer. If each of the replica node can consistently order each of the request it receives such that the sequence of that request across each of the replica is the same, then we have total ordering. From literature we learnt that, network round trip delays are relatively stable in a short time period and recent network measurements can be used predict current network behaviour. There are various features that contribute to the arrival of requests at a replica node with most of it being network layer features. Hence, a weighted computation of all such features should offer some information about a request's sequence number. We use an ML based approach which learns from the data distributions of past requests to eventually converge to sequencing requests on a replica node without need for communication with other replicas and arrive at a consistent order.



3 Related Work

The problem that we are trying to solve has been around for quite a while and there have been several studies, explicating ways to alleviate the extra overhead incurred with Paxos. We shall look into a few relevant researches:

3.1 Designing Distributed Systems Using Approximate Synchrony in Data Center Networks

This study stands on the foundational idea that "Distributed Systems today are designed to run in datacenters in which networks are more predictable, more reliable and more extensible". Going with this, the common intuition behind all their design decisions is that messages can be sent along predictable paths through the data center network topology with low latency and high reliability in the common case. They run a modified version of Paxos over their engineered network with fallback on a reconciliation protocol that runs the actual Paxos algorithm in case of inconsistencies binding the worst case performance to that of the traditional approach. They make a huge assumption about the underlying network and the usecase seems too rigid. Moreover, this optimisation requires additional support in the network layer.

3.2 Fast Paxos

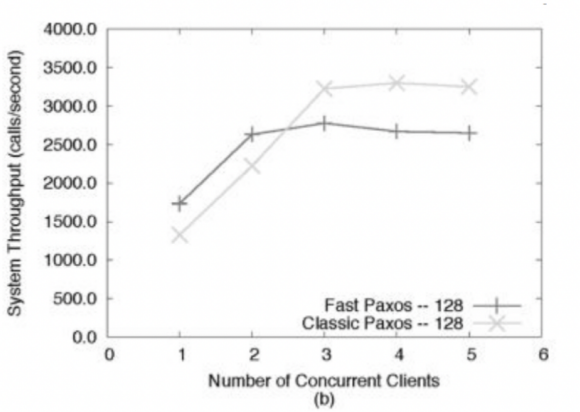
This protocol is just like Paxos but has two kinds of rounds:

- **Classic round:** Paxos

- **Fast round:** Clients skip the leader and directly propose the value to the acceptor which forwards the proposal

to the learners. If quorum, then success else fallback to classic round to resolve.

Fast Paxos performs well on when there are no concurrent clients issuing requests on the same object. (Image from the fast paxos paper depicting this shown below)



Also, we could have network geometry impacts that affect the overall throughput.

3.3 Just Say NO to Paxos Overhead: Replacing Consensus with Network Ordering

This study proposes a zero overhead implementation of sequencing requests, which relies on the network fabric to handle them, using software-defined networking technologies, advanced packet processing capabilities of next-generation data center network hardware.
What about bigger networks? Networks bigger than a datacenter. Moreover, this is also an optimisation done on the network layer.

3.4 Domino: using network measurements to reduce state machine replication latency in WANs

The main idea advocated in this study is that estimated arrival time of requests across a quorum of replicas can be leveraged to deterministically order the requests across the replicas. The authors show that recent network measurements can be used to predict the current network behaviour and this information is used to guide the estimation of arrival times across replicas. Domino runs both FastPaxos and the Classic Paxos and switches between the two based on conditions. If a majority of ack is received at the predicted time, then FastPaxos succeeds else slow conventional Paxos resolves the conflict.
This work is quite an exciting step in the direction of using network features in generating ordering of requests at the replicas at a higher level in the application stack. However, they consider a limited number of features and use heuristic based approach in generating estimations.

4 ML based sequencer for total ordering

Building on the foundational idea from Domino, we leverage over the facts that:
- Network round trip delays are relatively stable in a short time period.
- Recent network measurements can be used predict current network behaviour.

The broad questions that we seeked answers for through experimentation revolved around:
- Can we leverage ML to weigh network measurement features in determining when a request should be executed at each replica?
- What are the network features we should consider?
- Can other features like CPU and memory utilisation at the replicas, contribute to this decision? If so, how much do they actually contribute?

ML is best suited in generating decisions which involve weighing features according to the data distribution sample it is trained on, in order to generalise the solution.

We explored ML based solutions in generating total ordering of requests across replicas without the need for coordination. This involves training and deploying ML models on each of the replicas, that sequences any incoming request, taking into consideration features that affect the delivery of that multicast at other replicas. This ordering is governed by past delivery information gathered from logs of request-arrivals at each replica node and hence ordering module operates at the application layer unlike other solutions seen before that enforce ordering at the network layer.

4.1 What do the models at each replica do?

Each replica receives requests from the clients and makes an informed data driven decision of when the request needs to be executed. Requests are buffered on the replicas and each buffer is committed after a buffer commit timeout period. At commit, each of the request currently in the buffer is sequenced. A buffer can be committed if all the requests in the buffer are consistent across all the replicas. At commit step, the ML model ensures that requests in a buffer are placed such that they are consistent across each of the replicas. For the commit of buffer B(t), where t represents the current timestamp, ML model sequences all the requests in the buffer and keeps the requests it predicts would have arrived at all the other replicas at the current commit time t in the current buffer and moves each of the other requests to their respective buffers b(t+x) which represents a buffer that would be committed at a later time. This decision by the ML model basically means that the model predicts that for the given request, arrival times at other replicas would be at a later time and hence decides to commit that request at a later point when it thinks it would have arrived at each replica. ML model is

trained to learn ordering from the past data to initialize the model.

At the commit time, we have a buffer with requests that are to be committed. The requests in the buffer may still not be ordered in a consistent way across the replicas, but we ensure that the set of requests that are in the buffer committed at the current time are the same. Once we have the set of requests to be committed in the current buffer, we then choose a deterministic way of ordering the requests, for instance: sorted order of the request id's. This would ensure that all the requests that are executed at each of the replicas are the same and are in the same order.

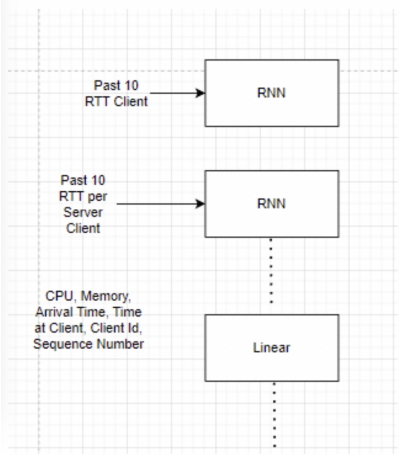
However, all this banks on the accuracy of the model to predict correct buffers for each of the request it processes.

4.2 Challenges

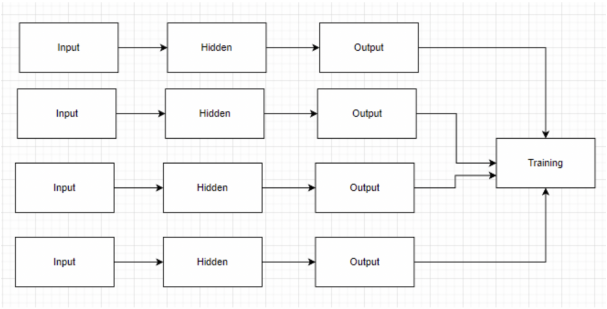
There are quite a few challenges involved in designing this systems, a few of which are that we need to balance the buffer commit time such that it does not subdue the gains obtained by reducing one RTT. Moreover, training of the model is challenging as the data is unlabeled and there is no right ordering of requests that can be learned. We cannot use Paxos's orderings on the past data to train the model because Paxos ordering does not capture any inherent network subtleties. An important thing to consider is that the model should not be too complex that the inference time is too high which again overshadows the one RTT gain.

4.3 Design

Each replica runs a model which takes the request as an input and returns the buffer commit time that the request belongs to. Each request is represented as a collection of features that capture the network conditions, conditions at the client, condition at the server etc. The model generates an embedding for a given request that weighs all the different features that constitute a request. The features that we have currently tied to a request include: request arrival time at the replica, request id, time at the client when the request was sent, duration since the last request from the same client, duration since the last request at the replica irrespective of the client, the client who made the last request that hit the replica, memory utilisation at the replica, CPU utilisation at the replica, RTTs of the last 5 requests sent by the client irrespective of the replica, RTTs of the last 5 requests from the client per replica node. This input request feature is converted into a dense embedding using sequence models and feed forward networks as shown below.



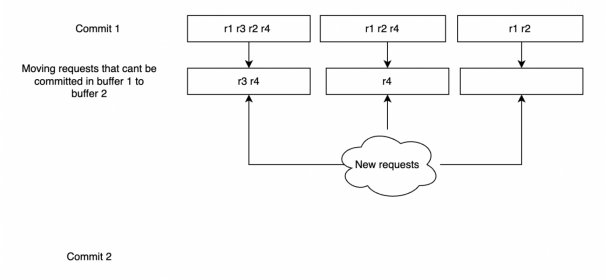
The problem boils down to finding the best representation of the input request at each replica such that the model generates the same buffer commit time for that request at all of them. This implies that training a model for a replica would be influenced by the weight updates and the decisions that are made for models at the other replicas. We have an offline training phase where we jointly train the models that belong to each of the replicas. We use the requests that were delivered at each of the replica from the log as training data. The system for offline training is as follows:



Here, each of the row represents the model at a replica node. We relay a set of requests from the logs to each of the model and do the training.
Since the requests data is unlabeled, we designed an algorithm that would use the model predictions itself while training to derive the correct label for the request per model and learn its embedding.

4.4 Algorithm to generate labels

For each epoch, the labels will be generated for each batch of requests. We use the current model to get a first prediction for each request. After a pre-determined time passes, commit all the requests in the earliest buffers that are the same across all the servers. Push the uncommitted requests to next buffer. Repeat the process until all requests are committed.



4.5 Training

For the training process, we currently utilize 9 different batches of data. As stated previously, during each epoch, the labels will be generated by the current iteration of the model using the algorithm. After the labels are determined, the current label will be treated as the ground truth for that duration. As the model essentially contain multiple solutions, there can be multiple values that the training optimize to. Therefore, we have a loss term that penalizes based on MSE loss when comparing the label output, request placements, and outliers. The overall loss equation is similar to the following:

$$loss = \sum_{i \in C} (\hat{y}_i - y_i)^2 + \gamma \sum_{i \in C} \text{argmax}_{\text{index}} \hat{y}_i + \beta(size(S) - size(C))$$

In this equation, γ and β are hyper parameters which indicates how much weight each of the loss terms should contain. At every epoch, C is the set of request that can be computed based on the shifting requests from the predicted buffer placement. For example, suppose that we have 20 buffers and request A is placed in the 0th buffer from our prediction. As long as request A only shifts at most 19 times to maintain total ordering across the servers, request is part of set C.

5 Implementation

The implementation of the model is as described earlier. There will be a model for every replicated server which are all trained at the same time. The three types of layers used in the model are embedding, RNN, and linear. The embedding layers are used to generate a sparse vector representation of clients. The RNN handles historical data such as previous RTT at the client and previous RTT per server at the client side. The rest of the features such as time of arrival are passed through linear layers. The output of all these layers will be concatenated and pass through a hidden layer and finally an output layer.

5.1 Algorithm Pseudocode

U is a set of uncommitted Requests
B is a list of buffers
D is a list of data for each server
M is a list of models
while U is not $\emptyset$ **do**
  for $r \in R_{D_i}$ **do**
    if r.arrival time is within commit time **then**
      $B_j \leftarrow r$ where $j = M_i(r)$
    end if
  end for
  $T \leftarrow \bigcup B_i[0]$
  $U \leftarrow U - T$
  for Each$i$ **do**
    $B_i[1] \leftarrow B_i[1] + (B_i[1] - T)$
    Update labels accordingly
  end for
end while

As stated before, at the beginning of every epoch, the true labels are computed based on current prediction and shift. One of the parameter of the model is the buffer time which will be used to repeatedly allocate request into the relevant buffers. As the time expires, the intersection of the earliest buffers will be committed and the reset are pushed into the next buffer.

Of course there is the possibility that certain requests cannot be possibly shifted while maintaining correct ordering. To solve for this, we remove the conflicting request from training in this epoch. The hope is that modification to the model with other requests will update the model such that, in the next epoch, this request can either be an exact match or can be shifted through this algorithm.

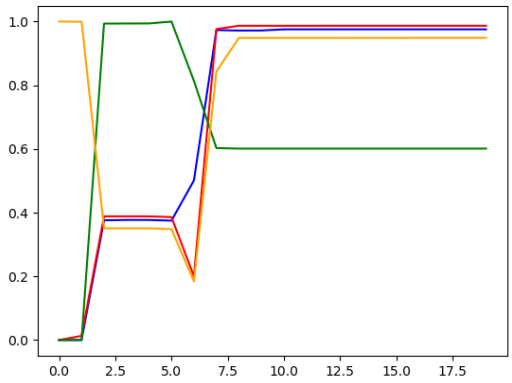
6 Evaluation

6.1 Data

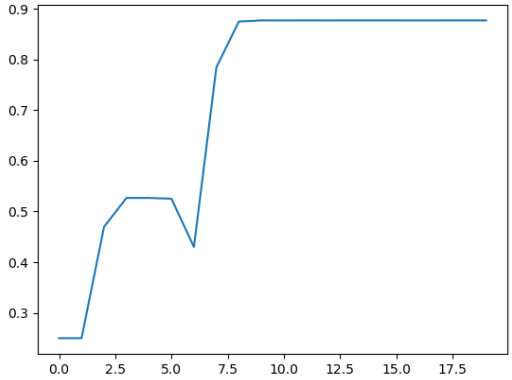
The data used to train the models are gathered from cloudlab. In total there are 3 different clients flooding requests to 4 geo-replicated servers. The data consists of certain client information as well as server information. Most importantly, historical data is sent through the request from the client to the servers. Additionally, the data is processed, normalized, and transformed in to useful features vector prior training.

6.2 Experiment

One of the experiments we conducted is to check the training accuracy for each model at every epoch. For our experiment, we decided to use buffer commit time of 0.1 seconds and 20 buffers total. As shown in the graph below, We trained the model for 20 epochs. The graph below show the model accuracies.



Each colored line is a separate model representing a replicated server. The fluctuation of accuracies is normal since we have multiple loss terms. Eventually, the model do converge into a high total accuracies as shown in the image below:



The total accuracies is the summation of correct predictions for all models over total requests from all models. Overall the results are as expected since the models try to optimize convergence.

6.3 Limitations

Due to the complexity of the training there are couple of glaring issues to be careful about. First, the models are very sensitive to weights. Depending on the initial weights, there could be problems with the shifting algorithm deeming all request to be non-computable. To combat this, we set the initial weights of every layer to zero. However, we still notice issues with initial labelling due to the bias terms in the layers.

Another problem with this approach is that the loss function needs to be carefully calibrated. Specifically, the γ and β term needs to be selected properly to provide adequate push to the loss while not completely overwhelming the MSE loss.

Lastly, buffer count and the buffer time also matter. If the buffer time is too large, most requests will be placed in the first buffer which makes the problem trivial. Additionally, large buffer time means that clients will have to wait longer for responses. Low buffer could

7 Future Work

We currently have only explored offline training of the models that are deployed om the replicas. We can integrate an online learning phase that would periodically update the models on each replica using the data that is logged on the replica nodes since the last time the model was updated. We could have an auxiliary node that pulls in the models from each of the replica nodes periodically, fetch the logs, extract the data and rerun the model and update the weights.

The current policy used in generating the final ordering of the requests in a buffer is a rather simple one, where we just sort the requests based on their IDs. Since we are ordering the requests at the application level, we can leverage application feedback in tuning the ML models to get orderings that could be beneficial to the application. For instance, a consensus protocol based concurrency control for distributed transactions could ensure that the ordering of the requests are such that they minimize the abort rate.

8 References

- Leslie Lamport, Paxos Made Simple, ACM SIGACT News (Distributed Computing Column) 32, 4 (Whole Number 121, December 2001), December 2001
- Paul Krzyzanowski, Understanding Paxos, November 2018
- Lamport, L. Fast Paxos. Distrib. Comput. 19, 79–103 (2006) https://doi.org/10.1007/s00446-006-0005-x
- D. R. K. Ports, J. Li, V. Liu, N. K. Sharma, and A. Krishnamurthy, “Designing distributed systems using approximate synchrony in data center networks” in USENIX NSDI, May 2015
- J. Li, E. Michael, N. K. Sharma, A. Szekeres, and D. R. Ports, “Just say NO to Paxos overhead: Replacing consensus with network ordering” in USENIX OSDI, November 2016
- Xinan Yan, Linguan Yang, and Bernard Wong. Domino: using network measurements to reduce state machine replication latency in wans. In Proceedings of the 16th International Conference on emerging Networking Experiments and Technologies, pages 351–363, 2020