All On Cloud 9

CS293B - Cloud Computing, Edge Computing, and IoT Project

Aarti Jivrajani, Abtin Bateni, Daniel Shu, Yiyang Xu

Overview

Our project is essentially an attempt to implement and improve CAPER[1]. In this report, we discuss the contributions and improvements that this paper makes. We also discuss the “plug-and-play” feature of our implementation which allows us to switch between various consensus protocols in order to evaluate our system design.

Motivation

The current digital world consists of a lot of collaborative tasks. In many cases, different parties are working for the same goal each maintaining a part of the responsibilities. Different organizations need to work with each other so that a single achievement can be made. However, there is always a huge problem among them. Trust! Different solutions and algorithms have been tried to address this problem from different aspects. Protocols like Paxos[2], Fast Paxos[3], EPaxos[4], etc. assume that the nodes fully trust each other and try to achieve consensus with this assumption. These protocols only tolerate crash failures. On the other hand, protocols like PBFT[5], MinPBFT[6], Zyzzyva[7], etc. assume that the nodes do not trust each other at all, and using this assumption tries to achieve consensus and are tolerant to byzantine failures as well.

While both of these approaches are very efficient, when used in distributed systems with appropriate fault tolerance criteria, none will work efficiently when dealing with nodes partially trusting each other. This means, assume there exist different groups of nodes that within each group the nodes fully trust each other but each pair of nodes from different groups have no trust at all. Please note that this condition is not very uncommon when different parties are collaborating with each other. Each party may consist of a group of nodes running in parallel while trusting each other since they belong to the same party. However, none of them trusts a node

[1] (2019, July 1). CAPER: a cross-application permissioned blockchain .... Retrieved June 2, 2020, from https://dl.acm.org/doi/10.14778/3342263.3342275

[2] (n.d.). Paxos Made Simple - Leslie Lamport. Retrieved June 2, 2020, from https://lamport.azurewebsites.net/pubs/paxos-simple.pdf

[3] (n.d.). Fast Paxos - Microsoft. Retrieved June 2, 2020, from https://www.microsoft.com/en-us/research/wp-content/uploads/2016/02/tr-2005-112.pdf

[4] (2013, November 3). There Is More Consensus in Egalitarian Parliaments. Retrieved June 2, 2020, from https://www.cs.cmu.edu/~dga/papers/epaxos-sosp2013.pdf

[5] (n.d.). Practical Byzantine Fault Tolerance - MIT. Retrieved June 2, 2020, from http://pmg.csail.mit.edu/papers/osdi99.pdf

[6] (n.d.). Efficient Byzantine Fault Tolerance - Informática. Retrieved June 2, 2020, from http://www.di.fc.ul.pt/~bessani/publications/tc11-minimal.pdf

[7] (n.d.). Zyzzyva: Speculative Byzantine Fault Tolerance - Computer .... Retrieved June 2, 2020, from https://www.cs.utexas.edu/users/dahlin/papers/Zyzzyva-CACM.pdf

from outside of the party. Hence, we are looking for a mixture of the crash tolerant and byzantine tolerant protocols so that we can efficiently use the system resources like CPU and network bandwidth.

Moreover, establishing a consensus is not the only problem. Many times, the nodes within the party need to confidentially inform each other of a recent event while not letting the other nodes know. However, they may sometimes want to refer to these private events in global events as well. While the private events remain private to those nodes, they are able to relate the global activities to the private incidents.

CAPER, is the solution that we found for this problem. Each node is provided with a public and private blockchain that can store its events in them accordingly. We want to maintain the confidentiality of separate private blockchains while still maintaining good performance. This is done by maintaining application-specific views on each node instead of replicating the entire ledger. Each node will then only see their own transactions and any cross-application transactions that pertain to them. The ledger will be modeled as a DAG.

We also wanted to implement this in a way to make the consensus pluggable thereby allowing us to easily switch between different algorithms. This way, our application is versatile and can be modified to tolerate different standards.

Motivating Example

Supply Chain serves as a great use case of blockchain. Supply-chain has a highly collaborative workflow where different parties need to communicate across organizations to provide services. The collaborations are defined in service level agreements(smart contracts) which are agreed upon by all participants. The participating entities could be - manufacturer, buyer, carrier, and supplier.

In our report and demo, we use this example of supply chain to explain various aspects of our implementation.

Features

Consensus Plug and Play

Oftentimes, a distributed system is designed in such a way that the edge applications(clients) are tightly coupled with the core underlying algorithms(the consensus algorithms in this case). This makes it difficult to evaluate the multiple features of a distributed system. Our implementation decouples the underlying blockchain skeleton from the consensus algorithms used to decide the blocks to be added to the blockchain. This allowed us to “plug-and-play” consensus algorithms and collect scale metrics. This design also keeps the system extensible.

Consensus algorithm selection based on means of inter-node communication

Nodes that are communicating with each other can freely send each other different messages and if they are able to behave maliciously they can try messing with the consensus establishment. However, by using trusted hardware, we are introducing one more restriction to the system. Nodes cannot send two different messages to two different nodes and they either send a message to all or do not send it all. This restriction known as **using trusted hardware** will help us use a faster protocol - **MinBFT protocol**. As we discuss later, it helps us reach consensus faster. However, we took into consideration the case where using such a service is not possible. A future user can simply substitute NATS with whatever communication tool they prefer and therefore they may remove the previous restriction. As a result, we considered implementing the original PBFT protocol as well to ease the process of migrating to non-trusted hardware infrastructure.

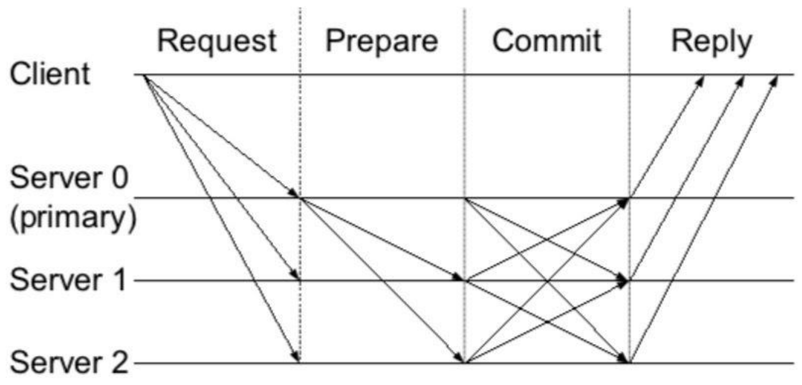

Figure 1: An overview of the execution of MinBFT to handle one message (photo taken from the paper). MinBFT due to its nature does not require the pre-prepare phase of the consensus

Blockchain Views and Privacy

CAPER is a permissioned blockchain system that supports both internal(local) and cross-application(global) transactions of collaborating distributed applications. Usually, permissioned blockchains suffer from confidentiality issues since a single blockchain ledger with all the transactions is maintained at every node. Agreement on the shared state of collaborating applications is needed without trusting a central authority or any particular participant. On one hand, the cross-application transactions are visible to all the applications, the internal transaction of each application needs to be kept confidential. The CAPER blockchain is modeled as a Directed Acyclic Graph(DAG). Each application orders and executes its internal transactions locally while cross-application transactions are public and visible to every node. **The blockchain ledger is not maintained by any node. Each application maintains its own local view of the ledger including its internal and all cross-application transactions.**

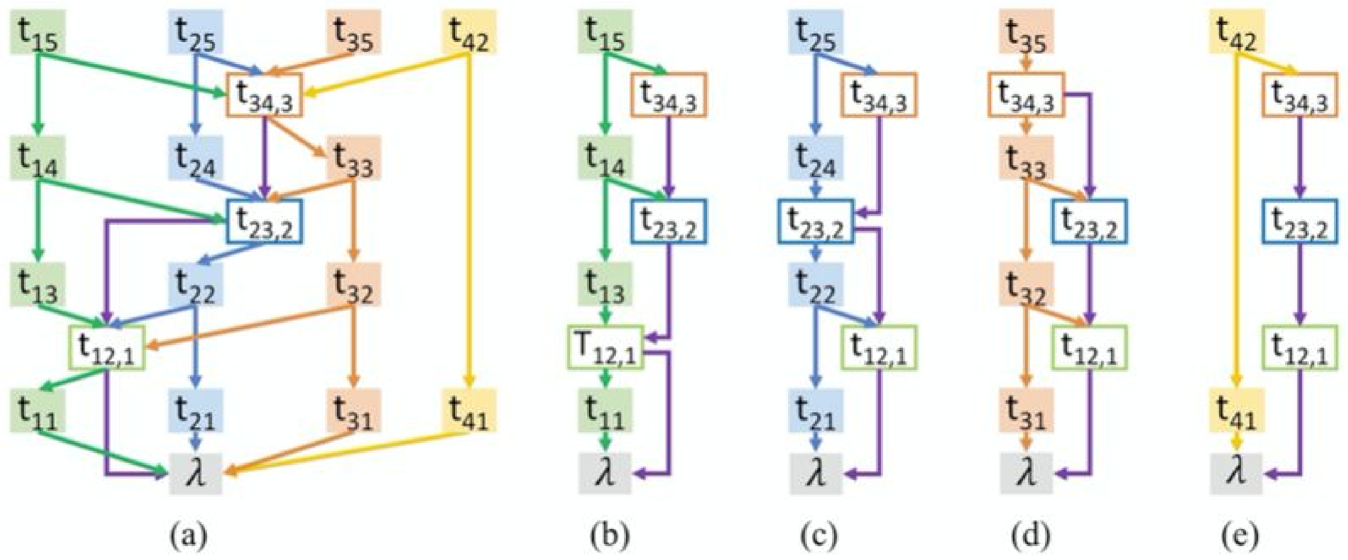

Figure 2: (a) A blockchain ledger consisting of 4 applications, (b), (c), (d) and (e): The views of the blockchain from different applications. (Photo is taken from the CAPER paper)

System Architecture

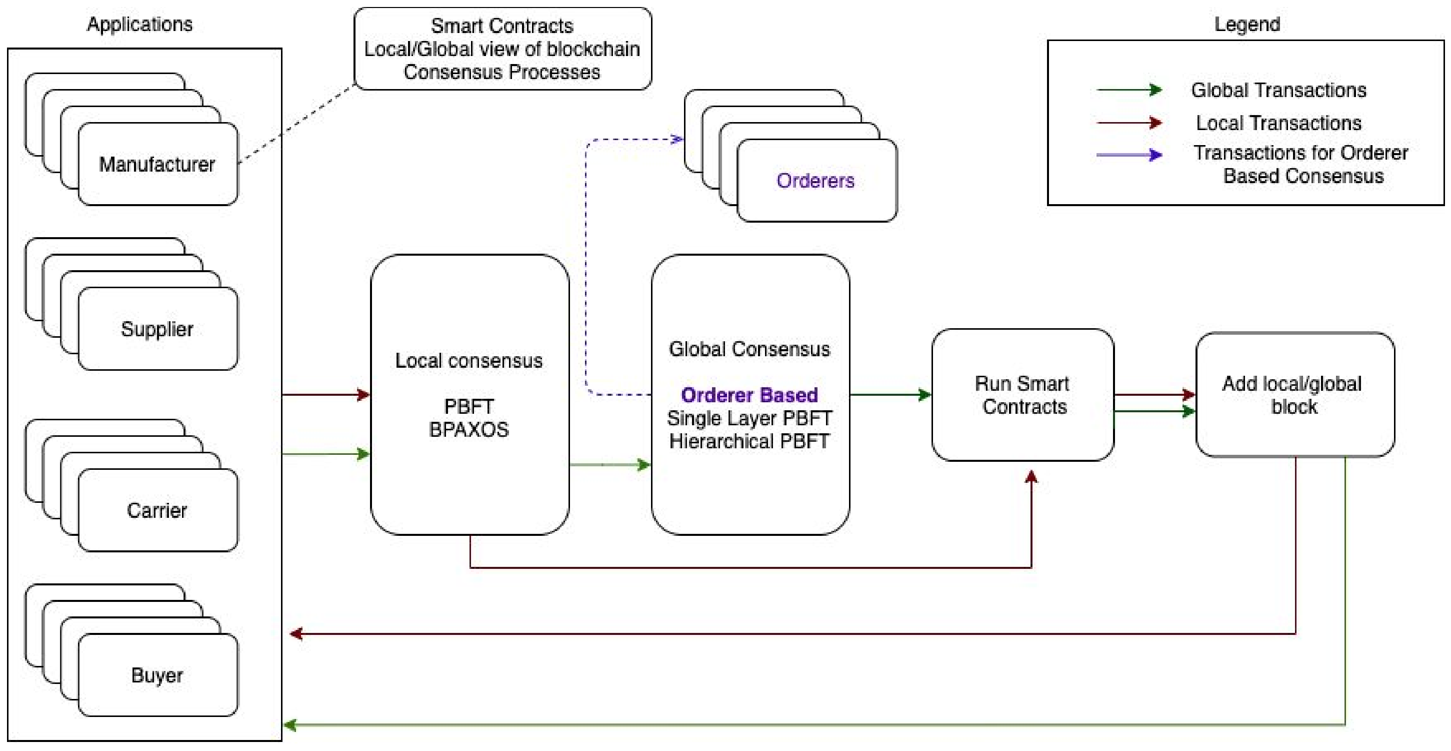

Figure 3: Bird's eye view of the system architecture

Our implementation of the CAPER blockchain consists of the following modules -

Applications

This module consists of the business logic of the system. Each application consists of the following:

- Its own view of the blockchain; which is a DAG
- Smart contracts: These are processes running within the applications which are used to check the validity of the transaction before they are added to the blockchain.

Local Consensus Module

From an implementation point of view, the "local consensus" module consists of distributed consensus algorithms that help achieve global and local consensus. We implemented BPaxos, PBFT, and MinBFT consensus algorithms.

BPaxos

BPaxos is a modification of Paxos that is meant to be able to scale well with an increase in the nodes. It does so through **modularization, separating out the bottleneck components.** BPaxos separates out the components for selecting a leader, determining any dependencies between a proposed value and previously committed values, proposing a new value, achieving consensus, and replicating the result. For this project, the consensus module in our implementation was based on the majority vote, but any consensus algorithm could be used.

Implementation Deviations from BPaxos

- Our application does not require the dependency modules, and because of this modification, we could merge the proposer and leader functionality, leaving only three modules (leader, consensus, and replica)
- We used NATS as a single point of communication instead of single point-to-point communication as described in the paper, which might have had an impact on our performance. We chose NATs because it provides several communication features, but it adds an overhead.
- The timeout was statically set to 2 seconds, so if the nodes failed to gain consensus, then the long timeout would lower throughput. We did not dynamically determine the timeout based on the workload. This 2-second timeout was a very generous upper bound to ensure that it would not time out too early, but most likely a lot of this time was wasted.
- BPaxos described in the paper has each leader determines values for different vertices in a dag, but for us, each leader was proposing values for the same vertice

PBFT

PBFT is the standard solution to tolerate Byzantine failures in a network. Each node can act maliciously and beside the possibility of crash may lie or alter the messages or try to interfere with the process of establishing consensus. Each standard deployment of PBFT using 3f+1 nodes is able to tolerate up to f faulty nodes. We used the standard PBFT protocol introduced in its original paper in 1999.

Global Consensus Module

CAPER introduces 3 different cross-application consensus algorithms:

1. Orderer based consensus
2. Single Layer PBFT
3. Hierarchical PBFT

While (1) required us to spin-off dedicated nodes in order to reach consensus, (2) and (3) are implemented as libraries - which allows us to further enhance the plug and play capability of our system.

In addition to implementing the 3 global consensus algorithms introduced in CAPER, **we also designed a hierarchical protocol to establish consensus considering trusted hardware/communication channels.**

Blockchain

The CAPER blockchain is not maintained per node, but a local view is maintained per node per application. This means that every node stores its transactions in a blockchain consisting of local(intra-application transactions) and global(cross-application transactions) blocks. Each block contains only one transaction, however, it is possible to batch the transactions into blocks.

Smart Contracts

Smart contracts are the “service level agreements” between the collaborating distributed applications. These smart contracts are processes running in the background which validate every transaction(post consensus) before a block is added to the blockchain. For example, in the case of the supply chain management example discussed above, a supplier may send an order transaction to the manufacturer asking them to make X units and also sending some amount in advance. The smart contract can check the validity of this transaction by summing the total price of the goods and shipping charges and comparing it to the amount being transacted. The transaction is not added to the blockchain in case of a contract violation.

Communication Channels

We use NATS for messaging between the nodes - be it application nodes or the orderer nodes. NATS is highly scalable and secure - providing TLS out of the box. NATS is also lightweight and easy to deploy. NATS works on a log-based data structure - this means that a “configurable” number of messages are stored in NATS at any given point of time, and can be retrieved based on both sequence number of transactions and timestamp. Also by using nats as the means of communication between the nodes, we apply a restriction to our system that nodes cannot send different messages to different nodes. A node either sends the same message to all the nodes or does not send it at all. This restriction enabled us to use algorithms for consensus-based on trusted hardware. Later we discuss how these algorithms can be helpful.

Choice of Consensus Algorithms

The clients are able to choose from different protocols to establish consensus. Regarding the underlying environment of the system, clients can choose from three different protocols to establish consensus.

- If we only assume crash failure for the nodes and assume that the nodes cannot act maliciously, the BPaxos protocol can efficiently establish consensus.
- Considering the presence of byzantine activities in the network, we have also implemented the PBFT protocol to tolerate malicious behavior.
- Since we are using NATS, as discussed earlier, nodes are not able to send different messages to the other nodes. They either send a message to all of the nodes or do not send the message at all. This restriction also known as using trusted hardware lets us use the MinBFT protocol which has fewer phases than the PBFT protocol and can achieve consensus faster.

We also ran a performance benchmark to compare the algorithms. We used a very limited set of nodes since we were only curious about how these protocols compare with each other. With only three manufactures and three suppliers and one proposer, three consensus nodes, and one replica for the orderer based approach, we achieved the following results[Fig 4]. We speculate, orderers are the fastest since they require only an instance of local consensus and a consensus among the orderer nodes. The second fastest is the Single Layer PBFT which has no hierarchy and after three phases is done. Hierarchical MinBFT using trusted hardware has a phase removed and it is expected to be faster than the Hierarchical PBFT. Note that our results vary from our performance benchmarks since we used the minimum number of nodes required for deployment only to make this comparison. If we had used more nodes and had deployed more applications, then our results would become more similar to the performance tests.

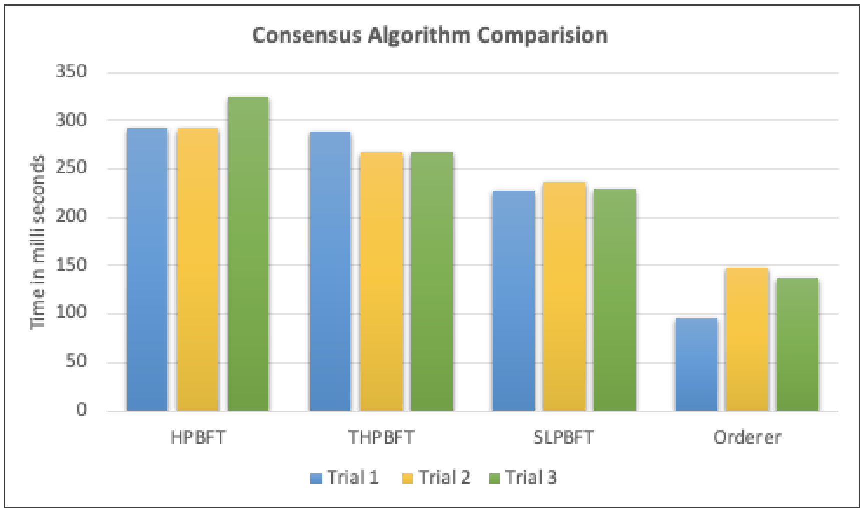

Figure 4: Performance comparison of the different algorithms

Fault Tolerance

CAPER is able to tolerate failures with different assumptions. Since each protocol sets its own threshold for the minimum number of nodes to tolerate up to f failure in the system, using different algorithms we require a different number of initial nodes to tolerate f node failures.

- If the nodes are using the BPaxos protocol, using 2f+1 nodes we can tolerate up to f failures.
- If the underlying protocol is PBFT, using 3f+1 nodes we can tolerate up to f failures.
- Assuming the trusted hardware condition and using the MinBFT protocol, we can tolerate f failures using only 2f+1 nodes.

Using NATS as our communication tool, we can neatly and effectively store the messages sent and received in the system. The NATS server can be configured using the typical fault tolerance techniques (e.g. RAID) to securely store the data on disks while considering disk failures. Note that this technique is not uncommon as almost every machine in a data center is using this technique or the similar ones to ensure that in case of failures we will not use our data.

Fault tolerance in byzantine environments is also related to secure communication. Malicious nodes should not be able to interfere with the other node's messages. Therefore, we needed to ensure a secure message transfer and message integrity. At first, we tried to implement integrity using the RSA algorithm and SHA256 hashing. Although we successfully implemented this feature for our protocol, we later realized NATS is also able to provide secure communication and since

Deployment Model

We chose to deploy our application on an AWS EC2 instance running Ubuntu 18.04. Each application in the system is containerized and each of these containers is deployed in separate Kubernetes deployments. We chose to containerize our applications primarily to maintain uniform runtime and manage virtual images of our application easily. For container lifecycle management, we chose Kubernetes cause of some of these features:

- Easy to scale: Easy interface to scale up/down the replica of the deployments
- Fault tolerance: If a pod crashes, it comes back up without any user interference

- Easy configuration: Since testing our project relied heavily on tweaking configurations of the applications; like fault tolerance allowed, number of replicas, type of local/global consensus algorithms to be used, we needed a quick way to configure and deploy the applications. Mounting configmaps and restarting the pods allowed us to do just that in a streamlined way.
- Easy networking: Easy built-in networking between pods. CNI allows us to do this quite easily.
- Convenient versioning: Can easily set the version numbers inside a container
- Dependency maintenance: Do not need to worry about version updates breaking code

Scale

We tested how our consensus scales with multiple concurrent requests. We scaled the number of proposers and the number of requests for our BPaxos module and measured the throughput and had fixed 2-second timeouts for re-proposing values. Since testing on AWS would be more expensive we chose to use Eucalyptus as our infrastructure. To be able to perfectly focus on the graphs, we decided to put them separately in the appendix. You can find our performance-test results in the appendix I.

Our results show that having too few proposers to requests or vice versa led to increased throughput times, and sometimes due to the nature of our fixed timeouts, livelock occurred. With too many proposers or requests trying to gain consensus, no work could be done. There is a sweet spot that yields the same average throughput as the number of requests scale.

These results do not match the throughput of the high BPaxos paper. We explained the possible reasons for this deviation in the system design section.

Conclusion and Project Plan

To start with, our main aim was to implement the CAPER blockchain along with the unique global consensus algorithms introduced in the paper. This paper treats the consensus modules like a black box - which meant that we were free to implement any algorithm we wished to. While looking at scale metrics of multiple consensus algorithms, we came across BPaxos - which due to its modular nature looked promising from a scale point of view.

While using the NATS request-response communication paradigm, we realized that due to its pub-sub feature as well as the TLS security that it provides, we could further simplify the PBFT algorithm and reduce the number of phases required in the algorithm.

At the outset, we began tracking our project plan with the CAPER implementation, but while doing so, we came across a scalable solution to implement Paxos. We also ended up leveraging the feature-rich nature of our communication module which further allowed us to simplify our consensus algorithm - and allowing us to reduce the number of nodes required to achieve consensus. On a concluding note, considering the plug and play feature of our implementation, users can decide to move away from NATS and use a simpler point to point mode of communication, and the CAPER system would still work as expected.

Acknowledgment

We would like to thank Mohammad Javad Amiri - the first author of CAPER - whose inputs were invaluable while designing our system.

Appendix I:

We have included our test results on the following pages, each test on a separate page. Note that for our testbed we used EMI: xenial-server-cloudimg-amd64-disk1, Disk size: (hi1.4xlarge).

Test 1

5 proposers
5 consensus
5 replicas

Trial #	Time for 50 requests(s)	Time for 100 requests(s)	Time for 200 requests(s)
1	4.052417678	6.023267983	6.05747131
2	6.014698557s	6.023603942	6.137788142
3	4.049516944	6.104722877	5.146985346
4	5.060227131	5.125787179	8.013464527
5	6.030925991	6.056897832	5.137939204
AVG	5.0415572602	5.8668559626	6.0987297058

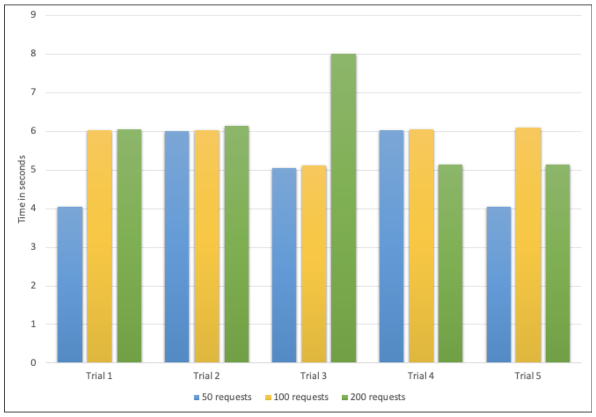

Figure 1: Performance of requests to 5P, 5C, 5R

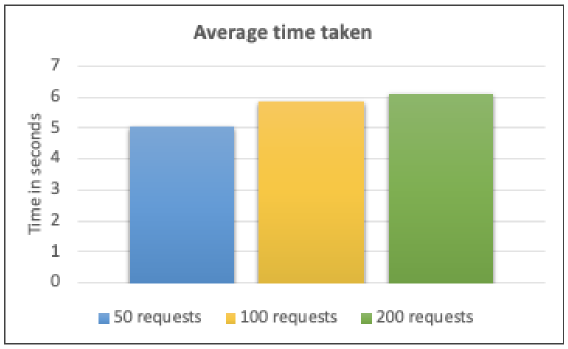

Figure 2: Average Performance of requests to 5P, 5C, 5R

Test 2

50 proposers
5 consensus
5 replicas

Trial #	Time for 50 requests(s)	Time for 100 requests(s)	Time for 200 requests(s)
1	18.02451609	22.029810434	20.067494101
2	18.015938229	20.017982078	22.054381063
3	18.022694332	16.012476306	18.039661814
4	20.029388473	18.016105008	Livelocked
5	24.020437844	18.024674997	Livelocked
AVG	19.62259499	18.8202097646	Indeterminate

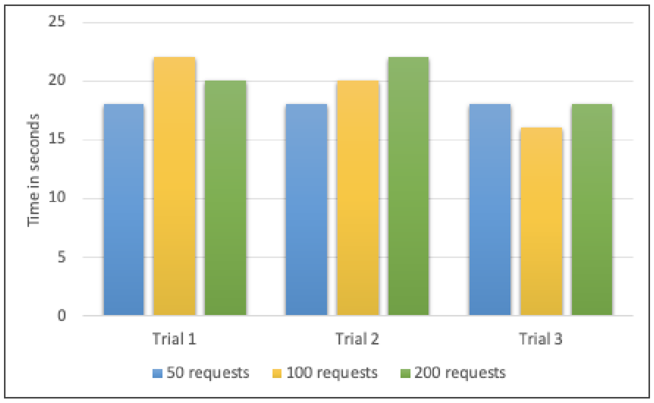

Figure 3: Performance of requests to 50P, 5C, 5R

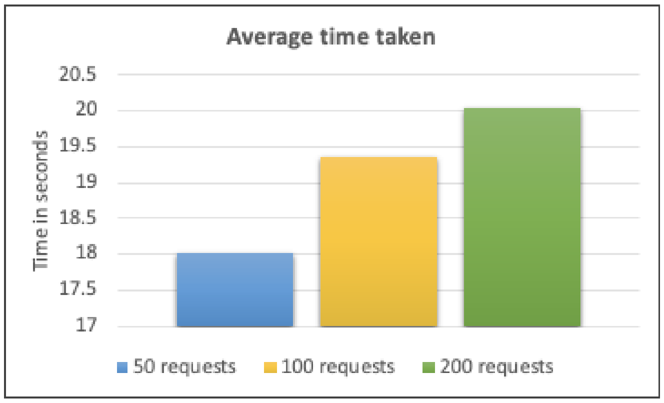

Figure 4: Average Performance of requests to 5P, 5C, 5R

Test 3

# Proposers: 50
# Consensus: 50
# Replica: 50

Trial #	Time for 50 requests(s)	Time for 100 requests(s)	Time for 200 requests(s)
1	18.147361538	18.285851579	Livelocked
2	20.202824074	12.344094139	44.03710731s
3	18.149623661	16.21896535	44.038525795
4	16.124502459	16.361339079	Livelocked
5	20.130738548	14.293177964	Livelocked
AVG	18.551010056	15.5006856222	Indeterminate

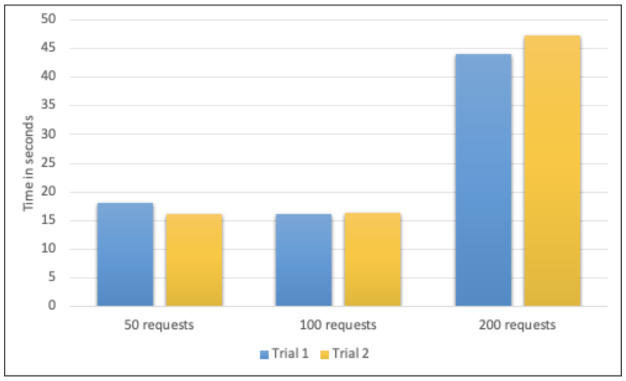

Figure 5: Performance of requests to 50P, 50C, 50R

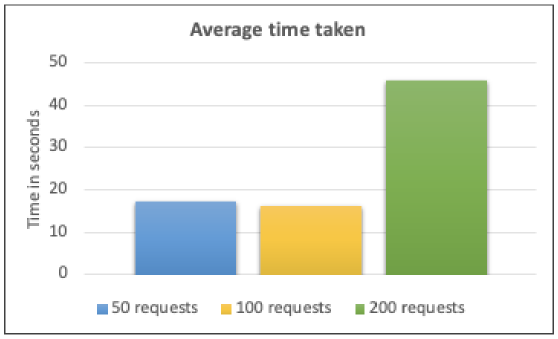

Figure 6: Average Performance of requests to 50P, 50C, 50R

Test 4

# Proposers: 70
# Consensus: 50
# Replica: 50

Trial #	Time for 50 requests(s)	Time for 100 requests(s)	Time for 200 requests(s)	Time for 300 requests(s)
1	14.173244301	18.048044282	18.573795826	29.070939283
2	18.139977811	14.297005253	19.05055489	24.393602175
3	14.098967591	20.347070503	24.471226522	Livelocked
4	14.173244301	18.048044282	18.573795826	29.070939283
5	18.139977811	14.297005253	19.05055489	24.393602175
AVG	16.1265229576	17.8767078674	20.1608918948	Indeterminate

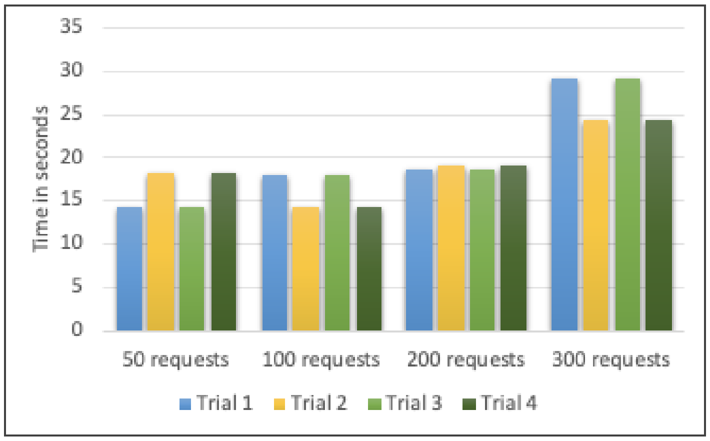

Figure 7: Performance of requests to 70P, 50C, 50R

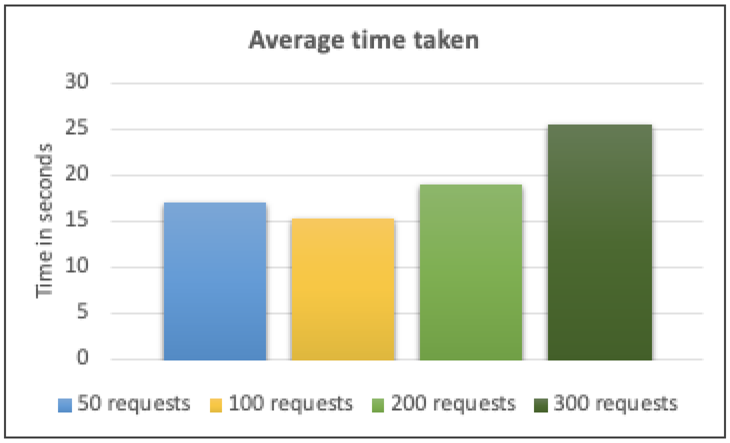

Figure 7: Average Performance of requests to 70P, 50C, 50R

Test 5

# Proposers: 100
# Consensus: 50
# Replica: 50

Trial #	Time for 50 requests(s)	Time for 100 requests(s)
1	18.180137115	68.07072608
2	18.156829889	84.085222472
3	18.204854156	78.076038828
4	18.223068381	72.063563346
5	18.044953384	80.068982212
AVG	18.161968585	76.4729065876

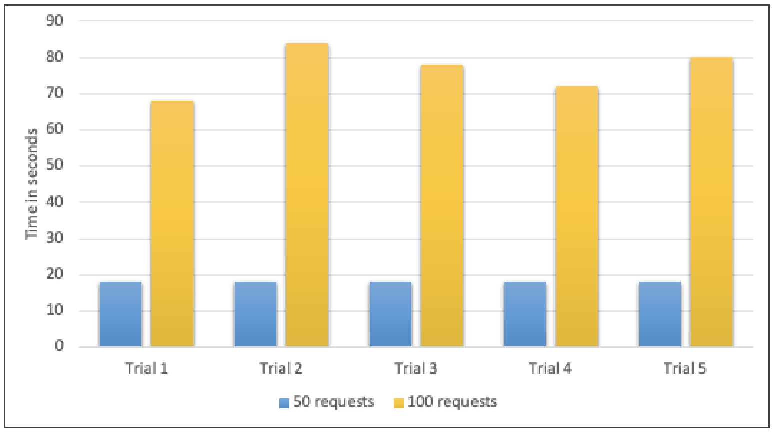

Figure 8: Performance of requests to 100P, 100C, 100R

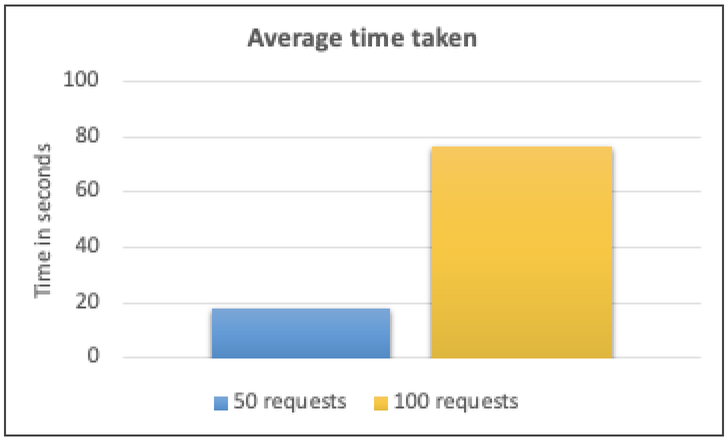

Figure 9: Average Performance of requests to 100P ,100C,100R