TESTING THE XRP CONSENSUS MECHANISM UNDER STRESS

Rishabh Krishnan, Rohan Khandelwal, Rushil Kapadia, Nikhil Mandava, Nathan Zhang,
University of California, Berkeley
Berkeley
{rishabh.krishnan, rohan.khandelwal, rushilkapadia, nikmandava, nathanzhang1}@berkeley.edu

ABSTRACT

The Blockchain@Berkeley team has worked under the David Schwartz (CTO of Ripple) to create a simulator of Ripple's distributed consensus mechanism which relies on separate nodes to reach a decision independently based on their list of trusted neighboring nodes. After creating the simulator, the Blockchain@Berkeley team ran experiments to test the robusticity of Ripple's consensus mechanism when faced with different factors.

1 Introduction

Ripple is a protocol that bases authentication of transactions off of a distributed ledger rather than proof of work or proof of stake. Authentication has historically been a tricky problem. In a network of many different participants, some of whom are malicious, how will a coordinated decision be made? A distributed system must be able to discern correct transactions from fraudulent transactions without the trust in a central authority that a traditional centralized system would take for granted. Another obstacle in distributed systems is latency; even if a transaction is correct it would be meaningless if a single node in the system took a year to respond and held back the entire network.

In the Ripple system, a node determines the correctness of a transaction from its UNL, unique node list, which is a subset of the whole network trusted by the individual node. It is organized in such a way to prevent collusion in an attempt to defraud the network. To reach consensus, each non-faulty node must reach a decision in finite time, and all non-faulty nodes must reach the same decision value, either 0 or 1. Based on the Ripple Whitepaper, the authentication mechanism progresses in steps. Each step of the authentication mechanism has a cutoff for validation and only transactions that receive more than the cutoff are moved on to the next round. The cutoff is raised each round until it reaches 0.8. All transactions that pass the final round are added to the ledger.

The strength of a consensus algorithm is usually measured by the amount of faulty transactions that it could process. Transactions could fail for many reasons. For example, there could be a malicious node trying to mislead the entire network or there could be a dropped connection between two nodes for whatever reason that hinders communication between the two nodes. Another issue with the network could possibly be the connection patterns of the network itself. If a certain subgraph of the total network has few connections to the rest of the graph, it will be easier for a malicious node to influence that subgraph contrary to how the rest of the graph works. We try to observe how the mean latency and the connections in a certain network will affect how likely it is for the network to come into consensus, in other words, agree on a proposal.

Our goal with this project is to examine the impact of such noted faults in the system and ultimately test the robustness of Ripple's consensus mechanism. We developed a simulator that could help us discern the protocol's reaction to various events and whether we would expect such a reaction or not. We do this by creating random underlying graphs that follow specific bounds that represent the underlying topography of the ripple network and send transactions with specific parameters through the network, looking for irregular behaviour. We primarily tested two major areas. Firstly, does the underlying topography of the the network significantly impact the ripple consensus? We believe that with low connectivity, we will experience a greater failure rate. We also hypothesize that with greater standard deviation in latency, we will experience a much larger delay in time, but we will not encounter failure. Secondly, how does the UNL and how it is chosen impact the consensus algorithm? We hypothesize that small UNL

sizes (similar to small connectivity) will lead to greater failure and that having a common UNL set should decrease failure rate in general as a shared UNL will lead to more similar results for nodes. We proceeded to run multiple experiments to test out these hypotheses.

2 Methods

In order to test out XRP's consensus mechanism, we created our own simulator to represent the consensus process for a singular transaction. Given input parameters such as the number of nodes and average connection size, we randomly generate a graph representing the Ripple network. Each of the nodes in this graph are servers that can communicate with other connected servers. They send out proposals which contain a binary vote for the transaction, and relay proposals sent to them to surrounding nodes. A server starts out with a randomly initialized vote in its initial proposal. Each server has a UNL, which is a set of trusted validators that the server relies on to update its own proposal at different thresholds of agreement: 50, 70, and 80 percent. Once a server reaches 80 percent agreement on the transaction, it broadcasts the binary result. We declare success if all servers reach consensus and broadcast the same result. Metrics such as time to consensus and conflicting broadcasts were recorded.

Throughout our experiments, all random variables were determined by sampling from a normal distribution developed by an inputted mean and standard deviation. In order to run simulations, we execute actions asynchronously through a priority queue based system. Latency was introduced into our system by adding a random latency to each sent proposal based on the latency distribution of each node, which in turn determined where the action was executed on the priority queue. We conducted tests on varying UNLs and graph architectures to measure the robustness of Ripple's consensus mechanism. The parts of our simulator are described in more depth below.

2.1 Proposal

Contains all the relevant values that define a vote sent from one node to another. Each vote contains an id which is a unique identifier for the proposal, a timestamp indicating when it was sent, and a value which indicates the actual vote of the sender.

2.2 Simulator

Represents the overall simulation and manages each individual node in the graph as well as their actions.

Init: Contains environment variables which are defined such as avgUNLsize and sdUNLsize which would set the average and standard deviation of the size of each node's UNL among other environment variables that are set by the caller. Timestamp is initialized to 0 and a graph is initialized with nodes containing a unique node id and a server class. Then, randomized connections are created between nodes in the graph based on the avConsize and sdConsize or average/standard deviation of connection size. For each node, a UNL is chosen as well based on the average and standard deviation of UNL size as well as the proportion that should be from a common UNL set. We also initialize the global action heap that keeps track of all the actions that nodes send one another.

AddSendAction/AddReceiveAction: Adds send or receive actions to the globalactionheap sorted in increasing order by timestep. Each action besides timestep, also contains a type indicator which indicates whether it is a send or receive action, id of the sender, id of the proposal, and id of the receiver.

ExecuteNextAction: Pops off the first action in the globalactionheap and determines action based on the type indicator of the action. Then it calls the corresponding function in server to execute the action. There are three main actions that can be taken: a send action to send a proposal to neighboring nodes, a recieve action to process proposals, and a broadcast decision to have the simulator process a node's final broadcast. If globalactionheap's length is 0, function simply returns.

Run: Initializes each node in the graph and adds an intial send action for each node based on their vote. Runs a while loop that calls ExecuteNextAction until the cutoff times (which is defaulted to 100 times the number of nodes in the graph) or when there are no more actions to take. Upon ending, if all broadcasts from nodes throughout the simulation are all positive or negative, we declare a success, if we do not have any broadcasts, we return failure case 1, and if we get conflicting results (both positive and negative broadcasts), we display failure case 2.

2.3 Server

Contains parameters that define a single Server (node), including a unique server id, the server's random latency, the candidate set of proposals to examine, a boolean for whether the server is malicious, and a probability with which a

malicious server will act maliciously. Server is also initialized with a UNL set, along with a dictionary of its knowledge. Its knowledge is the stored vote it has for any server in its UNL. If a proposal with a vote comes in that differs from the knowledge dictionary, given that the timestamp of the proposal is greater than that of the knowledge dictionary, we update this knowledge dictionary.

Send: If the proposal hasn't already been processed before, it gets sent to all neighboring servers with the queue latency incorporated. We make sure that the proposal is only sent to neighbours it has not already visited by using a cache that keeps track of which nodes all proposals have visited.

Receive: We check if the recieved proposal is from a sender that is in the reciever's UNL set. If it is not, the node passes on the proposal through a new send action. If the sender is in the UNL set, the Validates the proposal and updates the Server's knowledge dictionary. We also check if the number of either positive or negative votes from the UNL exceed the current threshold, if it does we do one of the following: if the threshold is .5 or .7, we increment the threshold to .7 and .8 respectively and create a new proposal for the new vote of the current node; if the threshold is .8, we broadcast a decision which is done by creating a broadcast action.

3 Results

We conducted 11 experiments to test the robustness of the simulator whose results are displayed on the 11 tables above. For each experiment we had multiple trials. To make sure that the results were thorough, we ran 100 runs for each trial. In each run, we tracked 10 input values as well as five results. The positives track the number of cases in which there were only positive broadcasts during the process, the negatives track the number of cases in which there were only negative broadcasts during the process, failure 1 tracks the number of cases in which no broadcasts were made in the time span (which is 1000 times the number of nodes), failure 2 tracks the number of cases in which both negative and positive broadcasts were made, and average broadcast time tracks the average time till the first broadcast. We will be primarily examining two metrics during this process: accuracy and speed. Accuracy is determined by how many runs are successful (in positives and negatives) as opposed to the failure rows while speed is determined by average broadcast time.

3.1 Experiment 1

Our first experiment was a sanity check. The run had 100 nodes, connectivity centered at 15, and an average UNL size of 15. What we notice is that all 100 runs in each trial had only positive broadcasts, signifying a successful run. This was true for each of the three trails in this experiment (which were identical in terms of input). Another major finding was that there is some difference between the average broadcasts which we expected to be identical. This tells us that due to the randomness of the latencies each experiment, we should not consider minor variations in average broadcast time to be a major factor of comparison. Instead, we will only track general trends across average broadcast times.

Our first set of experiments will test the effect of both changes in UNL size and connectivity when we use a large number of nodes.

3.2 Experiment 2

We first test the impact of sweeping the UNL sizes from 5 to 90. Looking at the results table, we notice that there is surprisingly no major difference between each of the trials. All of the trials are roughly the same both in terms of average broadcast time as well as having a hundred out our a hundred successes. This highlights that at high number of nodes, the consensus protocol is robust to various UNL sizes.

3.3 Experiment 3

We now compare using a completely random UNL set for each node versus using a common UNL set. We determine this through the propRecommended variable which ranges from 0 to 1. 0 signifies a completely random UNL set while 1 signifies a completely common UNL set. Values in between signify what proportion of the UNL set will be from a common set of nodes while the rest are chosen randomly. Again, we see surprisingly little change across trials. With a relatively large number of nodes, all values of propRecommended perform similarly in accuracy and speed.

3.4 Experiment 4

Now that we compared the two methods to build UNLs, we shifted gears to examine the effect of changing connectivity (especially at low levels) by seeping the average connectivity size from 1 to 5. We see our first significant results here!

While all the trials succeeded in terms of accuracy (even for the trial with an average of 1 connection per node), a higher connectivity clearly denoted lower broadcast times. This can be seen by the steady and significant decrease of the broadcast times across trials. While having many nodes may overcome failure when we have very few edges, it does in fact, as we expect, lead to much slower speeds.

Now that we have examined the impact of these variables at a high number of nodes, we shift gears to examine the impact of the variables at a lower number of nodes.

3.5 Experiment 5

In this experiment, we continue the connectivity test, but this time with only 10 nodes. Sweeping connectivity averages of 1, 2, and 3, we see our first cases of accuracy errors. We can see that with only 10 nodes, we begin to have failure type 1, meaning that nodes do not broadcast in a reasonable amount of time. We see that as we increase connectivity, we increase both accuracy and average broadcast time. This is expected as paths to reach any broadcast may not be available.

3.6 Experiment 6

For this experiment, we set the average UNL size to $\frac{1}{3}$ of the number of nodes as we swept from 10 to 60 nodes. Our goal here is to see the impact of the number of nodes (given that the average UNL size is always a portion of the total number of nodes) on accuracy and speed. This experiment was done with very high connectivity to isolate variables. We see that in general, as we increase the number of nodes and proportionally the UNL size, we get steadily faster broadcast times. In addition, regarding accuracy, while the first trial has some failures, all other trails have full accuracy.

3.7 Experiment 7

This experiment is very similar to experiment 6, except we also set our connectivity average to be $\frac{1}{3}$ of the number of nodes. Doing the same sweep, we observe identical trends as experiment 6. Our one biggest difference is that we encounter more failures on trial 1 and in general speeds for each trial are slower, which is to be expected with lower connectivity.

3.8 Experiment 8

In this trial, we wanted to re-compare the impact of having a common UNL versus a random one with 10 nodes and low connectivity (average connectivity of 2). To do this, for every consecutive three trials, we kept the propRecommendation the same and increased the average UNL size from 3 to 5. For every trio of trials, we incremented propRecommendation to sweep from 0 to 1. If we look at the results, we see the trend we learned from our previous runs: increasing the average UNL size leads to better accuracy and speed. However, across propRecommendations, we see that while using more a common UNL does not decrease speed, it decreases accuracy steadily. This is probably beacuse the more there is a common UNL, the higher the chance that we get a "bad" starting UNL set that is impossible to lead to a final broadcast. Thus, when it is possible for the broadcast to be made, it makes it as fast as a random UNL would, but when it is impossible, it would fail.

For the remaining experiments, we swept over standard deviation parameters to examine the impact of servers choosing differently versus making similar decisions, especially given these decisions are "bad" decisions.

3.9 Experiment 9

We swept over UNL size standard deviation for 100 nodes with a UNL size of 20. Similarly to our early findings, when we have enough nodes and a high enough UNL size, all trials seem to perform successfully and have similar average broadcast times. Thus the variation between choices here that are all mostly "good" choices makes little impact.

We now explore some of these "bad" decisions

3.10 Experiment 10

We do the same experiment as 9, but we now have an average UNL size of 1. We see that with no standard deviation, we fail all 100 cases as it is not possible to reach a singular broadcast with only 1 item in the UNL. As we increase standard deviation, we begin to get more successful results meaning that our system is rather robust given even a small portion of the population makes "good" decisions of having larger UNL lists.

Table 1: Experiment 1: Basic sanity testing

Property	Trial 1	Trial 2	Trial 3
avgUNLSize	15	15	15
sdUNLSize	0	0	0
minUNLSize	1	1	1
avConSize	15	15	15
sdConSize	2	2	2
minConSize	3	3	3
numNodes	100	100	100
propRecommended	0	0	0
avgLatency	2	2	2
sdLatency	1	1	1
Positives	100	100	100
Negatives	0	0	0
Failure1	0	0	0
Failure2	0	0	0
AvgBroadcastTime	0.9334	0.8709	0.8046

Table 2: Experiment 2: UNL size sweep from 5-90 with 100 nodes.

Property	Trial 1	Trial 2	Trial 3	Trial 4	Trial 5	Trial 6
avgUNLSize	5	15	30	50	70	90
sdUNLSize	0	0	0	0	0	0
minUNLSize	1	1	1	1	1	1
avConSize	15	15	15	15	15	15
sdConSize	2	2	2	2	2	2
minConSize	3	3	3	3	3	3
numNodes	100	100	100	100	100	100
propRecommended	0	0	0	0	0	0
avgLatency	2	2	2	2	2	2
sdLatency	1	1	1	1	1	1
Positives	100	100	100	100	100	100
Negatives	0	0	0	0	0	0
Failure1	0	0	0	0	0	0
Failure2	0	0	0	0	0	0
AvgBroadcastTime	0.9516	0.8045	0.8845	0.9217	0.8125	0.8587

3.11 Experiment 11

Lastly, we sweep over latency standard deviation. Keeping 100 nodes, and 20 as our average UNL size and connectivity, we notice that while accuracy stays at 100, the average broadcast drastically reduces with greater standard deviation. This makes sense as even with a few fast edges, proposals can find a path to its destinations rather fast. Only if all paths are equally slow does average broadcast time slow down the most.

4 Conclusion

By running various experiments that tested the ripple consensus protecol's robustness to variations in the UNL choosing and underlying topography, we made many surprising findings. While, as expected, lower UNL size and connectivity led to decreased accuracy and speeds with a low number of nodes, having many nodes made the simulation robust against most changes in these parameters. The only parameter that really made an impact was the connectivity which determined speed. With enough nodes, which in our case seemed to be around 20 nodes, and a UNL and connectivity greater than 4, we seemed to get ideal performance. When the number of nodes were around 10 and the connectivity or UNL sizes were low, we ran into performance issues. Furthermore, our assumption that using a common UNL being better than using a random UNL was not entirely correct. There were cases, especially with a low number of nodes, in which using random UNLs proved to show better performance. Overall, we encountered many surprising results, but our simulation showed that with enough nodes, the ripple consensus protocol is robust.

Table 3: Experiment 3: Proportion of the UNL size that is from a common UNL set sweep from 0 to 1

Property	Trial 1	Trial 2	Trial 3	Trial 4	Trial 5	Trial 6
avgUNLSize	20	20	20	20	20	20
sdUNLSize	0	0	0	0	0	0
minUNLSize	1	1	1	1	1	1
avConSize	15	15	15	15	15	15
sdConSize	2	2	2	2	2	2
minConSize	3	3	3	3	3	3
numNodes	100	100	100	100	100	100
propRecommended	0	0.1	0.5	0.7	0.9	1
avgLatency	2	2	2	2	2	2
sdLatency	1	1	1	1	1	1
Positives	100	100	100	100	100	100
Negatives	0	0	0	0	0	0
Failure1	0	0	0	0	0	0
Failure2	0	0	0	0	0	0
AvgBroadcastTime	0.8673	0.9450	0.9334	0.8560	0.8843	0.8813

Table 4: Experiment 4: Average connectivity size at low levels (1-5 sweep)

Property	Trial 1	Trial 2	Trial 3	Trial 4
avgUNLSize	20	20	20	20
sdUNLSize	0	0	0	0
minUNLSize	1	1	1	1
avConSize	1	2	3	5
sdConSize	2	2	2	2
minConSize	1	1	3	3
numNodes	100	100	100	100
propRecommended	0	0	0	0
avgLatency	2	2	2	2
sdLatency	1	1	1	1
Positives	100	100	100	100
Negatives	0	0	0	0
Failure1	0	0	0	0
Failure2	0	0	0	0
AvgBroadcastTime	3.2899	2.9283	1.9025	1.5840

Table 5: Experiment 5: Connectivity test, but with low node size to see more significant impact.

Property	Trial 1	Trial 2	Trial 3
avgUNLSize	3	3	3
sdUNLSize	0	0	0
minUNLSize	1	1	1
avConSize	1	2	3
sdConSize	2	2	2
minConSize	1	1	3
numNodes	10	10	10
propRecommended	0	0	0
avgLatency	2	2	2
sdLatency	1	1	1
Positives	75	84	93
Negatives	0	0	0
Failure1	25	16	7
Failure2	0	0	0
AvgBroadcastTime	6.6470	6.3079	5.3582

Table 6: Experiment 6: Average UNL size = $\frac{1}{3}$ of num nodes. Num nodes swept from 10-60 by tens. High connectivity.

Property	Trial 1	Trial 2	Trial 3	Trial 4	Trial 5	Trial 6
avgUNLSize	3	6	9	12	15	18
sdUNLSize	0	0	0	0	0	0
minUNLSize	1	1	1	1	1	1
avConSize	10	20	30	40	50	60
sdConSize	2	2	2	2	2	2
minConSize	1	1	1	1	1	1
numNodes	10	20	30	40	50	60
propRecommended	0	0	0	0	0	0
avgLatency	2	2	2	2	2	2
sdLatency	1	1	1	1	1	1
Positives	95	100	100	100	100	100
Negatives	0	0	0	0	0	0
Failure1	5	0	0	0	0	0
Failure2	0	0	0	0	0	0
AvgBroadcastTime	2.8306	0.7607	0.5412	0.3967	0.3581	0.2545

Table 7: Experiment 7: Connectivity = Average UNL size = $\frac{1}{3}$ of num nodes. Num nodes swept from 10-60 by tens.

Property	Trial 1	Trial 2	Trial 3	Trial 4	Trial 5	Trial 6
avgUNLSize	3	6	9	12	15	18
sdUNLSize	0	0	0	0	0	0
minUNLSize	1	1	1	1	1	1
avConSize	3	6	9	12	15	18
sdConSize	2	2	2	2	2	2
minConSize	1	1	1	1	1	1
numNodes	10	20	30	40	50	60
propRecommended	0	0	0	0	0	0
avgLatency	2	2	2	2	2	2
sdLatency	1	1	1	1	1	1
Positives	93	100	100	100	100	100
Negatives	0	0	0	0	0	0
Failure1	7	0	0	0	0	0
Failure2	0	0	0	0	0	0
AvgBroadcastTime	5.0654	2.0701	1.4569	1.0653	1.0571	0.8146

Table 8: Experiment 8: Two sweeps. First swept over propRecommended from 0-1. Then for each one swept over avgUNLSize 3-5

Property	Trial 1	Trial 2	Trial 3	Trial 4	Trial 5	Trial 6	Trial 7	Trial 8	Trial 9	Trial 10	Trial 11	Trial 12	Trial 13	Trial 14	Trial 15
avgUNLSize	3	4	5	3	4	5	3	4	5	3	4	5	3	4	5
sdUNLSize	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
minUNLSize	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
avConSize	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2
sdConSize	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2
minConSize	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
numNodes	10	10	10	10	10	10	10	10	10	10	10	10	10	10	10
propRecommended	0	0	0	0.2	0.2	0.2	0.5	0.5	0.5	0.7	0.7	0.7	1	1	1
avgLatency	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2
sdLatency	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
Positives	86	95	100	86	94	99	83	92	99	83	93	99	87	92	97
Negatives	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Failure1	14	5	0	14	6	1	17	8	1	17	7	1	13	8	3
Failure2	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
AvgBroadcastTime	5.9324	5.1694	3.4178	5.9979	5.4618	3.3634	6.1847	5.2158	3.6443	5.8168	5.8048	3.5961	5.5693	5.4459	3.3510

Table 9: Experiment 9: Sweep over sdUNLSize from 0,4,7

Property	Trial 1	Trial 2	Trial 3
avgUNLSize	20	20	20
sdUNLSize	0	4	7
minUNLSize	1	1	1
avConSize	2	2	2
sdConSize	2	2	2
minConSize	1	1	1
numNodes	100	100	100
propRecommended	0	0	0
avgLatency	2	2	2
sdLatency	1	1	1
Positives	100	100	100
Negatives	0	0	0
Failure1	0	0	0
Failure2	0	0	0
AvgBroadcastTime	2.8692	2.9462	2.9685

Table 10: Experiment 10: Same as experiment 9, but lower average UNL size

Property	Trial 1	Trial 2	Trial 3
avgUNLSize	1	1	1
sdUNLSize	0	4	7
minUNLSize	1	1	1
avConSize	1	1	1
sdConSize	2	2	2
minConSize	1	1	1
numNodes	100	100	100
propRecommended	0	0	0
avgLatency	2	2	2
sdLatency	1	1	1
Positives	0	99	100
Negatives	0	0	0
Failure1	100	1	0
Failure2	0	0	0
AvgBroadcastTime	0	3.7735	3.5133

Table 11: Experiment 11: Latency standard deviation (from 0 to 4 to 7)

Property	Trial 1	Trial 2	Trial 3
avgUNLSize	20	20	20
sdUNLSize	0	0	0
minUNLSize	1	1	1
avConSize	20	20	20
sdConSize	2	2	2
minConSize	1	1	1
numNodes	100	100	100
propRecommended	0	0	0
avgLatency	10	10	10
sdLatency	0	4	7
Positives	100	100	100
Negatives	0	0	0
Failure1	0	0	0
Failure2	0	0	0
AvgBroadcastTime	20.0	5.7362	1.7276

However, there are tests that we still hope to conduct. An unused environment variable in our code was malicious nodes which would be nodes that would vote against the consensus on purpose. Our next steps could be to test the functionality for malicious nodes and test the system's robustness against one or multiple nodes working together to undermine the network. While the functionality is already implemented in the code, we decided not to focus on malicious node testing for this project.

Another direction to explore is to create a feature to customize the connectivity of each graph to test edge cases that may occur. We were surprised that we did not get many type 2 failures which was probably due to the robustness of the protocol against random connectivity loss, so to catch such errors, we would probably need more contrived or hard coded graphs. Thus, we could manually test graphs such as graphs that have separate connected clusters and have low connectivity to each other or graphs with entire clusters of malicious nodes. Furthermore, we could implement features to allow users to manually customize the graph's connectivity.