CSE535 - Asynchronous Systems

Course Project Final Report

Algorand Byzantine Agreement in DistAlgo

Group 7

Sai Parthasarathy Miduthuri
Chetan Kasireddy
Zenab Bhinderwala

Table of Contents

1. Project Proposal and Plan

Introduction

The Algorand cryptocurrency paper[1] proposes a method of blockchain consensus that is based on integrating the security assured by cryptographic hash protocols, namely, *Verifiable Random Functions*, with the *proof-of-stake* based blockchain consensus problem, for *Byzantine Fault Tolerance* in distributed systems.

The paper proposes a mechanism to use cryptographic security to limit the agency of traitor processes in the distributed system and suggests a new solution that incorporates this security in arriving at a consensus upon proposed blocks to be added to a blockchain.

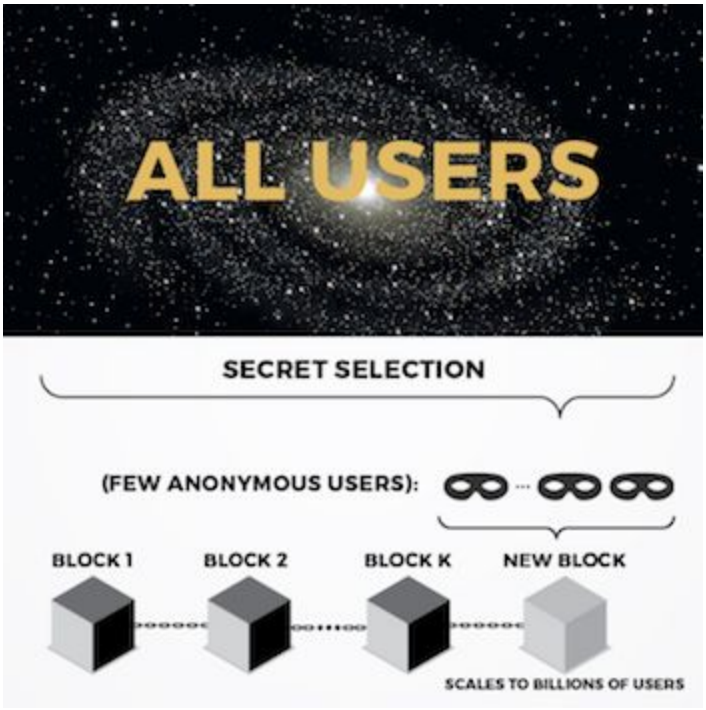


Figure 1: A random self-selected small committee is selected via cryptographic sortition[2]

Problem Description

Our project proposal is the implementation of the Byzantine Agreement algorithm $BA*$ in Algorand, with the cryptographic functions abstracted away. Thus, the implementation will only involve the process of consensus between a randomly chosen committee, in order to accept (or tentatively accept) a proposed block, as per the algorithm in the paper. We intend to implement this algorithm in DistAlgo.

Inputs

1. The existing context of the blockchain (the chain itself, the current round of consensus, etc.)
2. The number of users to run the algorithm for.
3. Length of blockchain to be simulated.
4. The number of users that are traitors in the system.
5. Stakes of all the traitors combined.
6. Timeouts for receiving all proposals and timeouts to complete every step during BA*.
7. The number of committee members at every step in BA*.

Outputs

A Blockchain ending in a Finalized or Tentative Block, upon which consensus has been reached, or ending in a fork in the chain (The fork will be resolved by performing BA* upon all of the blocks resulting from the round at which the fork took place, and is not a part of the implementation of consensus, here.)

Performance and Correctness Metrics

1. Validity - The Block upon which consensus is reached is one of the blocks that has been proposed
2. Agreement - Safety, i.e., all users in the system agree upon the same Block
3. Termination - Liveness, i.e., the system is actually able to come to a consensus in "strong synchrony" conditions
4. Assumptions:
    a. The fraction of money held by honest users is above some threshold h (a constant greater than 2/3), but that an adversary can participate in Algorand and own some money.
    b. In order to successfully attack Algorand, the attacker must invest substantial financial resources in it, i.e., the attacker must have a stake > 1/3.
    c. The attacker can control up to 1/3 of the users.
    d. Committee Member nodes which are attacked are capable of sending their votes before they are attacked with denial of service
5. Performance Evaluation of time taken for consensus to be reached

Applications

This consensus algorithm is used in the Algorand blockchain cryptocurrency model.

Apart from the Algorand model, this consensus algorithm can also be used for proof-of-stake based Byzantine Consensus in distributed systems with large numbers of participating processes, where it is possible to randomly select proposers and acceptors such that an adversary cannot easily predict them, or if the adversary is capable of attacking the acceptors in the consensus step, once they are made known in any way.

State of the art existing implementations

One algorand implementation[3] was found on GitHub, which was implemented in Rust language. The Readme present on GitHub does not give an overview of the implementation, and the lack of comments and documentation makes it hard to understand. Examining the code did not give any useful insights, either

Weekly plan and Project subtasks

	Sai Parthasarathy	**Chetan Kasireddy**	**Zenab Bhinderwala**
Week 1	Abstract out cryptographic fns *Sortition* and *VerifySort*	Understand in-depth flow of the algorithm	Figure out cryptographic hash function *H* - its usage and replacement
	Implement Gossip communication of messages between nodes, with a random selection of recipients of gossip.	Understand and plan the coding of specifics in the random selection of committee members in each step of BA*.	Creating a Controller to manipulate the nodes and introduce Byzantine Failures and to abstract out cryptographic sections in BA* implementation.
Week 2	*Implementing CommiteeVotes, Reduction*	*Implementing CountVote, BinaryBA*	*Implementing ProcessMsg, Sortition, VerifySort.*
Week 3	Debug any issues that arise from merging the work done in Week 2. Verify Correctness of implementation. Performance Evaluation for different parameters		
Week 4	Make Visualization For Test Results, Code Documentation Project Report, Presentation, and Project Demo		

2. Project Design Plan

Design Documentation

The Design Documentation for the source code was generated using the epydoc tool, and is hosted at the following web address:

https://kchetan.github.io/Algorand-Documentation/

Design Summary

A Blockchain consists of a series of Blocks, each of which contains pointers to the previous block, all the way from the latest block to the first.

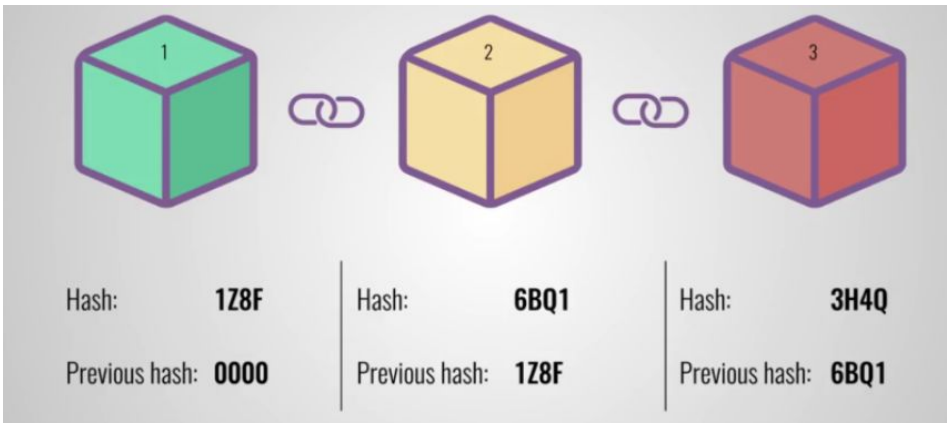


Figure 2: A simple illustration of the concept of Blockchain[4]

Blocks are appended to the end of the chain, such that each block contains the hash of the last block on the chain. However, a block can only be allowed to join the chain after a consensus among the users of the Blockchain system. In Algorand, randomly selected committees of some particular size, independent of the actual number of users on the entire system, will perform the function of coming to a consensus upon any proposed block.

Algorand contains multiple security protocols and redundancies that preserve the safety and liveness of the consensus process, through the use of cryptography. These cryptographic functions have been abstracted by replacing them with a Controller process can coordinate the consensus process instead of the cryptographic functions.

The design consists of a Sortition function, which decides who among the users will play the role of Proposer and Committee Member, a Reduction function, which forces all users to choose two blocks that have been proposed, for consensus.

The BinaryBA* function then runs on all users, until all committee members reach consensus. These committee members are randomly changed at every step of the consensus

algorithm, in order to prevent an adversary from gaining an advantage by taking over any known committee member.

The design consists of:

1. A Context class in the src/context.da file, which contains global information about the state of the entire system such as the last block on the chain, the chain itself, functions to print the chain, etc.
2. A Controller class in the src/controller.da file, which manages the assignment of users to various roles, and acts as the generator of results of the consensus.
3. A User class in the src/algorand.da, which forms the template for all the user processes in the system.
4. A Monitor class in the src/Monitor.da file, which contains the code to perform correctness and performance tests upon the implementation, for a given set of parameters.
5. A Block class in the src/block.da file, which contains the description of the contents of each block that has been proposed (or maliciously added)
6. A hash function H in the src/H.da file, which converts an input into a pickle byte array, hashes this byte array using SHA256 and returns the resultant 256-byte hash.
7. A Parameters class in the src/parameters.py file, which contains the list of pre defined parameters to be easily modified and passed around between processes to make the setup process much easier.

The source code is stored in the src/ directory.

The code can be run by running the file src/algorand.da, with parameters specified as needed.

The code can be tested for correctness and performance characteristics by running the file src/Monitor.da, with parameters given in the Monitor.da as needed to run the tests, as described in Part 4.

The following illustration shows the flow of Byzantine Consensus in Algorand.

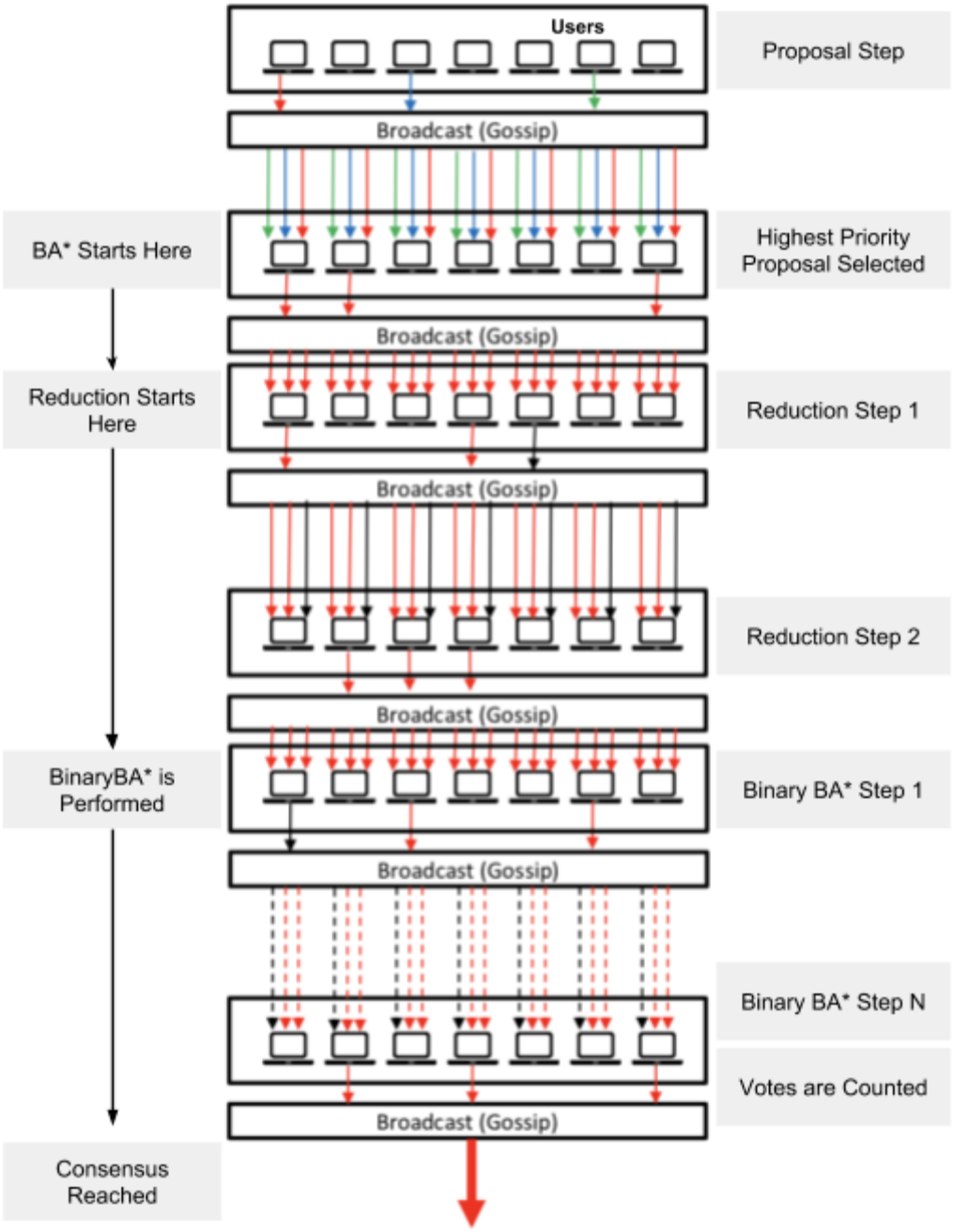


Figure 3: Illustration of Byzantine Agreement in Algorand.[5]

3. Implementation

Implementation Source Code

The following link contains the source code as implemented now:

https://github.com/unicomputing/algorand-ba-distalgo

Implementation Summary

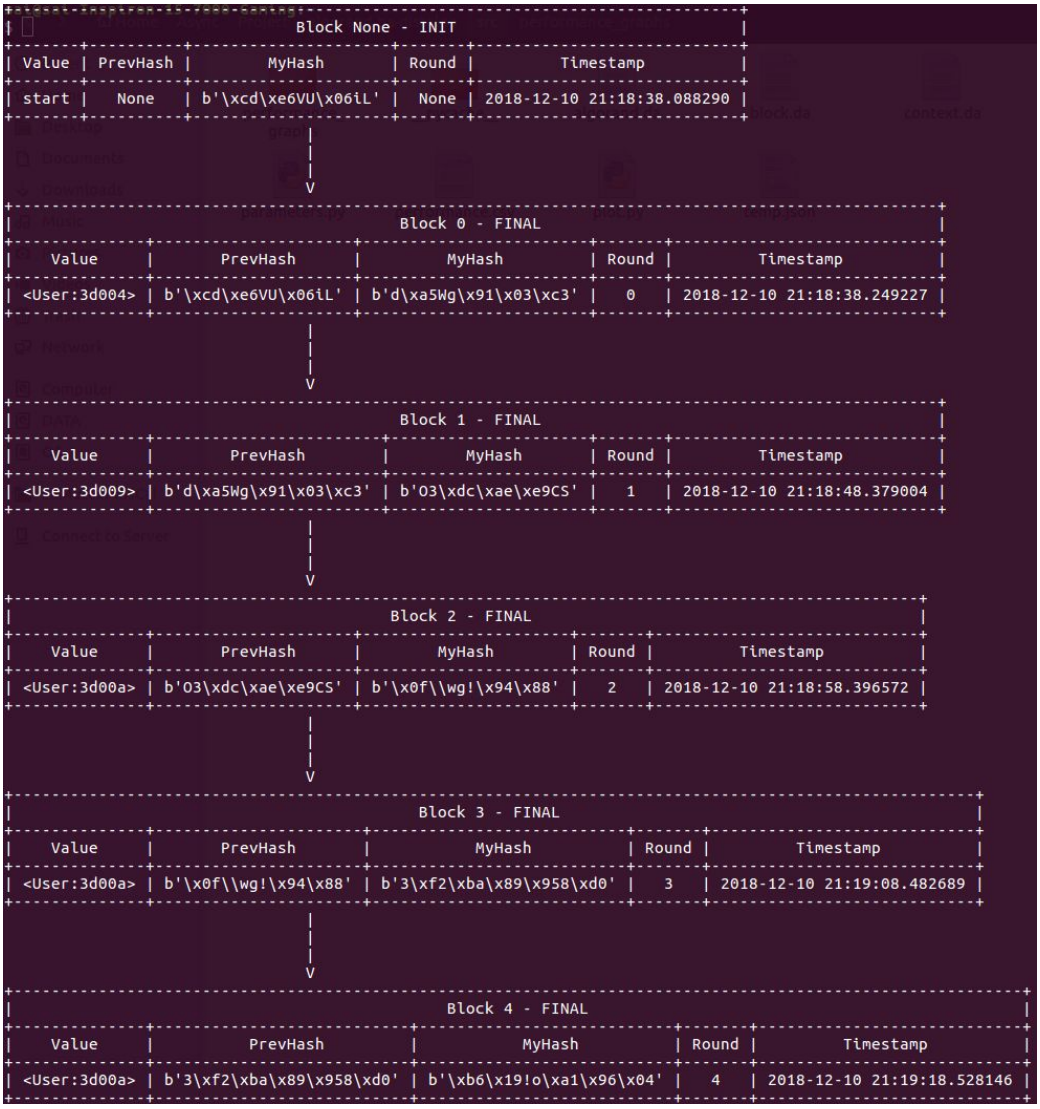


Figure 4: Image of a Blockchain produced by our implementation

For the DistAlgo implementation of Algorand we started by going through the BA* pseudo code specified in the Algorand paper[1]. The following functions from the paper were translated into DistAlgo:

- CountVotes - Algorithm 5: Counting votes for round and step, Pg 9 in Algorand paper[1]
- ProcessMessage - Algorithm 6: Validating incoming vote message m, Pg 9 in Algorand paper[1]
- Reduction - Algorithm 7: The two-step reduction, Pg 9 in Algorand paper[1]
- Binary BA* - Algorithm 8: Binary BA* executes until the consensus is reached on either block_hash or empty_hash, Pg 10 in Algorand paper[1]

DistAlgo with its comprehensive functions for distributed consensus made it easier to translate the above pseudo codes line by line into functions in .da files.

Development Efforts

Tasks	**Total Hours Spent (Hours)**
Project Plan and Proposal	20
Design	20
Core Implementation	60
Correctness and Performance Testing	20
Bug Fixing	15

Code Sizes

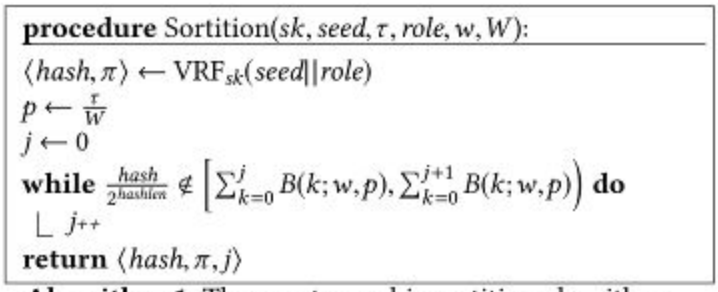
File	**Blank**	**Comment**	**Code**	**Total**
algorand.da	110	214	364	688
block.da	11	43	40	94
context.da	12	29	56	97
controller.da	40	45	215	300
H.da	4	9	6	19
Monitor.da	23	0	117	140
parameters.py	1	16	12	29
Total	201	356	810	1367

Implementation Details

Apart from the pseudocode specification as given in the paper, we have also implemented functionalities to abstract Cryptography, monitor/control the working of all users in the system, continuously check the correctness parameters at every round, and keep track of the more mundane details of a particular run, such as which of the users are traitors, and how much stake each of them own, etc. We have also written additional code to define the ways in which a user must fail, whether innocently or maliciously.

Controller to abstract VRF-based Cryptographic Sortition

The paper requires the use of Verifiable Random Functions to generate "hashes" and "proofs" for each committee member selected in a step, and each proposer selected in a round, as visible in the pseudocode below taken from Pg 5 of the paper[1].



Algorithm 1: The cryptographic sortition algorithm.

This VRF function takes the private key for the user running it as a parameter, and a concatenation of the random seed and the role for which Sortition is being performed as input, and generates a "sorthash - *hash*" and a "proof π". These two values, sent along with the public key for that user, are meant to unequivocally identify a user who sent a message as being a committee member or a proposer.

We have replaced this with an abstraction in the Controller which randomly selects a user for that role according to their stake in that round. The PK-SK pairs have been removed, along with the sorthash and proof. Instead, each original sender of Gossip sends their own process ID in the message, and we assume that this cannot be tampered with.

Whenever a User performs Sortition or VerifySort, it requests the Controller for its role in that step. The Controller notifies each User about its role as a Proposer or Committee Member, which then triggers the User to take the appropriate action per its role.

In the original algorithm, tampering with any part of a message signed with a private key will cause public key decryption to fail, hence causing the message to be rejected. If the objective of a traitor is to cause a forwarded message to be rejected, they might as well not send it at all.

Controller to abstract VRF-based Sortition Verification

The paper uses VRFs to verify the origin of a message sent to any user, through the use of the "sorthash" and "proof" values, as visible in the pseudocode below taken from Pg 6 of the paper[1].

procedure VerifySort($pk, hash, \pi, seed, \tau, role, w, W$):
if $\neg VerifyVRF_{pk}(hash, \pi, seed||role)$ **then return** 0;
$p \leftarrow \frac{\tau}{W}$
$j \leftarrow 0$
while $\frac{hash}{2^{hashlen}} \notin \left[\sum_{k=0}^{j} B(k;w,p), \sum_{k=0}^{j+1} B(k;w,p)\right)$ **do**
  j++
return j

Algorithm 2: Pseudocode for verifying sortition of a user with public key *pk*.

This function also returns the number of votes that a committee member can cast "*j*", which is returned as 0, means that the values do not correspond to a committee member.

We have abstracted this by performing a message exchange between the User and the Controller, where the Controller holds all the information about the role of each User at each round/step.

Gossip Neighbor Assignment For Users

The paper extensively relies on Gossip for communication between users. This is a reliable mode of communication in a real-world scenario, where users can enter and leave the network at will, and not all users are connected to all others. However, not many details have been given in the paper as to the exact implementation of Gossip used.

We did not want to focus on adapting existing Python implementations of Gossip to DistAlgo, and we did not find a standard implementation of Gossip in DistAlgo (other than the project in development being worked on by Group 29). The majority of our development effort was spent in adapting BA* to DistAlgo, and hence we have implemented an extremely simplified version of Gossip.

Neighbours for each user are chosen before user setup is called, using a function named create_gossip_neighbors, in algorand.da. This function generates a random integer, *n_n*, between 2 to the total number of user processes, *n_u*, each time it is called to assign a user its neighbours. This number is the number of neighbours that user can have.

Now, the smallest number that is co-prime to the total number of users, *n_cp*, is found using the function find_smallest_coprime. The first process that is the neighbour of a user is itself. Starting from its own index in the list of all users, the index is incremented by *n_cp* modulo *n*. The fact that *n_cp* is coprime w.r.t. *n* means that eventually all indices in the users' list will be visited.

Through this, we can ensure that each user is connected to at least one other user, and is going to receive any Gossip eventually (assuming no message loss).

Controller and User to generate Byzantine Proposal Failures

The Proposals generated in any round can come from either a normal user or a malicious user. The paper suggests some possible methods in which a malicious proposer can try to prevent consensus in Pg 7[1]. We have thus implemented this in our Byzantine behaviour for Proposers.

The Proposal process is implemented such that the Controller notifies each Proposer of its role as a Proposer for that round. Now, if a malicious user receives the role of Proposer, it generates a normal Block, and then modifies the value (a string) in two different ways through the ByzantineGossip function. These are then Gossiped to all Users in its neighbourhood.

Thus, some users receive one of the proposed blocks, some receive the other, and some receive both. If the wait time for all users to receive proposals is high enough, then every user will receive all blocks. In this case, if this proposed block has the highest-priority, then all users will start with an Empty Block, as specified in Pg 16, section 10.4 of the paper[1].

However, the highest priority proposer can also always be forced to be a Byzantine Proposer, regardless of whether it is malicious, for testing purposes, through an input parameter in our implementation as per Pg 16, section 10.4[1].

Controller and User to generate Byzantine Committee Vote Failures

A traitor can be chosen as a committee member in any step, and probably will be, based on the amount of stake they hold in the network.

We have implemented ByzantineGossip for CommitteeVote such that a malicious committee member will create a fake block and block hash (it isn't necessary to have a corresponding block, but we wanted to see the fake block in the results, if it comes up), and vote for that fake hash, or will vote for an empty block.

User message loss

We have implemented message loss, but as a part of the receive function for a Proposal or CommitteeGossip. This is so that the Gossip protocol can actually cause some nodes to never receive some messages, and simulate a network loss. If this were a part of send, then a check of the sent-box for a user will let it know that it never sent the Gossip to some user (which is something that we want to happen as an error, but the algorithm wants to prevent). Knowing this is important to prevent repeated sends of the same message to the same user, as a feature.

Unfixed Bugs

There is a bug that occurs in controller.da, at lines 215-220 of the code, where the user needs to wait for the lambda_block amount of time after notifying the proposers and before starting BA* in all the users. This bug causes the “await-timeout” to never properly wait till timeout, even though the parameter that “await” waits to be “set to True” is just boolean False. Hence, it instantaneously “jumps” from line 215 to line 220 of the code, in execution.

This bug is triggered when the previous round passes an Empty Block. We have consulted the Professor regarding this issue, but haven’t been able to reach a fix.

This bug can only be triggered when using pyDistAlgo==1.1.0b12 or 1.1.0b13, and does not exist in pyDistAlgo==1.0.12.

4. Testing and Evaluation

System Configuration / Environment

RAM: 16 GB
Processor: Intel(R) Core(TM) i7-7700HQ CPU@2.80GHz, 4 Core(s), 8 Logical Processor(s)
Operating System: Ubuntu 16.04 (64 Bit)

Correctness Results

The following Correctness Metrics were tested for:

1. **Agreement** is satisfied if BA* in all the users returns the same block as tentative or final consensus.
2. **Validity** is satisfied if the agreed block is either a proposed block or an empty block.
3. **Liveness** is satisfied if the BinaryBA* returns a block_hash before maximum steps are reached.

The following results were obtained after running the following command and all the test cases are configured in Monitor.da:

```
python3 -m da Monitor.da
```

The results are all stored in correctness.csv and segregated as below.

Test Cases

The Monitor runs the tests for combination of following set of parameters

- No.of Users range 20-50
- No.of traitors range 0-17
- Traitor stakes range 0-0.36
- No.of committee members in step range from 5-20
- Block size range from approximately 0-3Kb
- Message Loss Probability in range 0-1 in increments of 0.1

Varying number of users

The following table shows the correctness results of running Monitor.da with 20-50 users with a step 0f 6 and keeping other parameters constant.

Users	Traitors	Stake traitors	Tau_step (No. of Committee members)	Consensus	Agreement	Validity	Liveness
20	5	0.33	5	Final	✓	✓	✓
26	5	0.33	5	Final	✓	✓	✓
32	5	0.33	5	Final	✓	✓	✓
38	5	0.33	5	Final	✓	✓	✓
44	5	0.33	5	Final	✓	✓	✓
50	5	0.33	5	Final	✓	✓	✓

Varying number of traitors

The following table shows the correctness results of running Monitor.da with 0-15 traitors keeping other parameters constant.

Users	Traitors	Stake Traitors	Tau_step	Consensus	Agreement	Validity	Liveness
30	0	0.33	5	Final	✓	✓	✓
30	3	0.33	5	Final	✓	✓	✓
30	7	0.33	5	Final	✓	✓	✓
30	11	0.33	5	Final	✓	✓	✓
30	15	0.33	5	Final	✓	✓	✓

Varying stake of traitors

The following table shows the correctness results of running Monitor.da with traitors stake from 0-0.36 keeping other parameters constant.

Users	Traitors	Stake Traitors	Tau_step	Consensus	Agreement	Validity	Liveness
30	5	0	5	Final	✓	✓	✓
30	5	0.072	5	Final	✓	✓	✓
30	5	0.144	5	Final	✓	✓	✓
30	5	0.216	5	Final	✓	✓	✓
30	5	0.288	5	Final	✓	✓	✓
30	5	0.36	5	Tentative	✓	✓	✓

Varying No.of committee members

The following table shows the correctness results of running Monitor.da with 5-15 committee members keeping other parameters constant.

Users	Traitors	Stake Traitors	Tau_step	Consensus	Agreement	Validity	Liveness
30	5	0.33	5	Final	✓	✓	✓
30	5	0.33	7	Final	✓	✓	✓
30	5	0.33	10	Final	✓	✓	✓
30	5	0.33	12	Final	✓	✓	✓
30	5	0.33	15	Final	✓	✓	✓

Thus from the above results, we conclude that Correctness is always satisfied in every round of Algorand as all the users reach consensus on a Final block, except when the traitors hold stakes more than 0.33 in which case the Correctness is still held but with a Tentative block. This is expected behaviour as the Algorand assumption is that traitors do not hold more than 1/3rd of the total stakes.

Performance Results

The following results were obtained by running the following command and all the test cases are configured in Monitor.da:

python3 -m da Monitor.da

The graphs were then obtained by running in the folder where the performance.csv is:

python plot.py

Varying number of users

From the below graphs we see that the Latency is almost constant when the number of users increases.

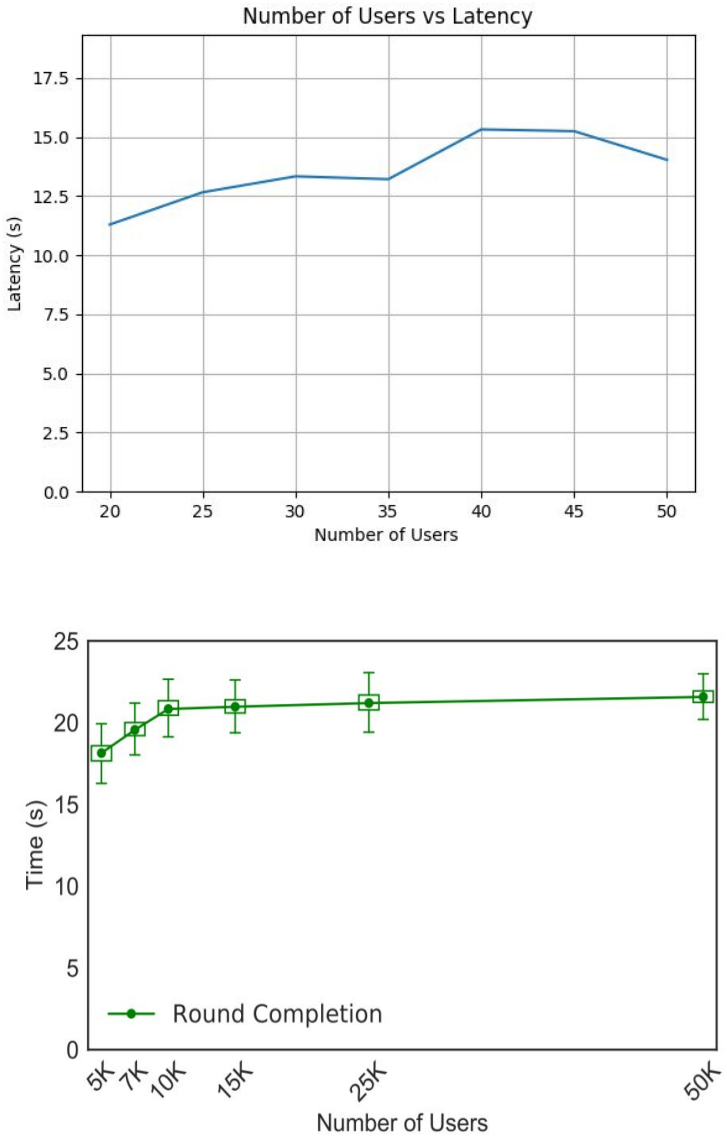


Figure 5: Comparison of Latency (secs) vs Number of Users (Top-Our results, Below-From paper)

Varying Number of Malicious Users

The graphs below indicate that Latency remains almost constant and within a very specific range when number of malicious users increase.

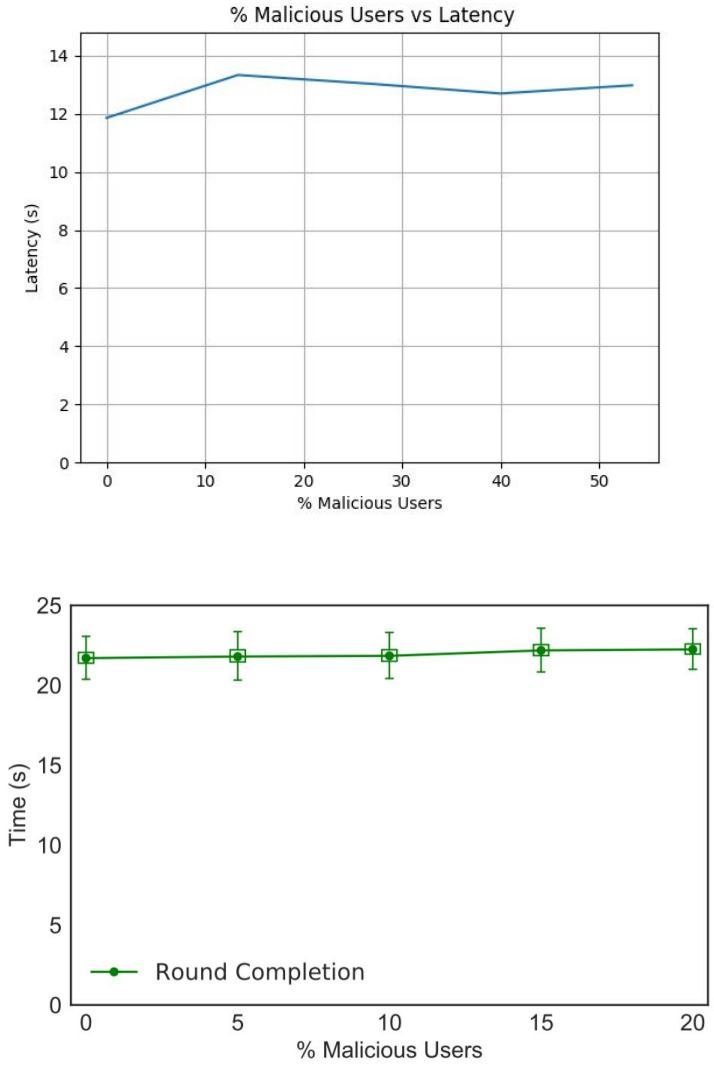


Figure 6: Comparison of Latency (secs) vs % Malicious Users (Top-Our results, Below-From paper)

Varying % stake of Malicious Users

This analysis was not done in the paper[1] and we thought the below results were very interesting as they go on to validate the assumption that malicious users do not hold more than ⅓ stakes i.e more that 33% as we see as the stakes reach 30% the latency steeply increases.

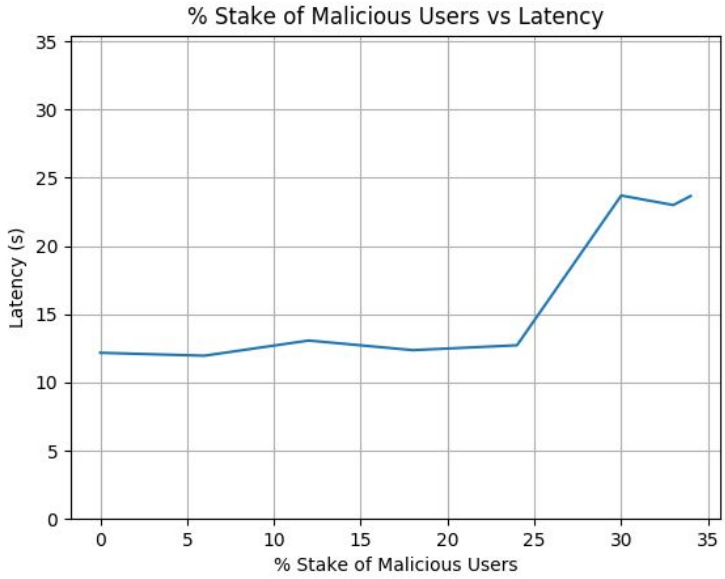


Figure 7: Latency in secs when % Stake of Malicious Users increases

Varying No.of Committee members

The below tests was also not performed in the paper[1]. The latency slowly increases as the number of users chosen to be in the committee increases.

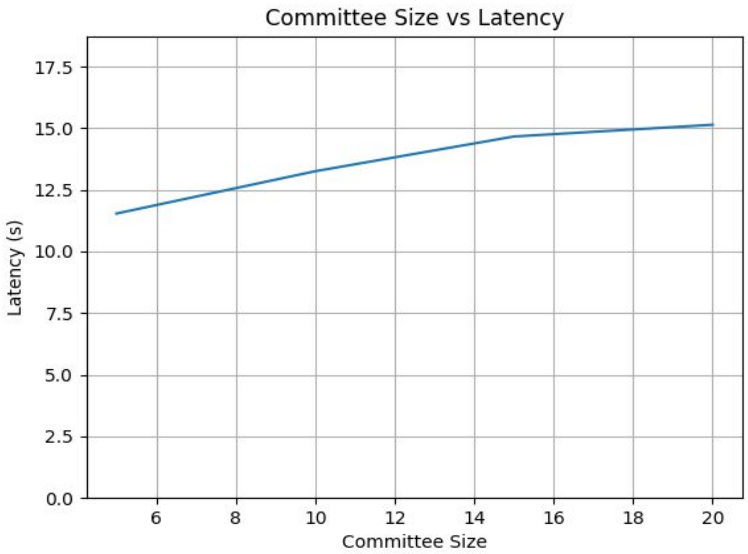


Figure 8: Latency in secs when Committee Size increases

Varying block size

Again there is not much effect on the time taken for BA* to complete when the block size is increased from 0-3KB. It was not possible to test for larger block sizes than 3KB due to the limitations of DistAlgo. As seen in the Block size v/s Latency graph from the paper[1] , the Latency almost remains constant till block size of 4MB and then only the proposal time increases but the final agreement i.e BA* round still takes constant time.

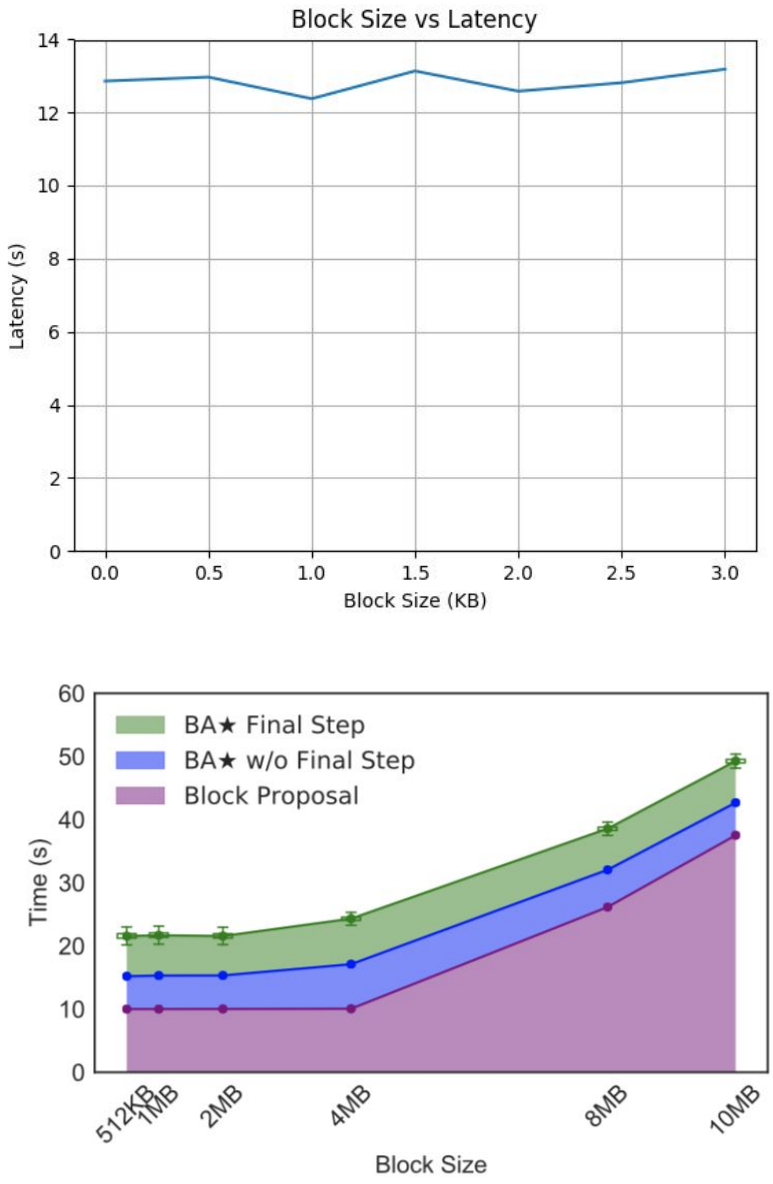


Figure 9: Comparison of Latency to **reach Final Step** vs Block Size (Top-Our results, Below-From paper)

Varying Probability of Message Loss for Gossip

There is a very small effect on latency for an increasing message loss probability until about 60% probability of network loss for Gossip. We also start seeing more forks around 0.8 probability, and the consensus reached is upon empty blocks (Tentative), rather than any proposed block (Tentative or Final).

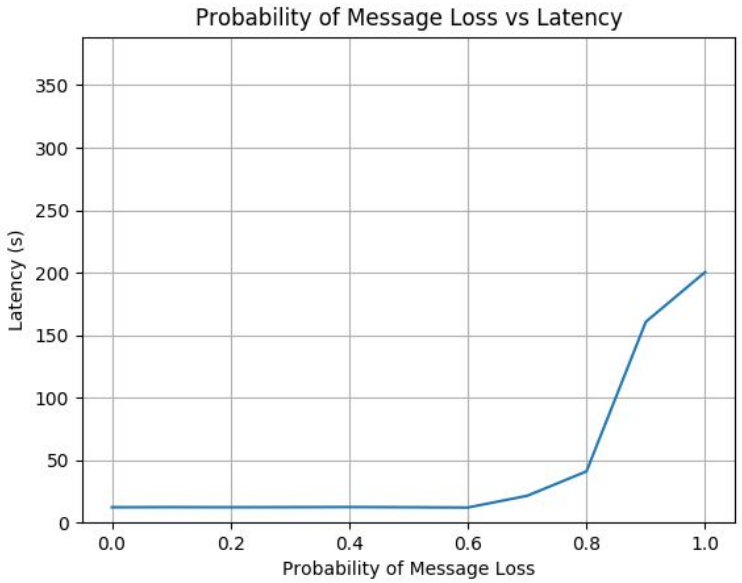


Figure 10: Latency in secs when probability of message loss increases

5. Conclusions

The results of Correctness Tests show that the correctness of this implementation of Algorand's BA* is preserved in all reasonable modes of operation where Byzantine faults are tolerated until at most 1/3 of the stake is held by traitors. While consensus is still "possible" beyond this, it is not guaranteed.

The results of Performance Tests show that our implementation's latency w.r.t. various parameters follows the trends in the paper, to the extent that was tested. In addition to the parameters tested in the paper, we have provided performance tests upon other parameters, and have shown the robustness of our implementation.

Our implementation was limited by the fact that it was only possible to create up to 80 nodes (with increasing lambda_block and lambda_step requirements) before running into Errors due to message sizes (for the setup segment of the code), and that it was only possible to create blocks of size up to 3 KB to be sent in Proposals, once again due to Errors caused for larger message sizes in DistAlgo.

6. Future Work and Further improvements

Future Work

Our implementation focussed on reaching consensus on a block at every round of Algorand with Byzantine fault tolerance. Thus future work would include reaching consensus on multiple branches of Algorand after a fork has occurred with a BA* being run on the multiple branches to merge the fork.

Also with the limited server capacity of this project, future work could also include testing on multiple servers with up to 50k users.

Further Improvements

Improvements can be done in sending Gossip messages, where the current implementation can be replaced by a more robust and realistic Gossip implementation.

When DistAlgo is capable of handling block size greater than 3KB, performance testing of block sizes upto 1MB needs to be performed.

The synchronization of all users for each round is being done by forcing all of the users to wait until the controller receives the results at the end of the round, and until the controller tells all the users to begin. The paper suggests that loosely synchronized clocks be used across all the users in case of weak-synchrony (e.g., NTP), for which no implementation details have been given in the paper.

7. References

[1] "Algorand: Scaling Byzantine Agreements for Cryptocurrencies - People - MIT." https://people.csail.mit.edu/nickolai/papers/gilad-algorand-eprint.pdf. Accessed 9 Dec. 2018.
[2] "Algorand Releases First Open-Source Code of Verifiable Random Function." 9 Oct. 2018, https://medium.com/algorand/algorand-releases-first-open-source-code-of-verifiable-random-function-93c2960abd61. Accessed 9 Dec. 2018.
[3] “GitHub - peitalin/algorand-rust: demo implementation." https://github.com/peitalin/algorand-rust. Accessed 9 Dec. 2018.
[4] "Blockchain in Healthcare: Both sides of the coin | Savvycom Insight." 27 Jul. 2018, https://savvycomsoftware.com/blockchain-in-healthcare-both-sides-of-the-coin/. Accessed 9 Dec. 2018.
[5] "GitHub - j1schmid/algorand: Algorand / Kryptowährung." https://github.com/j1schmid/algorand