Using Weighted Voting to Accelerate Blockchain Consensus

Intrusion tolerance during performance degradation attacks

Artur Brodovic
Supervisor(s): Jeremie Decouchant, Rowdy Chotkan
EEMCS, Delft University of Technology, The Netherlands

A Thesis Submitted to EEMCS Faculty Delft University of Technology,
In Partial Fulfilment of the Requirements
For the Bachelor of Computer Science and Engineering
June 23, 2024

Name of the student: Artur Brodovic
Final project course: CSE3000 Research Project
Thesis committee: Jeremie Decouchant, Rowdy Chotkan, Kaitai Liang

Abstract

This paper examines the impact of faulty nodes on Practical Byzantine Fault Tolerance (PBFT) algorithms, focusing on the AWARE optimization. While AWARE improves average-case latency by assigning larger voting weights to well-connected nodes, it is vulnerable to exploitation by Byzantine nodes. We propose modifications to AWARE to enhance resilience of dynamic link latency estimation. Experiments show that comparing predicted and actual consensus latencies helps detect and mitigate performance issues caused by malicious nodes. Integrating latency information with BFT-SMaRt's leader change algorithm offers a more robust solution. We also analyse shortcomings of AWARE dynamic leader selection system.

1 Introduction

Blockchain consensus algorithms make it possible to execute a state machine in a distributed system where some nodes might be faulty. This is called distributed execution. However, distributed execution requires a lot of information exchange between nodes. In many systems deployed in the real world, latency is limited, so communication takes up more time compared to other steps. One attempt to solve this issue was made by introducing WHEAT[8] optimization. It allows achieving subsecond latency improvement in a geo-diverse setting, by assigning larger voting weight to a better connected nodes. This approach was automated with AWARE[2] algorithm. It introduces extra reconfiguration step to a BFT system that periodically sends latency measurement messages and based on the results reassigns voting weights.

Experiments with these optimizations focus on acceleration of BFT distributed algorithms in an average case, when high voting weights are obtained by correct nodes. However, when analysing worst case researchers reach only theoretical conclusions, by showing that algorithms still satisfy all properties that BFT replication algorithm must have: safety, availability and consistency. This approach follows Lampson's recommendation for computer systems: "Handle normal and worst case separately as a rule because the requirements for the two are quite different. The normal case must be fast. The worst case must make some progress" [7].

Goal of our research it to take more critical look at the worst case performance. We are going to analyse how faulty nodes can leverage weighted voting to degrade BFT algorithm performance. Also, we will provide a modification of AWARE link latency estimation that mitigates some possible attacks.

2 Background

Practical Byzantine Fault-Tolerance (PBFT) [3] is an algorithm that allows to execute state machines in a distributed environment where up to f nodes and any number clients can be faulty. It guarantees that in a system with $3f + 1$ nodes all correct replicas will agree on the same output state machine should give.

2.1 BFT-SMaRt

BFT-SMaRt [9] is a Java library that implements all features required to run PBFT in a network of computers with JVM support connected using Internet Protocol (IP) [1]. It is a well maintained open source project that is used in multiple deployed to production distributed systems. That makes it a good foundation for researchers who want to experiment with a real world performance of BFT replication algorithms. Also, developers of it provide multiple demo applications. Their performance can be measured under different BFT-SMaRt modifications and used as a comparasent metrics.

2.2 WHEAT

In a geo distributed settings BFT system spends most of the time on a node to node message exchange due to high network latency. WHEAT paper provides literature survey on latency-related optimization topic. Most importantly it proposes to include extra replicas into a system without increasing quorum size, it makes number of different possible quorums larger. Also, WHEAT adds voting weights assignment scheme that allows well-connected nodes to be more important and form quorums of smaller size. These two optimizations allow achieving protocol latency improvement in a wide area networks.

2.3 AWARE

AWARE extends on the WHEAT weighted voting idea by introducing algorithm that dynamically allocates voting weights. It introduces monitoring mechanism, that periodically sends latency measurement messages to every other node. Results are later reported using a consensus mechanism and stored in a latency matrix. Final step of AWARE is to periodically use exhaustive search or simulated annealing to reassign voting weights and a leader in a way that minimizes overall consensus latency.

2.4 Intrusion tolerance

Intrusion tolerance is an original term describing a property of a system that can function with some components being faulty [6]. Making it possible was one of the original motivations behind developing BFT algorithms. However, wider adoption of systems based on BFT made other issues such as high latency or non-malicious faults more relevant for researchers. BFT-SMaRt paper mentions that its throughput can be reduced to 10% of a normal one during well coordinated attack by Byzantine nodes. Some algorithms were proposed to improve worst case scenario including Advrak [5] and Spinning [10]. We are going to analyse how intrusion tolerant are AWARE optimizations.

3 Methodology

First task of our algorithm should be to detect situations when faulty nodes are slowing down performance. Luckily AWARE computes predicted latency of reaching consensus while searching for the best voting weight configuration. We can compare that estimated number with a real amount of time it took to reach consensus. Large difference between them would mean that some kind of attack on the network is happening.

Next task for every correct node should be to identify attackers and punish them. Our proposal is to do it in extra self reflection step, right before latency reporting. During it every node reviews all successfully completed consensus instances to identify delayed ones. Next node will calculate estimated arrival time of each message in the consensus. Finally, it will run a new Reflection algorithm that can identify nodes who are delaying messages and punish them by increasing reported link latency to them. In a case when multiple correct replicas report high latency to a faulty node, that node should get low voting weights during the next reconfiguration.

3.1 Reflection algorithm

Reflection algorithm executes right before invoking report latency order. First it finds all messages in a monitoring window that arrived with a significant delay. It is calculated using latency matrix from a consensus message was part of. Significance is an adjustable parameter in our algorithm, we propose to calculate it as a ratio of delay to estimated message travelling time. It allows to compensate for some network randomness. Then we compute new estimation of that message arrival time using latest latency matrix. It gives a chance for a node who sent a delayed message to explain it by blaming some other link

it did depend on. However, if both estimations are earlier than a real arrival time it means node delays messages and can not explain it. So we will punish that node by setting new link latency to a value that would explain a delay. Instead of number that was computed using measurement messages.

This approach has a zero tolerance to late messages. Any delay in a monitoring window needs to be explained in the latest latency matrix. That means we not only mitigate faulty replicas effect by exposing their real latencies, but also get more conservative latency matrix that shows worst case delay for every link.

3.2 Delay estimation

Expected and actual message arrival times are essential information to make reflection algorithm work. Luckily AWARE provides a simulator to estimate global consensus speed. That means we just need slightly modify it to return intermediate expected times. In a BFT-SMaRt model there are 3 types of consensus messages PROPOSE, WRITE and ACCEPT. Their arrival times are P_t, W_t^n and A_t^n with n being sender ID (see Figure 1.). We can estimate waiting times between them using a latency matrix and compare calculations with actual measurements.

In the next section symbols will have this meaning:
$l_{a,b}$ - link latency between nodes a and b.
L - leader ID.
c - current node ID.
N - set of all node IDs.

3.2.1 PROPOSE messages

It is a first message that tells replica to create a new consensus instance. That means it can arrive any time, and we can only use real P_t as a reference point to start our delay estimation.

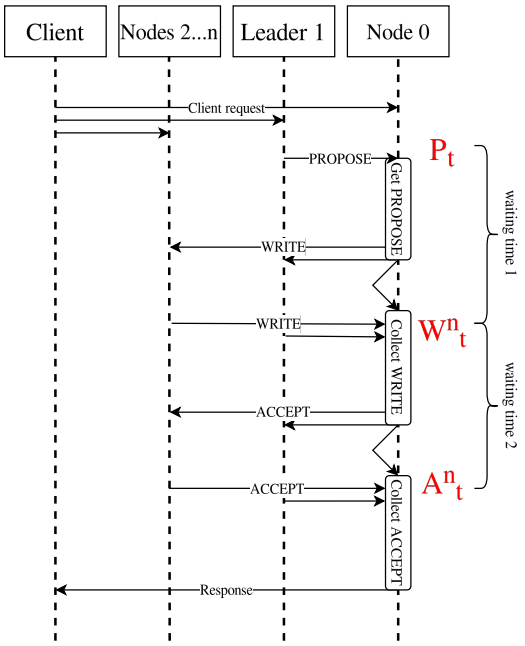


Figure 1: Sequence of message arrival times in BFT-SMaRt from Node 0 perspective.

3.2.2 WRITE messages

Every node going to receive some number of WRITE messages, at most one from every replica. Expectation of every arrival time W_t^n can be computed using a simple formula $W_t^{n*} = P_t - l_{L,c} + l_{L,n} + l_{n,c}$ First we start with a reference time P_t then we need to subtract $l_{L,c}$ from it to get a time when leader dispatched PROPOSE messages to all replicas. Last step is to add travel times of PROPOSE from a leader to node n and WRITE from n to a current node. Now we can subtract expected W_t^{n*} from actual W_t^n and get a delay.

3.2.3 ACCEPT messages

It is a bit more complicated to estimate ACCEPT arrival time, because correct node sends it only after collecting enough WRITE messages. That means to estimate A_t^{n*} we first need to estimate when WRITEs reached node n. If we know the time of WRITE arrival W_t^k to a current node c, then its arrival time to n should be $W_t^k - l_{k,c} + l_{k,n}$. So next we need to find latest of all WRITE messages that reached node n and allowed it to form a quorum. Finally we can estimate A_t^{n*} as $l_{n,c} + W_t^k - l_{k,c} + l_{k,n}$ where latest message sender ID is k in a quorum of node n.

3.3 Latency reporting

At this step we should finally punish nodes that are suspected to be faulty. However not all nodes that delayed a message are faulty. It might be that some of them could not collect enough voting weights on time. To distinguish between faulty nodes and nodes that are slowed down, we are going to do simple calculation. Currently, we have real message arrival times W_t^n, A_t^n and estimated times using old latency matrix W_t^{n*}, A_t^{n*}. Now we are going to compute other estimated arrival times $W_{t(new)}^{n*}$, $A_{t(new)}^{n*}$ using latest latency matrix $l_{N\times N}^{new}$ this time. That will give a chance to correct nodes that delayed a message to explain it by increasing some values in a latency matrix. However, in a case when still $W_{t(new)}^{n*} < W_t^n$ or $A_{t(new)}^{n*} < A_t^n$ that means node did not provide an explanation for the delay yet, so latency to it will be increased for now.

Also, we are going to introduce coefficient γ that will be used to calculate if delay is significant. To do it we will multiply γ by sum of all positive link

latencies used in message arrival time estimation and compare it with the delay. For example, if γ is 0.1 reflection algorithm will act only if WRITE delay is longer than $0.1 * (l_{L,n} + l_{n,c})$.

3.4 Formal algorithm outline

In an original AWARE implementation latency reporting algorithm runs periodically. During it node does 2 things:

1. Sends a measurement message with a random number to all other nodes to calculate link latencies.

2. Makes a request to start new consensus that will update corresponding row in a latency matrix with new latency vector $\vec{\boldsymbol{L}}$.

In our implementation we introduce extra reflection step between 2 mentioned tasks:

1. We start with new $\vec{\boldsymbol{L}}$ and latest latency matrix $l^{last}_{N\times N}$.
   $i \leftarrow$ Current node ID.
   $N \leftarrow$ Set of all node IDs.
   $\vec{\boldsymbol{L}} \leftarrow$ Vector of latest measurements.
   $l^{last}_{N\times N} \leftarrow$ Latest latency matrix from a Monitor.
   $\gamma \leftarrow$ Coefficient to calculate if delay is significant.
   $R_t \leftarrow$ Time when node sends response to a user.

2. Loop over every successfully completed (without a view change) Consensus instance in a monitoring time window. Compute a delay of every WRITE message:
   for c $\in$ {All stored consensus IDs} **do**
   $l^{old}_{N\times N} \leftarrow$ latency matrix at a consensus start.
   $P_t \leftarrow$ Propose for this consensus arrived.
   $L \leftarrow$ Leader of this consensus ID.
   for $W^n_t \in$ {WRITE arrival timestamps from every node} **do**
   $W^{n*}_t \leftarrow P_t - l^{old}_{L,i} + l^{old}_{L,n} + l^{old}_{n,i}$ ▷ Estimated arrival time.
   $W^{n*}_{t(new)} \leftarrow P_t - l^{last}_{L,i} + l^{last}_{L,n} + l^{last}_{n,i}$ ▷ Estimated new arrival time.
   $d^n_c \leftarrow W^n_t - W^{n*}_t$ ▷ Delay of the WRITE.
   end for
   end for

3. Now we can assume that leader is correct. So we can adjust latencies based on WRITE delays. This part makes sure that waiting times between PROPOSE and WRITE calculated based on the latest latency matrix are longer or equal than observed ones.

for c $\in$ {All stored consensus ids} **do**
  $l^{old}_{N\times N} \leftarrow$ latency matrix at a consensus start.
  $L \leftarrow$ Leader of this consensus id.
  for $n \in N$ **do**
    if $d^n_c = \infty$ and $W^{n*}_{t(new)} + \gamma * (l^{last}_{L,n} + l^{last}_{n,i}) < A^t_i$ **then**
      $\vec{\boldsymbol{L}}_{\boldsymbol{n}} \leftarrow \infty$
    else
      if $d^n_c > \gamma * (l^{last}_{L,n} + l^{last}_{n,i})$ **then**
        $\vec{\boldsymbol{L}}_{\boldsymbol{n}} \leftarrow max\{W^n_t - (P_t - l^{last}_{L,i} + l^{last}_{L,n}), \vec{\boldsymbol{L}}_{\boldsymbol{n}}\}$
      end if
    end if
  end for
end for

4. Last extra step has a similar goal as a previous one. It ensures that calculated waiting times between WRITE and ACCEPT messages are longer or equal to real times.

for c $\in$ {All stored consensus ids} **do**
  $l^{old}_{N\times N} \leftarrow$ latency matrix at a consensus start.
  for $A^n_t \in$ {ACCEPT arrival timestamps from every node} **do**
    $v \leftarrow 0$ ▷ Voting weight counter
    $K \leftarrow N$ ▷ Set of all node ids
    while $v < Q_v$ **do**
      $k \leftarrow$ such that $min\{W^k_t - l^{old}_{k,c} + l^{old}_{k,n}\} \mid k \in K$
      $v \leftarrow v + V_k$ ▷ Add voting weight
      $K \leftarrow K \setminus \{k\}$
    end while
    $A^{n*}_t \leftarrow l^{old}_{n,c} + W^k_t - l^{old}_{k,c} + l^{old}_{k,n}$
    $v \leftarrow 0$ ▷ Voting weight counter
    $K \leftarrow N$ ▷ Set of all node ids
    while $v < Q_v$ **do**
      $k \leftarrow$ such that $min\{W^k_t - l^{last}_{k,c} + l^{last}_{k,n}\} \mid k \in K$
      $v \leftarrow v + V_k$ ▷ Add voting weight
      $K \leftarrow K \setminus \{k\}$
    end while

$A^{n*}_{t(new)} \leftarrow l^{last}_{n,c} + W^k_t - l^{last}_{k,c} + l^{last}_{k,n}$
$d^n_c \leftarrow A^n_t - A^{n*}_t$ ▷ Delay of the ACCEPT from n.
if $d^n_c = \infty$ and $A^{n*}_{t(new)} + \gamma * (l^{last}_{n,c} + l^{last}_{k,n}) < R_t$ **then**
$\vec{\boldsymbol{L}}_{\boldsymbol{n}} \leftarrow \infty$
else
if $d^n_c > \gamma * (l^{last}_{n,c} + l^{last}_{k,n})$ **then**
$\vec{\boldsymbol{L}}_{\boldsymbol{n}} \leftarrow max\{A^n_t - (W^k_t - l^{last}_{k,c} + l^{last}_{k,n}), \vec{\boldsymbol{L}}_{\boldsymbol{n}}\}$
end if
end if
end for
end for

5. Now that vector $\vec{\boldsymbol{L}}_{\boldsymbol{n}}$ have been adjusted we can report it in a same way as it is done in AWARE implementation.

4 Experiments and results

To test how reflection algorithm affects latency of AWARE we are going to run counter app in a locally simulated network. You can find it in the BFT-SMaRt demo applications folder. This is an implementation of distributed counter that increases by 1 with each request and then returns latest value. So, to compare performance of AWARE and Reflection algorithms we are going to measure how fast value increasing transactions are executed in a specific network set up. Also, as goal of reflection algorithm is to mitigate performance degradation caused by Byzantine nodes we will test it under multiple attack patterns.

4.1 Configuration

First we need to create a specific network configuration where Byzantine nodes can cause evident damage to performance. One possible attack for Byzantine nodes is to respond correctly to all latency measurement messages, but to ignore or delay some consensus messages. That can lead AWARE to assign V_{max} voting weights to faulty nodes and consequently increase quorum size correct replicas need to form to advance Blockchain.

For example, in a configuration with parameters $f = 2$ and $\Delta = 1$ (see Figure 2.). Where 7 replicas have similar node to node latencies,

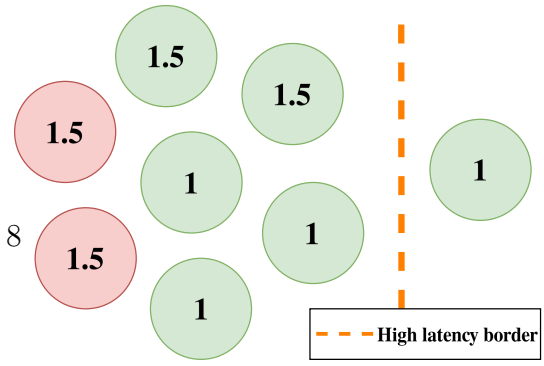


Figure 2: Experiment setup with $n = 8$ nodes and voting weights allocation

however one extra correct replica is located in a different region and all links to it have high latency. In a case when 5 correct well-connected replicas obtain all 4 V_{max} weights they will manage to collect $Q_v = 7$ and advance blockchain without waiting for distant replica or Byzantine node response. However, if any of faulty nodes will manage to get V_{max} distant node will become necessary for the progress. As a result transactions will be completed way slower than in a case where all nodes function properly.

4.2 Latency simulation

To create a simulation we used Linux Network Namespaces feature, with separate namespace for every node and `veth` devices to link them together (See Figure 3.). Also, Linux Traffic Control (TC) subsystem allowed me to introduce queueing discipline (qdisc) that simulates latency with a jitter between nodes. So final setup includes 8 namespaces for nodes and 1 for a client each of them has a `veth` device that attaches forwards all messages to the bridge. This setup does not allow simulating every node to node latency independently, however it is sufficient to create network with a distant node.

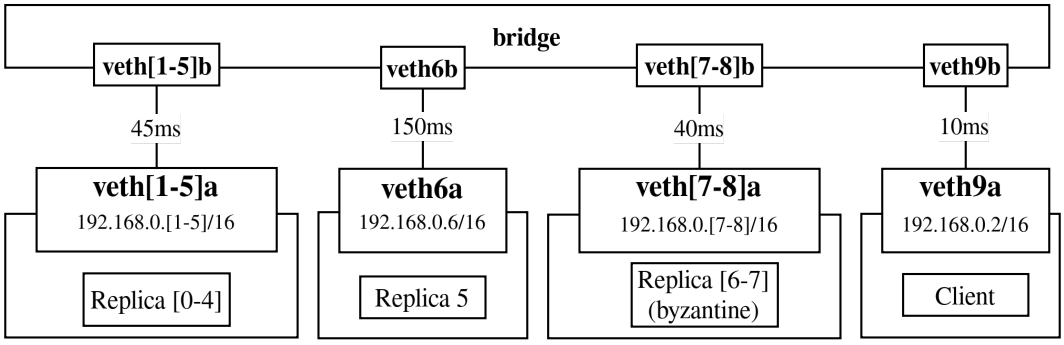


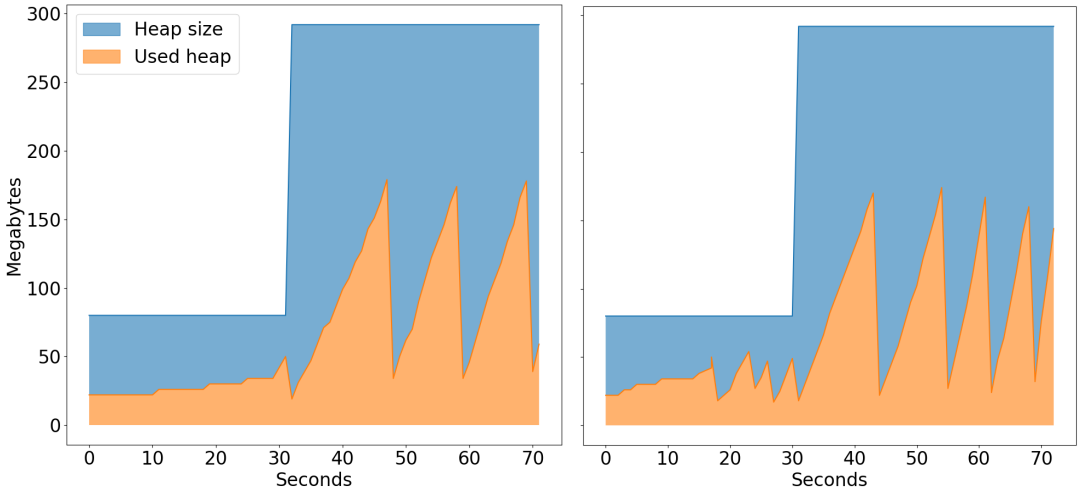
Figure 3: Namespaces and virtual cables with latency to connect a network.

Using kernel features like namespace makes latency Introduction more reliable. User space program isolated by kernel simply can not reach any other process without using virtual cables. Also, it allows simulating latencies without any modifications of original AWARE source code. That minimizes chance of causing bugs when adding latency.

4.3 Results

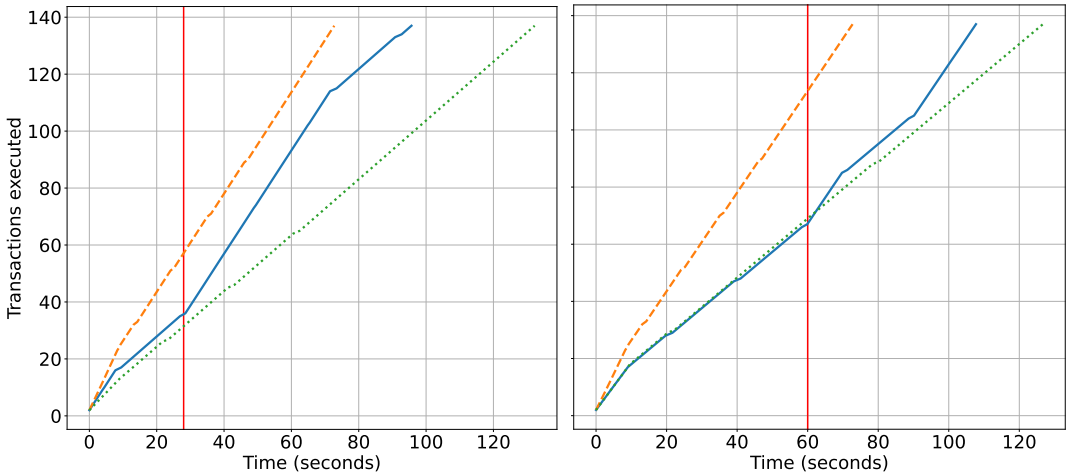
All experiments show reflection performance compared to an AWARE in previously mentioned configuration (Figure 2). We can see the number of transactions a distributed counter executes during different attacks made by Byzantine nodes (see Figure 5.). In a first attack (Sub figure 5a.) both Byzantine nodes respond to measurement messages, but don't send WRITE or ACCEPT. In a second attack (Sub figure 5b) Byzantine nodes don't send only ACCEPT messages. We can see how Byzantine nodes slow down the network. They manage to obtain V_{max} and force other nodes to rely on a slow quorum with a distant node to make progress. It explains why AWARE attacked executes transactions slower than AWARE normal. However, Reflection algorithm manages to detect nodes that produce fake reports and increases latency to them. We can see that after few epochs of working in a slow configuration reconfiguration happens and Reflection algorithm reallocates voting weights of Byzantine nodes to correct nodes. The result is that counter speed gets on par with a AWARE normal network speed. Yet later Byzantine nodes manage to get V_{max} and slow down network again. That happens due to short Reflection window, but its length is configurable parameter that can be adjusted based on the use case.

Reflection algorithm keeps completed Epoch instances in memory to review message arrival times in them. So max number of latest Epochs on a heap is a parameter that should be adjusted. In experiments, it was set to 100 compared to 3 in a default AWARE. Yet we can see in a (Figure 4.) that it does not cause any significant heap utilization increase. So it can be set to a way larger number.

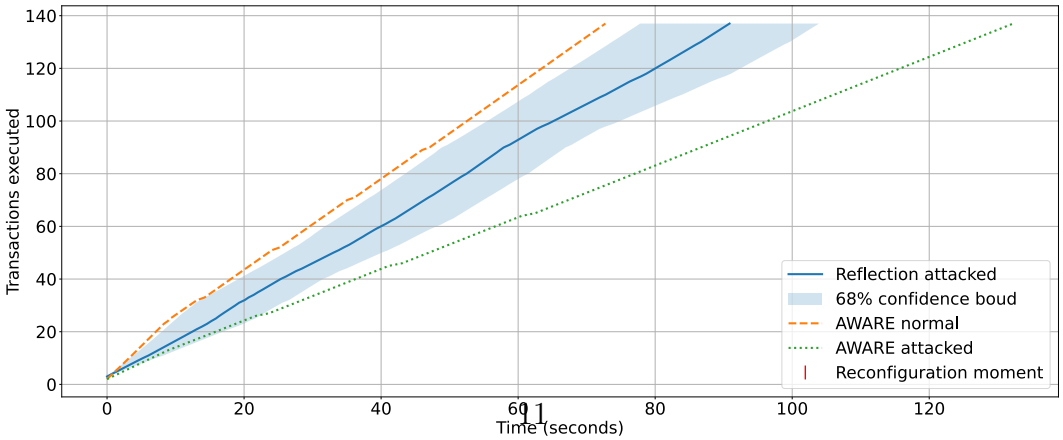


(a) Reflection (b) AWARE

Figure 4: Heap size comparison



(a) Drop all messages (b) Drop only ACCEPT messages



(c) Drop all messages Reflection average performance over 8 runs

Figure 5: Comparison of counter application speed in a correct network and networks with Byzantine nodes. Both tests show Reflection and AWARE algorithms.

5 Responsible Research

Experiment have been conducted in a virtual network simulation. That means everyone who has available computer with a Linux can reproduce configuration from fig 3. Also distributed counter application and AWARE implementation with setup instructions can be found in repo:
`https://github.com/bergerch/aware.git`.
Source code for reflection algorithm is located in our fork of AWARE repo:
`https://github.com/magisterbrown/bft-reflection.git`,
so it can be executed by following same steps. `main` branch has only changes that implement Reflection algorithm and network simulation. Branches that have `attack-*` prefix in their name, add Byzantine nodes to a network. We used them for our experiments.

6 Discussion

Reflection approach introduces intrusion tolerance into latency aware BFT replication algorithms. However, when building real world application, that expects to face attacks from coordinated Byzantine replicas, it might be not enough to use Reflection algorithm only. Because real attack will probably involve combination of delaying messages and sending fake data. That means such system will have to analsze message content in addition to arrival time. Luckily faulty node detection based on message data is well researched topic called Accountability. For example, on of algorithms solving this problem is called Polygraph [4] and can be easily integrated with AWARE because both of them are based on BFT-SMaRt library.

Another benefit of a Reflection algorithm is that it makes latency matrix more conservative. If AWARE latency measurement message completes round trip faster due to network randomness or a smaller size than payload messages, then latency matrix won't represent reality any more. But Reflection algorithm going to adjust its latency according to the slowest payload message, so weights allocation algorithm will make a decision based on a more realistic data.

6.1 Leader selection

AWARE algorithm has an optional optimization that does dynamic leader reallocation to a well-connected site, after voting weights reconfiguration. However, in a worst case scenario malicious leader can cause significant damage to a system performance. BFT-SMaRt paper mentions that throughput in such case can be reduced to 10% of what would it be in a fault free execution.

There are multiple ways to deal with a faulty leader. For example Spinning algorithm changes leader after each consensus. Advrak checks minimal throughput of PRE-PROPOSE messages per second from a leader node and starts reallocation if underperformance is detected. Both of them agree that any single node can not be trusted to stay a leader for a long time. Overheads introduced by regular view changes are small compared to delays that faulty leader can cause.

Increasing amount of view changes, that was already quite high in our experiments (Figure 5.), would completely remove any benefits that dynamic leader reallocation provides. Any time AWARE chooses the best leader it would get soon replaced during some other view change. We think that AWARE approach of implementing custom leader change mechanism that works in parallel with one from BFT-SMaRt is beneficial only in an average case, but can result in unpredictable behaviour under attack. More robust approach would be to leave leader changes to original algorithm, and modify ordering of nodes. Instead of constantly looping over all node IDs it would make better connected node a leader more than once per iteration. That would allow doing frequent view changes of AWARE to stick with one leader for a long period of time.

6.2 Improvement on AWARE base case

So as reflection algorithm manages to deduce extra information about link latencies without any additional information exchange. That means it should be possible to calculate link latencies based only on payload message arrival times. Current Reflection algorithm uses latency measurement vector as a baseline and increases value that are too low in it. But we could also use vector of zeroes instead. That would allow completely remove measurement messages and benefit BFT algorithms by reducing total amount of data exchange. However, this approach would make latency estimations between correct replicas less precise. So comparison between algorithms that use and don't use latency measuring messages can be a great next direction for the research of AWARE like algorithms.

7 Conclusion

Our approach adapts dynamic link latency estimation algorithm for networks with coordinated Byzantine nodes. It allows detecting links that delay messages and increase their latency in a matrix. Our experiments show that it

helps AWARE to reassign V_{max} to actually well-connected nodes, instead of blindly trusting results of measurement messages. Also, we provide multiple parameters to adapt Reflection algorithm to different levels of network noisiness and node compute capabilities.

Another conclusion of our research is that benefits of AWARE dynamic leader selection can be eliminated by faulty nodes. So it is better to disable this optimization in a network where attacks are expected. However, we provide a direction for research on how to use latency information within BFT-SMaRt leader change algorithm. That could help to combine AWARE with intrusion tolerant systems.

References

[1] Internet Protocol. RFC 791, September 1981.

[2] Christian Berger, Hans P. Reiser, João Sousa, and Alysson Bessani. Aware: Adaptive wide-area replication for fast and resilient byzantine consensus. *IEEE Transactions on Dependable and Secure Computing*, 19(3):1605–1620, 2020.

[3] M. Castro and B. Liskov. Practical byzantine fault tolerance. *Proceedings of the Third Symposium on Operating Systems Design and Implementation*, page 173–186, 1999.

[4] Pierre Civit, Seth Gilbert, and Vincent Gramoli. Polygraph: Accountable byzantine agreement. *2021 IEEE 41st International Conference on Distributed Computing Systems (ICDCS)*, pages 403–413, 2021.

[5] Allen Clement, Edmund Wong, Lorenzo Alvisi, Michael Dahlin, and Mirco Marchetti. Making byzantine fault tolerant systems tolerate byzantine faults. pages 153–168, 01 2009.

[6] Y. Deswarte, L. Blain, and J.-C. Fabre. *Intrusion tolerance in distributed computing systems*. 1991.

[7] Butler Lampson. Hints for computer system design. *ACM SIGOPS Operating Systems Review*, 15:33–48, 10 1983.

[8] João Sousa and Alysson Bessani. Separating the wheat from the chaff: An empirical design for geo-replicated state machines. *34th IEEE Symp. on Reliable Distributed Systems (SRDS)*, pages 146–155, 09 2015.

[9] João Sousa, Alysson Bessani, and Marko Vukolic. A byzantine fault-tolerant ordering service for the hyperledger fabric blockchain platform. *48th Annu. IEEE/IFIP Int. Conf. on Dependable Systems and Networks (DSN).*, pages 51–58, 06 2018.

[10] Giuliana Veronese, Miguel Correia, Alysson Bessani, and Lau Lung. Spin one's wheels? byzantine fault tolerance with a spinning primary. pages 135–144, 09 2009.