Improvement and Partial Simulation of King & Saia's Expected-Polynomial-Time Byzantine Agreement Algorithm

by

Ben Kimmett

B.Sc., University of Victoria, 2016

A Thesis Submitted in Partial Fulfillment of the

Requirements for the Degree of

MASTER OF SCIENCE

in the Department of Computer Science

Improvement and Partial Simulation of King & Saia's Expected-Polynomial-Time Byzantine Agreement Algorithm

by

Ben Kimmett

B.Sc., University of Victoria, 2016

Supervisory Committee

Dr. V. King, Co-Supervisor

(Department of Computer Science)

Dr. Y. Coady, Co-Supervisor

(Department of Computer Science)

ABSTRACT

We present a partial implementation of King and Saia 2016's expected polynomial time byzantine agreement algorithm, which which greatly speeds up Bracha's Byzantine agreement algorithm by introducing a shared coin flip subroutine and a method for detecting adversarially controlled nodes. In addition to implementing the King-Saia algorithm, we detail a new version of the "blackboard" abstraction used to implement the shared coin flip, which improves the subroutine's resilience from $t < n/4$ to $t < n/3$ and leads to an improvement of the resilience of the King-Saia Byzantine agreement algorithm overall. We test the King-Saia algorithm, and detail a series of adversarial attacks against it; we also create a Monte Carlo simulation to further test one particular attack's level of success at biasing the shared coin flip.

Contents

List of Algorithms

List of Figures

List of Tables

ACKNOWLEDGEMENTS

Special thanks to:

My parents, for their everlasting support.

Drs. King and Coady, for sticking with it despite every delay.

Everyone at EGM, for helping me keep on an even keel. Nick, Yas, Adam, this one's for you.

DEDICATION

Dedicated to groups, human or computer,

who work together to overcome a great adversary.

Chapter 1

About Byzantine Agreement

1.1 Statement of the Problem

The problem of *Byzantine agreement* deals with a network of n processors (also called nodes), which must agree on a value even though some processors may fail or act maliciously under the influence of an adversary. In *single-valued Byzantine agreement*, the value to be agreed on is a binary choice; this can be represented as a Boolean value, *true* or *false*. (For this thesis, further mentions of Byzantine agreement refer to single-valued Byzantine agreement, and Boolean values will be used for agreement instances' values.)

Each processor in Byzantine agreement has an *initial value* at the start of a Byzantine agreement instance, which is one of the two possible values to be decided upon. Processors communicate with one another by sending messages on a set of reliable one-to-one communication links; each processor can send any number of messages to any other processor, and each message is guaranteed to arrive eventually (that is, there is no limit to the amount of time that may pass between a message being sent and its arrival, but the message *must* arrive). In addition, any message sent on such

a communication link is guaranteed not to be altered in transit. There is, however, no guarantee that messages will arrive in any specific order.

The adversary in Byzantine agreement may corrupt up to t processors (where t is a constant fraction of n. The maximum t for any Byzantine agreement algorithm is referred to as the algorithm's *resilience*). Processors that become corrupted by the adversary are referred to as *adversarial*. For the purposes of the problem, adversarial processors behave in any way the adversary requires; this may include imitating a good node, sending messages that are substantially different from the ones a good node would send, or acting as though they have crashed.

Once a processor comes to the conclusion that all non-adversarial (good) processors agree on a value, it *decides* on a value. The adversary succeeds at disrupting Byzantine agreement if it can get good nodes to decide on differing values, or if it can convince any good processor to decide on a value that no good processor had as an initial value. The adversary also succeeds if it deadlocks the process, causing one or more good nodes to never decide.

1.2 History of Byzantine Agreement

Pease, Shostak and Lamport first introduced the concept of Byzantine agreement in 1980 [22], though it took two more years for the term "Byzantine agreement" to appear in the literature. Ben-Or came up with a solution in the asynchronous model of communication to Byzantine agreement in 1983, with a resilience of $t < n/5$ [2].

In 1987, Bracha improved Ben-Or's Byzantine agreement algorithm by adding a validation step. The validation step ensured that the algorithm would only accept messages that could have been sent by correctly operating processes, increasing the resilience to $t < n/3$ [5]. Since that time, the basic structure of the Byzantine

agreement algorithm has not changed for the model addressed in those pa. However, if the algorithm is subject to continued interference by an adversary, it may take an exponential amount of time to complete.

King and Saia put forward an attempt to speed up Bracha's Byzantine agreement in 2016 [16]. The King-Saia algorithm incorporated concepts from an algorithm known as Spread [3], which let the nodes running King-Saia quickly synchronize a "shared coin flip" among good nodes to let it recover from continued interference faster.

This thesis builds on King-Saia, improving its resilience in the "shared coin flip" section.

1.3 Implementation History of Byzantine Algorithms

There has been little attention paid to implementing Byzantine agreement compared to other types of resilient system. Many implementations of *byzantine fault-tolerant* (BFT) systems exist: that is, systems built to accomplish a specific task that withstand byzantine faults (the interference of an adversary). However, most of these BFT systems do not focus on the problem of agreement. BFT system implementations are commonly used for the task of state machine replication (where multiple nodes perform, or collaborate on, the same task, and if some nodes fail, the task can still be completed). Gray [14] suggests that little attention has been paid to implementing Byzantine agreement because other algorithms provide quicker results in exchange for less resilience and overhead.

In the field of implementations of BFT state replication, Castro and Liskov's PBFT [9] is of particular note for helping define the term "byzantine fault-tolerance", effectively naming the subfield around the problem. A number of other BFT state

replication algorithms have been made since then with various use cases and attributes, such as Kotla *et al.*'s Zyzzyva (in which, to speed up the algorithm, nodes return task results without synchronizing among themselves, and clients assist in validating results) [19], and EBAWA by Veronese *et al.* (which is designed to be used over nodes that are separated on a wide-area network, such as the Internet) [24]. The majority of these algorithms have been tested using a set of benchmarks that Castro and Liskov also proposed in their paper for PBFT. [9]

By comparison, there are few documented Byzantine agreement implementations. Only a few examples exist; perhaps the most well-known is Algorand [13], which uses a Byzantine agreement algorithm of its own devising as a building block of a hypothetical blockchain-based currency. Cachin and Poritz created a collection of fault-tolerant protocols known as SINTRA, which includes Byzantine agreement [8]. Agrawal and Daudjee surveyed the field of Byzantine agreement algorithms in 2016 [1], testing implementations of three such algorithms (Quorum [4], Pull-Push [6, 18], and EIG [20]). Finally, Oluwasanmi *et al.* implemented an algorithm based on King *et al.*'s earlier work, using a model where messages will always be delivered within a set amount of time [21]. Because of the relative rarity of byzantine agreement implementations, no benchmarking or testing suite similar to Castro and Liskov's exists for these algorithms.

Chapter 2

Algorithm Design

2.1 Bracha's Byzantine Agreement

The work in this thesis modifies the King-Saia Byzantine agreement algorithm. However, a large part of King-Saia is based on Bracha's Byzantine agreement. In order to explain the differences between King-Saia's and Bracha's algorithms, we need to recap how Bracha's Byzantine agreement works.

2.1.1 Reliable Broadcast

To solve the potential problem that adversarial nodes could lie about the content of a message to some nodes but not to others, all messages in Bracha's Byzantine agreement are sent through *Bracha's reliable broadcast* (Algorithm 1), a broadcast algorithm that ensures nodes all eventually receive the same message, as long as one good node receives it. A node that accepts a message sent through reliable broadcast has *r-received* that message. Bracha's reliable broadcast has a resilience of $t < n/3$.

Reliable broadcast has the following properties:

Algorithm 1 Bracha's Reliable Broadcast

1: Send $(initial, m)$ to all processors.
2: Upon receiving either $(initial, m)$, $\frac{n+t}{2}$ $(echo, m)$ messages, or $t+1$ $(ready, m)$ messages, send $(echo, m)$ to all processors.
3: Upon receiving either $\frac{n+t}{2}$ $(echo, m)$ messages or $t+1$ $(ready, m)$ messages, send $(ready, m)$ to all processors.
4: Upon receiving $n-t$ $(ready, m)$ messages, r-receive m.

- Consistent Value: Each node that r-receives a message m will receive the same m as all other nodes that r-receive the message.
- Duplicate Prevention: Once a message m is broadcast, each node may r-receive that message only once.
- Basic Delivery: If a good node broadcasts a message m, all other good nodes will eventually r-receive m.
- Adversarial Delivery: If an adversarial node broadcasts a message m, all good nodes *or* no good nodes will eventually r-receive m.
- Participation Guarantee: If a message is r-received, at least $n-2t$ good nodes participated in the broadcast. This is because $n-t$ nodes must send *ready* messages for a good processor to r-receive a message; of those, $n-2t$ must be good nodes.

2.1.2 Byzantine Agreement

Bracha's Byzantine agreement (Algorithm 2) works in sets of three "waves". In each wave, every processor reliably broadcasts its current value. Then, it may change its value depending on messages, r-received from other processors, that it can successfully validate.

Algorithm 2 Bracha's Byzantine Agreement

1: $v_p \leftarrow$ processor p's initial value.
2: **while** there is no decision, repeat **do**
3: **Wave 1:** Reliably broadcast v_p and wait until validate $n - t$ messages.
4: $v_p \leftarrow$ majority of the r-received messages.
5: **Wave 2:** Reliably broadcast v_p and wait until validate $n - t$ messages.
6: If more than $n/2$ messages have same value v then $v_p \leftarrow (v, decide)$.
7: **Wave 3:** Reliably broadcast v_p and wait until validate $n - t$ messages.
8: Let $x =$ number of messages of the form $(v, decide)$ that are r-received.
a) CASE $x > 2t$: decide v;
b) CASE $t < x \leq 2t$: $v_p \leftarrow v$;
c) CASE $x \leq t$: $v_p \leftarrow$ the result of a fair coin-flip.
9: **end while**

In the context of Bracha's Byzantine agreement, to *validate* a message refers to a process where the node receiving a broadcast value compares it against messages from previous rounds. A message is only validated if, given what the receiving node knows about past messages that have been broadcast, the sending node *could have* broadcast the message undergoing validation while operating correctly by the Byzantine agreement protocol. Messages that fail validation are discarded. Bracha's Byzantine agreement has $t < n/3$ resilience.

Once a set of three waves have completed, a processor either decides, or begins a new set of waves. This continues indefinitely until a decision is reached. If one good processor decides at the end of a set of waves, all good processors will decide by the end of the next set [5].

When a node begins a new set of waves, it resets its value based on the results of the most recent set of waves. In the worst-case scenario, Bracha's Byzantine agreement will continue until all good nodes start some set of waves with the same value, which may require that every node that flips a coin (case 8C) comes up with the same result; this value must also agree with the value of any nodes that set their

value according to case 8B. This may take a *very* long time (number of sets), though it will terminate eventually.

2.2 King and Saia's Changes

In King-Saia, one of the major changes is an attempt to make the algorithm quicker to complete in the worst case. King-Saia uses a modified form of Bracha's Byzantine agreement known as *Modified-Bracha* (Algorithm 3), which replaces each node's individual coin flip with an *interactive consistency* algorithm that creates a shared coin flip (Global-Coin) from the sum of multiple coinflips performed by each node.

Interactive consistency refers to an algorithm that runs on a network of nodes, where some nodes may be faulty. Each node has a *private value* and communicates with other nodes by sending messages on a set of reliable one-to-one communication links. The goal of an interactive consistency algorithm is that, once the algorithm is complete, each good node will know every other good node's private value. In the case of Global-Coin, the node's private value is the set of coinflips it makes during

Algorithm 3 Modified-Bracha

1: $v_p \leftarrow$ processor p's initial value.
2: **while** there is no decision, repeat **do**
3: **Wave 1:** Reliably broadcast v_p and wait until validate $n - t$ messages.
4: $v_p \leftarrow$ majority of the r-received messages.
5: **Wave 2:** Reliably broadcast v_p and wait until validate $n - t$ messages.
6: If more than $n/2$ messages have same value v then $v_p \leftarrow (v, decide)$.
7: **Wave 3:** Reliably broadcast v_p and wait until validate $n - t$ messages.
8: Let $x =$ number of messages of the form $(v, decide)$ are r-received.
a) CASE $x > 2t$: decide v;
b) CASE $t < x \leq 2t$: run Global-Coin but set $v_p \leftarrow v$;
c) CASE $x \leq t$: $v_p \leftarrow$ Global-Coin.
9: **end while**

Global-Coin's run; the 'result' of Global-Coin for a given processor is the sign of the sum of all the coinflips that processor receives through the Global-Coin algorithm.

Replacing the individual coinflips of Bracha with a shared coin flip helps each good node attain the same value as other good nodes much more quickly than Bracha's Byzantine agreement; any process that participates in Global-Coin either sets its value to the result of Global-Coin, or r-received more than t messages of the form $(v, decide)$ and will set its value to v, as in Bracha. (Bracha proves that v will be the same for all processors that set their values in this way. [5])

Assuming no adversary interference, the shared coin flip has a 50% chance to match any v that a node r-received more than t $(v, decide)$ messages for. In this case, all good nodes will start with the same initial value v in the next set of waves, and will decide.

(While it is possible for the adversary to interfere with Global-Coin, if interference continues for enough attempts, King-Saia has a third part it runs, Process-Epoch, which identifies probably-adversarial nodes and blacklists them for future runs of Global-Coin. We don't cover Process-Epoch in this thesis.)

In the original version of King-Saia, Modified Bracha had $t < n/3$ resilience, but Global-Coin had only $t < n/4$ resilience. This thesis introduces an upgraded version of Global-Coin which has resilience of $t < n/3$, while preserving the properties of the original Global-Coin.

2.3 Algorithm for Improved Resilience

The core of the Global-Coin interactive consistency algorithm (Algorithm 4) is a synchronization primitive known as an x-sync, which refers to a procedure where each node builds up a view of a "blackboard" that holds all values emitted by all

Algorithm 4 King-Saia's Global-Coin

1: **x-sync:** Participate in an n-sync, in which each message is the outcome of a private fair coinflip.
2: **Exclude suspicious columns:** Once the n-sync concludes, if the sum of the coins in any column j of the blackboard have absolute value greater than $5\sqrt{n \ln n}$, exclude column j from the blackboard.
(Coinflips are counted as +1 for heads, and -1 for tails.)
3: **Compute global coinflip:** Sum up the entries in all non-excluded columns of the blackboard, and set one's vote to the sign of the sum.

nodes as part of the process. In an x-sync, each node emits at most x values to the blackboard; in Global-Coin's x-sync, each of the values a node emits is a coinflip performed by that node.

An emitted value is *written to the blackboard* if all good nodes are guaranteed to have that value in their view of the blackboard by the time the x-sync concludes. If a node's coin flip has not been written to the blackboard, that cell of the blackboard may be *ambiguous* (some good nodes will have the value in their view of the blackboard once the x-sync concludes, and some will not) or *null* (no good nodes will have the value in their view of the blackboard).

Global-Coin's x-sync is required to have the following properties:

- Order: Each node emits up to x numbered coinflips, $f_1 ... f_x$. No good node will emit a coinflip f_i, where $i > 1$, unless it can be sure that its f_{i-1} is written to the blackboard.

- Column Guarantee: The blackboard organizes all coin flips emitted by the same node in a "column". Once the x-sync has concluded, there will be at least $n - t$ "full" columns containing x values, where every value emitted by that column's node has been written to the blackboard.

- Leftover Column Ordering: Once the x-sync has concluded, any column that is not a full column will have, in emit order, some number of values i ($0 \leq i < x$) that are written to the blackboard, followed by up to one ambiguous value, followed by some number of null values (which have not been emitted).

We present an updated x-sync algorithm which improves the procedure's resilience from $t < n/4$ to $t < n/3$. The updated algorithm is based on King-Saia's x-sync algorithm, which we detail first so that the two algorithms can be compared.

2.3.1 King-Saia's x-sync

King-Saia's x-sync (Algorithm 5) operates in three phases. The first phase is a *generate* phase, where each node j reliably broadcasts its own messages (coinflips), reliably broadcasting acknowledgements when it r-receives messages. A node only broadcasts its next message after it r-receives $n - t$ acknowledgements to its most recent message; this prevents the adversary from breaking the x-sync's "Order" guarantee. (If a node r-receives $n - t$ acknowledgements for a message in King-Saia's x-sync, the node can be sure that message will be written to the blackboard.)

During the *generate* phase of King-Saia's x-sync, each node j also assists in the broadcast of messages from any other node j' – but only as long as enough acknowledgements for previous messages from j' have been r-received. A node leaves the generate phase once it has r-received $n - t$ full columns of messages.

The second phase of King-Saia's x-sync is a *spread* phase, where each node reliably broadcasts their view of the blackboard. By only participating in other nodes' broadcasts if the messages in the broadcasted views have already been r-received, the spread phase acts to delay the algorithm to ensure that any message that was acknowledged enough will be written to the blackboard properly.

Algorithm 5 King-Saia's x-sync for each processor j

1: $i \leftarrow 1$

{*Generate Messages:*}

2: **While** there are fewer than $n - t$ processors j' such that $message(1, j'), ..., message(x, j')$ have been r-received **do in parallel**

3: Generate and reliably broadcast $message(i, j)$. Participate in reliable broadcast for $message(i, j')$ for any processor j', only if $i = 1$ or have r-received $n - t$ acknowledgements for $message(i - 1, j')$.

4: Upon r-receiving $message(i, j')$ for any processor j', reliably broadcast an acknowledgement for $message(i, j')$.

5: **if** acknowledgements for $message(i, j)$ are r-received from $n - t$ processors and $i < x$ **then**

6: $i \leftarrow i + 1$ (increment i).

7: **end if**

8: **end while**

{*Spread:*}

9: Reliably broadcast the matrix BB_j (j's view of the blackboard), where $BB_j(i, j') = message(i, j')$ if $message(i, j')$ has been r-received, and null otherwise. Participate in the reliable broadcast of $BB_{j'}$ for any other processor $j' \neq j$ only if $BB_{j'}$ contains $n - t$ full columns and each entry in $BB_{j'}$ has been r-received.

10: Wait until matrices $BB_{j'}$ are r-received from $t + 1$ different processors j''.

11: Update BB_j: replace any null entry $BB_j(i, j')$ by the non-null $message(i, j')$ which has been r-received.

{*Resolve:*}

12: Reliably broadcast BB_j.

13: Wait until matrices $BB_{j'}$ are r-received from $n - t$ different processors j''.

14: Update BB_j: replace any null entry $BB_j(i, j')$ by the non-null value in $BB_{j'}(i, j')$ if it appears in the $BB_{j'}$'s of $t + 1$ different processors j''.

A node leaves the spread phase once it has r-received $t + 1$ *spread* views, enough to ensure it has r-received a view from at least one good node.

The final phase of King-Saia's x-sync is a *resolve* phase, where each node reliably broadcasts their view again. Unlike the spread phase, there is no participation requirement for the resolve phase; every node assists in each broadcast. This is because the purpose of the resolve phase is to share views as widely as possible; a node with a message missing from its blackboard will replace it with the version of the message that exists in the *resolve* views it receives, as long as the message's value is consistent across at least $t + 1$ views.

A node leaves the resolve phase and concludes King-Saia's x-sync once it has r-received $n - t$ *resolve* views.

The reliance on sharing views in the later phases of King-Saia's x-sync is what restricts the algorithm's resilience to $t < n/4$ (see Lemma 4.2 of [16]). While the initial mechanism for disseminating messages (reliable broadcast) works with $t < n/3$ resilience, the dissemination of copies of views as a mechanism is less resilient.

The updated algorithm relies only on reliable broadcast to disseminate messages, allowing the increased resilience.

2.3.2 Updated x-sync

The updated x-sync uses only two phases: a *generate* phase and a *resolve* phase.

The updated x-sync's *generate* phase is very similar to that of King-Saia's x-sync; each node reliably broadcasts its own messages, and when a node r-receives a message, it reliably broadcasts an acknowledgement message. Each node will only broadcast its next message once it has r-received $n - t$ acknowledgements for its most recent message, and it will only participate in the broadcast of other nodes' messages if it has r-received $n - t$ acknowledgements for the messages that preceded them.

Algorithm 6 Updated x-sync for each processor j

1: $i \leftarrow 1$

{*Generate Messages:*}

2: **Loop until** there are $n - t$ different j' such that, for each j', broadcasts of $acknowledge(x, j')$ have been r-received from $n - t$ different processors. **do in parallel:**

3:

A: Generate and reliably broadcast $message(i, j)$, where i is the value of a counter $(1 \leq i \leq x)$ that numbers j's flips in order.

B: Participate in reliable broadcast for $message(i', j')$ for other processors j' only if $i' = 1$ or have r-received $n - t$ acknowledgements for $message(i' - 1, j')$.

4: On r-receiving $message(i', j')$ for any processor j', add it to BB_j (j's view of the blackboard), and reliably broadcast an acknowledgement: $acknowledge(i', j')$.

5: **if** acknowledgements for $message(i, j)$ are r-received from $n - t$ different processors, and $i < x$ **then**

6: $i \leftarrow i + 1$ (increment i).

7: **end if**

8: **end loop**

Continue Step 3B (participation in reliable broadcast of messages) until Step 11 below. However, *do not* broadcast acknowledgements to r-received messages (Step 4) after exiting the loop.

9: {*Resolve Differences:*}

A: Reliably broadcast the list $list_j$, where $list_j(j')$ = the greatest i' of any $message(i', j')$ that j has r-received.

B: Participate in the reliable broadcast of list $list_{j'}$ from any other processor $j' \neq j$ only if, for each entry $list_{j'}(other\_j)$ in $list_{j'}$, $message(1, other\_j)$ through $message(list_{j'}(other\_j), other\_j)$ have been independently r-received by j (that is, only if j has r-received all the entries that j' claims to have r-received in $list_{j'}$).

10: Wait until $n - t$ lists $list'_{j'}$ are r-received from $n - t$ different processors.

11: **Update** BB_j: Replace any null entry $BB_j(i', j')$ with the non-null $message(i', j')$ that j independently r-received.

(Like in King-Saia's x-sync, if a node r-receives $n-t$ acknowledgements for a message, it can be sure (by Lemma 3.2.1, in the next chapter) that that message is written to the blackboard.)

The primary difference between the two algorithms is the condition for when a node will exit the *generate* phase. In King-Saia's x-sync, a node leaves the phase once its own view of the blackboard has enough values; the *spread* and *resolve* phases are then dedicated to synchronizing the nodes' views of the blackboard to prevent ambiguities.

In the updated x-sync, a node leaves the *generate* phase once it has r-received acknowledgements that indicate that at least $n - t$ full columns of messages *will* be written to the blackboard once the algorithm concludes – regardless of the state of the node's own view of the blackboard. This is because once a node has r-received these acknowledgements, the node can be sure that every good node will continue to run the updated algorithm's *resolve* phase until the acknowledged messages appear in that node's view of the blackboard.

In the *resolve* phase of the updated x-sync, nodes do not broadcast views of their own blackboard. As any message broadcast by a good node during the *generate* phase will eventually be r-received by all good nodes, the resolve phase does not need to share messages again; all it has to do is delay the node's completion of the x-sync until the node can be sure it has r-received enough messages.

Because of this, during the *resolve* phase of the updated x-sync, each good node broadcasts a list of the messages they have r-received so far. Each good node j can be sure that, if they r-receive a message $message(i', j')$ from a node j', all previous messages broadcast by j' are also written to the blackboard (see Lemma 3.2.2). As such, nodes do not have to broadcast a list identifying every message they have r-received – just the identifier of the most recent message r-received from each node.

This holds whether or not j has r-received all of $message(1, j')$ through $message(i', j')$ yet; once a message is written to the blackboard, all good nodes (including j) will receive it before the x-sync concludes, so j can claim it has r-received these messages.

As with views of the blackboard in King-Saia's x-sync, a node j will not assist in the reliable broadcast of the message list of another node j' unless it has also r-received all messages that j' claims to have r-received in its list. As $n - t$ lists must be r-received for a node to finish the x-sync, this effectively acts to keep each good node from ending the algorithm until the node can be sure that all messages that are written to the blackboard will be r-received by each good node. This also ensures that if a message is r-received by $n - 2t$ processors, it will appear in every good node's view of the blackboard once the algorithm is over, through Lemma 3.2.1.

Chapter 3

Proof

In this section, we prove that the updated algorithm is a valid x-sync.

3.1 Halting

In order to be a proper x-sync algorithm, all good processors must terminate, which this subsection proves.

Lemma 3.1.1. *One good processor will eventually leave the while loop.*

Proof. Suppose no good processor ever leaves the while loop. For each processor j, let i_j be the largest i set by that processor, and assume that all good processors have reached a state where no good processor increments their i_j. As such, one of two situations must be the case:

1. Some good processor j has $i_j < x$. In this case, j has reliably broadcast $message(i_j, j)$. Either $i_j = 1$, or j has r-received $n - t$ acknowledgements for $message(i_{j-1}, j)$. In the latter case, all other good processors will r-receive these acknowledgements. As all good processors are still taking part in the while loop, they will participate in the reliable broadcast of $message(i_j, j)$, leading to its eventual

r-receipt by all $n - t$ good processors. This will cause those processors to reliably broadcast acknowledgements, which j will r-receive. j's r-receipt of $n - t$ acknowledgements for $message(i_j, j)$ will cause j to increment i_j. This is a contradiction.

2. Every good processor j has $i_j = x$. In this case, all $n - t$ good processors have generated and reliably broadcast their own $message(x, j')$. By reliable broadcast, all good nodes (which are still in the while loop) will eventually r-receive these messages, causing them to reliably broadcast acknowledgements for the $n - t$ $message(x, j')$'s broadcast by good nodes. Once these acknowledgements are broadcast, all good nodes will (by reliable broadcast) eventually r-receive all of them (as we assume all good nodes are stuck in the while loop). The first good processor that r-receives all acknowledgements that were sent by the $n - t$ good processors for the $n - t$ $message(x, j')$'s will have fulfilled its condition to leave the while loop (a contradiction). □

Lemma 3.1.2. *If one good processor leaves the while loop, so do all other good processors.*

Proof. For a good processor to leave the loop, it must have r-received $n - t$ acknowledgements for the last element of each of $n-t$ columns in the blackboard. As one good processor has r-received these acknowledgements, so will all other good processors. So, all other good processors will eventually leave the while loop. □

Lemma 3.1.3. *Every good processor that leaves the while loop will terminate.*

Proof. Choose any good processor j that leaves the while loop. This processor will broadcast its $list_j$ in step 9A of the algorithm. When processor j broadcasts its list $list_j$, the list acts to enumerate each $message$ that j had r-received before the broadcast. Every other good processor will eventually r-receive those messages, and thus also be able to participate in the reliable broadcast of $list_j$ in step 9B. This means that every good processor will eventually r-receive $list_j$.

As every good processor leaves the while loop, every good processor j' will reliably broadcast its own $list_{j'}$, and all good processors will r-receive $n-t$ lists $list'_{j'}$. This fulfills every good processor's condition to end the algorithm (having r-received $n-t$ *list*'s). □

3.2 Correctness

This subsection proves that the algorithm upholds the properties of an x-sync.

Lemma 3.2.1. *If $message(i,j)$ is r-received by at least $n-2t$ good processors before they start step 9, then $message(i,j)$ is written to the blackboard (all good nodes have this message in their view of the blackboard before the algorithm ends).*

Proof. A $message(i,j)$ is r-received by a set S of at least $n-2t$ good processors, each of which r-receives the message before it starts step 9. Each member j' of S will have $list_{j'}(j) \geq i$ in step 9A. At most t good processors are not in S.

Once any good processor p r-receives $n-t$ lists in step 10, it must have r-received at least $n-2t$ lists from good processors. At least one of those good processors must be in S. If this were not the case, there would be two sets; S, and another set R of $n-2t$ good processors that had sent lists but were not in S. However, this is impossible; at most t good processors are not in S, and R is guaranteed to be of size at least $t+1$.

In order to have r-received the list from the good processor in S, p must (by step 9B) have independently r-received $message(i,j)$. This applies for every good processor p, so $message(i,j)$ is written to the blackboard. □

Lemma 3.2.2. *In the blackboard, all entries in a column that follow an ambiguous entry are null, and all entries that precede it are nonnull and nonambiguous.*

Proof. For any processor j', let i' be the smallest value where $message(i', j')$ is *not* written to the blackboard (that is, the associated blackboard cell is ambiguous or null, because some good node does not have the message in its view of the blackboard once the algorithm ends). By the contrapositive of Lemma 3.2.1, fewer than $n - 2t$ good processors r-received $message(i', j')$ before they begin step 9.

As such, fewer than $n-2t$ good processors reliably broadcast an acknowledgement for $message(i', j')$ before step 9, which means that no good processor will r-receive $n - t$ acknowledgements for $message(i', j')$. This means that (by Step 3B) no good node will participate in the reliable broadcast of $message(i' + 1, j')$, preventing it from being r-received by any good node. Thus, the blackboard cell associated with $message(i' + 1, j')$ is null, and so is the cell associated with any $message(i' + n, j')$ after it (as that *message* will not be r-received by any good node either). □

Lemma 3.2.3. *At least $n - t$ full columns of the blackboard have no entries that are ambiguous or null.*

Proof. Any good processor only exits the while loop once they have r-received $n - t$ acknowledgements $acknowledge(x, j')$ for each of $n - t$ different processors j'.

Consider the first processor to exit the while loop, j_e. The exiting processor j_e will only r-receive this many acknowledgements for $message(x, j')$ for a processor j' if $n-2t$ good processors have r-received $message(x, j')$ while they are in the while loop, which means that by Lemma 3.2.1, $message(x, j')$ is written to the blackboard. By Lemma 3.2.2, this means $message(i', j')$ is also written to the blackboard, for any i'. Therefore, the column j' of the blackboard has no entries that are ambiguous or null.

As there are at least $n - t$ such processors j', each of their complete columns will be in the final blackboard. □

Lemma 3.2.4. *For all $1 < i \leq x$, each good processor j emits $message(i, j)$ only after $message(i-1, j)$ is written to the blackboard.*

Proof. For $i > 1$, a good processor, j, will only generate a $message(i, j)$ after it has r-received $n - t$ acknowledgements for $message(i-1, j)$. So, at least $n - 2t$ good processors must have r-received $message(i-1, j)$, while they were in the while loop. By Lemma 3.2.1, this means that $message(i-1, j)$ is not ambiguous or null, and therefore $message(i-1, j)$ is written to the blackboard. □

Lemma 3.2.5. *If $t < n/3$, Algorithm 6 implements an x-sync in $O(x)$ time.*

Proof. Correctness follows from Lemmas 3.2.2, 3.2.3, and 3.2.4. Regarding time complexity, the x-sync algorithm's while loop consists of $O(x)$ broadcast rounds, where a *broadcast round* refers to the reliable broadcast of $message(i, j)$ and $acknowledge(i, j)$ for some i and all j. All messages reliably broadcast by good processors take O(1) time, and only good processors' broadcasts are required for a good processor to exit the while loop. As there are x rounds of $O(1)$ broadcasts, all good processors will exit the while loop in $O(x)$ time.

After each good processor exits the while loop, it reliably broadcasts one list, which takes $O(1)$ time to r-receive. As such, each good processor finishes the broadcast and r-receipt of lists (steps 9-11 of the algorithm) in $O(1)$ time.

All in all, no part of the algorithm takes more than O(x) time to complete.

□

Chapter 4

Implementation & Results

The modified King-Saia algorithm with updated blackboard algorithm was implemented in Python, using the Kombu networking library [11] with a RabbitMQ message-sending server [23]. Each node was represented as a separate Python process, as was the adversary when one was present. The message-sending server was represented as a RabbitMQ process on the same machine as the nodes.

We tested the algorithm on three primary testing environments. The first two, for initial testing, were a MacBook Pro with a 2.8 GHz 4-core Intel Core i7-4980HQ processor, and an iMac with a 2.7 GHz 4-core Intel Core i5-3330S processor. Both computers ran macOS 10.12.6. The third, for further testing including stress testing, was a PC running Windows 10 v1909, with a 2.9 GHz (running at 3.9 GHz) 6-core Intel Core i5-9400 processor.

The configuration used had 10 byzantine-agreement processes (so, $n = 10$ and $t = 3$) and one adversary process, as well as the message-sending server process. However, the number of byzantine-agreement processes could be scaled to test larger networks.

4.1 Differences Between the Design As Written and As Implemented

The nature of the Kombu networking library required some differences in the implementation as compared to the assumptions listed in the algorithm. Kombu uses a central server architecture; messages are sent to the server, which forwards the messages to the desired destination. The server is not ever affected by the adversary (see Section 4.2).

Kombu does not natively track the source of a message, so the implementation's networking code adds a "sender" attribute to each message's metadata, which is used by all nodes to determine the originator of a message. This attribute is not ever altered by the adversary.

Kombu allows what are known as "broadcast exchanges", where a message sent to the exchange is copied to every node that has subscribed to the exchange. This is used for convenience when sending a single message to all other processors (i.e. all sending steps of reliable broadcast), but it does not remove the implied requirement for every processor to know the number of other processors (and individual identifiers for each).

Due to time constraints, the "Process-Epoch" step of King-Saia's original Byzantine agreement algorithm (which, after several sets of Modified-Bracha waves with no decision, identifies processes that are likely adversarial and prevents them from contributing to future blackboards) was not implemented.

4.1.1 Validation in Bracha and Modified-Bracha

Bracha's Byzantine agreement and Modified-Bracha both state to continue their phases until a number of messages have been "validated". This is taken to mean that the validating processor believes that the message *could have* been sent by a

good node, given what the validating processor is aware of about messages that have been r-received. However, both Bracha and King-Saia don't give implementation details of validation. In our implementation, validation places the following checks on incoming Bracha messages:

- Wave 1 messages do not have any validation checks, as a node could broadcast a Wave 1 message with any value. It might be possible to add further validation based on the results of the most recent iteration of Modified-Bracha, if one exists.

- Wave 2 messages are checked to ensure that more than $\frac{n-t}{2}$ Wave 1 messages with the same value as the Wave 2 message have been r-received. If this is indeterminate due to some number of Wave 1 messages having not yet arrived, the Wave 2 message will be held to wait for a sufficient number of Wave 1 messages, as long as this is possible.

- Wave 3 messages are treated differently based on whether they carry the *decide* flag or not.

    - If a Wave 3 message has the *decide* flag, it's treated similarly to a Wave 2 message; this type of Wave 3 message is valid when more than $\frac{n}{2}$ r-received Wave 2 messages have the same value as the Wave 3 message. Deciding Wave 3 messages that cannot yet be validated will be held until more than $\frac{n}{2}$ Wave 2 messages with the same value are r-received; once this happens, the Wave 3 message in question will clearly be either valid if the values match, or invalid otherwise.

- If a Wave 3 message does not have the *decide* flag, it means that the sender of that message did not change its value after Wave 2; so, the current message's value must match the value of the sender's Wave 2 message. In addition, of the n total Wave 2 messages that can hypothetically be r-received, there must not be a supermajority of more than $\frac{n}{2} + t$ or more messages of *either* value; otherwise, the node that sent the message would have r-received more than $\frac{n-t}{2}$ Wave 2 messages with the supermajority value, causing the Wave 3 message's *decide* flag to be set. Non-deciding Wave 3 messages will be held until this condition can be verified.

Algorithm 7 Validation for messages received by Modified-Bracha

```
1: v ← the message's value.
2: if message is a Wave 1 message then
3:     Message is valid.
4: end if

5: if message is a Wave 2 message then
6:     m_1v ← the number of Wave 1 messages with value v that have been r-received.
7:     s_1 ← the number of Wave 1 messages that have not yet been r-received.
8:     if m_1v > (n−t)/2 then
9:         Message is valid.
10:    else if m_1v + s_1 > (n−t)/2 then
11:        Message could be valid; hold it until more Wave 1 messages arrive and rerun
           this algorithm then.
12:    else
13:        Message is invalid; discard.
14:    end if
15: end if
```

cont'd next page

16: **if** message is a Wave 3 message **then**
17: $m_{2v} \leftarrow$ the number of Wave 2 messages with value v that have been r-received.
18: $m_{2\neg v} \leftarrow$ the number of Wave 2 messages with value $\neg v$ that have been r-received.
19: $s_2 \leftarrow$ the number of Wave 2 messages that have not yet been r-received.

20: **if** message's *decide* flag is set **then**
21: **if** $m_{2v} > \frac{n}{2}$ **then**
22: Message is valid.
23: **else if** $m_{2v} + s_2 > \frac{n}{2}$ **then**
24: Message could be valid; hold it until more Wave 2 messages arrive and rerun this algorithm then.
25: **else**
26: Message is invalid; discard.
27: **end if**
28: **end if**

29: **if** message's *decide* flag is not set **then**
30: **if** the Wave 2 message sent earlier by the sender of this message does not have the same value **then**
31: Message is invalid; discard.
32: **end if**
33: **if** $m_{2v} > \frac{n}{2} + t$ **or** $m_{2\neg v} > \frac{n}{2} + t$ **then**
34: Message is invalid; enough Wave 2 messages of one value exist to make it impossible for the *decide* flag not to be set. Discard.
35: **else if** $m_{2v} + s_2 > \frac{n}{2} + t$ **or** $m_{2\neg v} + s_2 > \frac{n}{2} + t$ **then**
36: Message could be valid; hold it until more Wave 2 messages arrive and rerun this algorithm then.
37: **else**
38: Message is valid.
39: **end if**
40: **end if**
41: **end if**

4.2 Differences Between the Adversary As Written and As Implemented

In the algorithm, the adversary is assumed to be able to take over any process (up to its limit of t), controlling its actions from then on. It is also able to eavesdrop on any message in transit and delay it. Having the adversary actually take over other Python processes and eavesdrop on / delay messages was rejected as unfeasible; instead, we decided to simulate this in the design.

In the simulation of modified King-Saia where an adversary is present, each node sends each message it is about to reliably broadcast to the adversary; the adversary may respond to the sending node with a command that indicates that node is now corrupted, along with a detailed plan of the behavior that node is to perform. Doing this counts as corrupting the node for the purposes of the adversary's limits. If the adversary decides not to corrupt a node, it will pass the message on to the destination. Good nodes don't know which nodes are 'corrupted', and corrupted nodes don't know which other nodes are corrupted - only the adversary knows its full roster.

In addition to this, when an adversary is present, each node sends each message it is about to r-receive to the adversary; the adversary cannot alter the message that is about to be r-received, but it can hold that message for an indefinite period of time. This does not count as corrupting the node, and simulates the theoretical adversary's control over network scheduling.

4.2.1 Adversary Behaviors: Modified-Bracha

In order to test the adversary, we came up with several ways the adversary can attempt to get a desired result out of a phase of the agreement algorithm, whether that phase is Modified-Bracha, Global-Coin, or both. The behaviors in this section

all focus on influencing the result of Modified-Bracha in some way. In addition to a naive approach, we came up with behaviors to induce each of the possible outcomes of Modified-Bracha: to make nodes decide, or choose to run Global-Coin, either setting their value to the result of Global-Coin or not. Note that these are *behaviors*, not *attacks*: several behaviors might have to be combined in order for the adversary to succeed at disrupting Byzantine agreement. (A demonstration of a full attack on modified King-Saia is found in section 4.3.3.)

With the exception of the "Naive Value Changing" behavior, which is not guaranteed to succeed, every behavior has a precondition necessary for the adversary to arrange conditions to its liking. The spirit of these preconditions can be boiled down to one observation: if every good node starts a run of Modified-Bracha with the same initial value, then any attempt at adversarial interference will fail; the adversary will not have the weight of numbers necessary to influence nodes.

Modified-Bracha: Naive Value Changing

In this behavior (Algorithm 8), the adversary corrupts as many nodes as it can. During Modified-Bracha, each corrupted node always insists its current value is the adversary's chosen value, regardless of whether this is possible under the validation rules and current state of the Modified-Bracha instance. In the third wave of Modified-Bracha, a corrupted node says whether or not it is deciding based on what it would say if it weren't corrupted.

Algorithm 8 Adversary Behavior: Modified-Bracha – Naive Value Changing

1: Corrupt as many nodes as possible.
2: During Modified-Bracha, whenever a corrupted node would broadcast its value, it broadcasts the adversary's chosen value instead.

The adversary does not use its control over network scheduling in this behavior, making it quite simple and lightweight. However, this behavior may fail if good nodes refuse to validate corrupted nodes' messages, or only validate them after a decision has been reached.

Modified-Bracha: Force Decide

In this behavior (Algorithm 9), the adversary forces all good nodes to decide on its chosen value once Modified-Bracha concludes. As a precondition, at least one good node must have the same initial value as the adversary's chosen value.

Algorithm 9 Adversary Behavior: Modified-Bracha – Force Decide

1: **Precondition:** A minimum g good nodes must have the same initial value as the adversary's chosen value (C), where $t + g > \frac{n-t}{2}$. When $n = 3t + 1$, $g = 1$.

Wave 1:

2: In the first wave of Modified-Bracha, corrupt nodes that would broadcast $\neg C$, and have them broadcast C instead. Continue until more than $\frac{n-t}{2}$ nodes (corrupted or not) have broadcast C as their initial value.

3: Through network scheduling, order message arrival so that all nodes r-receive every broadcast of C before every other broadcast. As each good node changes its value after r-receiving $n - t$ messages, the nodes will see a majority of Wave 1 messages with value C, and set their values to C.

Wave 2:

4: In the second wave, all good nodes will broadcast C naturally. Corrupted nodes also do so. All good nodes r-receive only Wave 2 messages with the value of C, and so change their value to $(C, decide)$.

Wave 3:

5: In the third wave, all good nodes will broadcast $(C, decide)$. Corrupted nodes do so as well.

6: All good nodes r-receive Wave 3 messages only of the form $(C, decide)$, and so decide on C.

Modified-Bracha: Force Coin Toss, Accepting Random Value

In this behavior (Algorithm 10), the adversary forces nodes to run Global-Coin once Modified-Bracha concludes, and to set their values to the result of Global-Coin. While this can result in agreement if done to all good nodes, doing it to a subset of good nodes can be part of more elaborate attacks. The algorithm description for the behavior gives the basic version (how to apply the behavior to all good nodes at once).

As a precondition, at least one good node must have the same initial value as the adversary's chosen value, and at least one good node must have the opposite initial value. More good nodes may be required for the precondition if $t < \lceil \frac{n}{3} \rceil - 1$.

Algorithm 10 Adversary Behavior: Modified-Bracha – Force Coin Toss, Accepting Random Value

1: **Precondition:** A minimum of g good nodes each must have the same initial value as the adversary's chosen value (C); also, a minimum of g good nodes must have the opposite value ($\neg C$). $t + g > \frac{n-t}{2}$ in both cases. When $n = 3t+1$, $g = 1$.

Wave 1:

2: In the first wave of Modified-Bracha, corrupt nodes and assign them to broadcast either C or $\neg C$, such that at least $\lfloor \frac{n-t}{2} \rfloor + 1$ nodes broadcast C, and at least the same number broadcast $\neg C$.

3: Through network scheduling, order message arrival so that half the nodes r-receive a majority of C messages in the first $n - t$ Wave 1 messages, and the other half of the nodes r-receive a majority of $\neg C$ messages.

Wave 2:

4: In the second wave, half the good nodes will broadcast C naturally, while the other half will broadcast $\neg C$. Corrupted nodes should broadcast C or $\neg C$ to ensure that roughly $\frac{n}{2}$ nodes broadcast each value in total.

5: Through network scheduling, ensure no node r-receives more than $\lfloor \frac{n}{2} \rfloor$ C Wave 2 messages, nor more than $\lfloor \frac{n}{2} \rfloor$ $\neg C$ messages. Leftover messages are held until all nodes begin Global-Coin.

cont'd next page

Wave 3:

6: In the third wave, no node r-receives a majority of Wave 2 messages, so all nodes indicate they will not decide when they broadcast their value. (Corrupted nodes behave as if they were good nodes.)

7: All good nodes r-receive only Wave 3 messages indicating there will be no decision, so they begin Global-Coin, and set their value to the result of Global-Coin.

Modified-Bracha: Force Coin Toss, Adversary Chooses Value

In this behavior (Algorithm 11), the adversary forces nodes to run Global-Coin once Modified-Bracha concludes, but ensures that each node ignores their view of the blackboard and sets their value to the adversary's chosen value. Similar to the "Force Coin Toss, Accepting Random Value" behavior, this results in agreement if done to all nodes. However, doing it to a subset of nodes can be part of more elaborate attacks. As before, the algorithm description for this behavior gives its basic version (how to apply the behavior to all good nodes at once).

This behavior's precondition is the same as the "Force Coin Toss, Accepting Random Value" behavior; at least one good node each must have the adversary's chosen value, and the opposite value, as initial values. More good nodes may be needed if $t < \lceil \frac{n}{3} \rceil - 1$.

4.2.2 Adversary Behaviors: Global-Coin

The next adversary behaviors focus on Global-Coin instead of Modified-Bracha. There is no way to validate the coinflip that a node claims to have made; as such, adversarial interference is easier to perform on a Global-Coin instance.

However, this comes at a cost: the Global-Coin behaviors could fail if the good nodes' coinflips, through sheer chance, create a blackboard sufficiently tilted in a certain direction that the adversary cannot influence it in its chosen direction.

Algorithm 11 Adversary Behavior: Modified-Bracha – Force Coin Toss, Adversary Chooses Value

1: **Precondition:** A minimum of g good nodes each must have the same initial value as the adversary's chosen value (C); also, a minimum of g good nodes must have the opposite value ($\neg C$). $t + g > \frac{n-t}{2}$ in both cases. When $n = 3t+1$, $g = 1$.

Wave 1:

2: In the first wave of Modified-Bracha, corrupt nodes and assign them to broadcast either C or $\neg C$, such that at least $\lfloor \frac{n-t}{2} \rfloor + 1$ nodes broadcast C, and at least the same number broadcast $\neg C$.

3: Through network scheduling, order message arrival so that at least $\lfloor \frac{n}{2} \rfloor + 1$ nodes r-receive a majority of C messages in the first $n-t$ Wave 1 messages, and at least $n - t - \frac{n}{2}$ nodes r-receive a majority of $\neg C$ messages. Nodes chosen to r-receive messages in this way should be preferably good, if good nodes not yet chosen exist.

Wave 2:

4: In the second wave, the good nodes that r-received a majority of Wave 1 C messages will broadcast C naturally, while the rest will broadcast $\neg C$. Corrupted nodes behave as if they are good nodes.

5: Through network scheduling, ensure that between $t+1$ to $2t$ good nodes r-receive more than $\lfloor \frac{n}{2} \rfloor$ C Wave 2 messages. Ensure that all other nodes r-receive at most $\lfloor \frac{n}{2} \rfloor$ C Wave 2 messages, and at most $\lfloor \frac{n}{2} \rfloor$ $\neg C$ messages.

Wave 3:

6: In the third wave, the good nodes that r-received over $\lfloor \frac{n}{2} \rfloor$ Wave 2 C messages will broadcast (C, decide) naturally. All other good nodes will broadcast messages saying they are not deciding. Corrupted nodes act as though they are not deciding.

7: Through network scheduling, ensure that all good nodes r-receive between $t + 1$ and $2t$ (C, decide) Wave 3 messages. Each good node will then begin Global-Coin, but ignore its result and set their value to C.

(King-Saia provides an additional solution to the lack of direct validation of coin-flips. The third part of the King-Saia algorithm, Process-Epoch, focuses on detecting adversaries that are influencing Global-Coin in a probabilistic way; put simply, for the adversary to continue to influence succeeding instances of Global-Coin, it must

eventually act in a manner such that its corrupted nodes can be identified by a lack of randomness in their behavior. Then, these nodes' influence can be excluded from the blackboard. [16])

In each behavior, the adversary has two methods it can use to sway the result of Global-Coin. The first is that the adversary can "see into the future" slightly: once a node makes a coinflip, but before it broadcasts it, the adversary is aware of it, and can choose to indefinitely delay the broadcast of the flip. This prevents the node from making further flips, but can be used to prevent an undesirable single flip from becoming written to the blackboard.

The second method the adversary has available to it is absolute control over what flips its corrupted nodes broadcast. The only thing it needs to be careful of is the check in Global-Coin (Algorithm 4): if a node broadcasts coin flips summing to an absolute value of $5\sqrt{n \ln n}$ or more, good nodes will ignore it.

Global-Coin: Biased Coin

In this behavior (Algorithm 12), the adversary attempts to sway the shared coin flip's result to its chosen value, C.

To do this, the adversary forecasts what coin flips each good node is about to broadcast. If a node is about to broadcast C, it lets it through. If a node is about to broadcast $\neg C$, it holds it. This continues until all good nodes are about to broadcast $\neg C$; this indicates the adversary has wrung all the influence from this method it can.

After that, the adversary lets the $n - 2t$ good nodes that emitted the most C coin flips build full columns of flips, and has its corrupted nodes supply full columns of flips as well – with a sufficient bias to ensure that the total sum of the shared coinflip falls in the direction of C.

Algorithm 12 Adversary Behavior: Global-Coin – Biased Coin

1: Through network scheduling, only allow good nodes to broadcast flips with value C, until no more good nodes are in a position to do so. Hold all other messages.
2: Once every good node is about to broadcast $\neg C$ (that is, all messages are held), allow the $n - 2t$ good nodes that have broadcast the most flips to broadcast messages freely.
3: Corrupted nodes broadcast full columns of coinflips, with values such that the resulting sum of the blackboard is in the direction of C. (Subject to limitations on the number of flips each node can broadcast, and the integrity check performed by Global-Coin.)

Global-Coin: Split Coin

In this behavior (Algorithm 13), the adversary attempts to alter the coin's result such that different good nodes believe that Global-Coin has settled on different values C and $\neg C$.

To do this, the adversary must first "center" the blackboard - ensure flips are written to it such that each node receives flips whose sum is exactly zero. To do this, the adversary uses its forecasting and hold capabilities to only let coinflips be broadcast in pairs of C and $\neg C$. (If every good node is about to broadcast C or $\neg C$, the adversary uses a flip from one of its corrupted nodes to even things out.)

Once the blackboard is centered and there are $n - t$ full columns, the adversary can then prepare two messages that will become ambiguous in the blackboard: one in the direction of C, the other in the direction of $\neg C$.

Through network scheduling, the adversary requires that nodes it wishes to settle on C r-receive the C message, while nodes it wishes to settle on $\neg C$ r-receive the $\neg C$ message. These good nodes come away with the conclusion that Global-Coin has finished, in the adversary's chosen direction.

Algorithm 13 Adversary Behavior: Global-Coin – Split Coin

1: Through network scheduling, ensure good nodes broadcast flips in pairs; as one node broadcasts C, allow another node to broadcast $\neg C$. Hold all other messages.
2: If every good node would broadcast C or $\neg C$, let one good node broadcast, and compensate by having a corrupted node broadcast the opposite value.
3: Continue the previous steps until at least $n - t$ full columns of messages are broadcast. When good nodes would broadcast coin lists, hold them for later.
4: Two corrupted nodes, cn_1 and cn_2, broadcast one coinflip each. cn_1 broadcasts a C coinflip. cn_2 broadcasts a $\neg C$ coinflip. Through network scheduling, the adversary holds these coinflips.
5: cn_1 broadcasts a coin list that includes their most recent coinflip and all other messages broadcast so far, *except* cn_2's most recent message. cn_2 does the same, except its coin list excludes cn_1's most recent message. Through network scheduling, the adversary holds these coin lists.
6: If the adversary wants a good node to conclude the shared coinflip result is C, it allows that node to r-receive cn_1's message and coin list, as well as coin lists from $n - t - 1$ good nodes. If the adversary wants a good node to conclude the shared coinflip result is $\neg C$, it allows that node to r-receive cn_2's message and coin list instead (along with $n - t - 1$ good coin lists).
7: In order for the adversary to have a good node r-receive the coin flip and coin list from cn_1 xor cn_2, at least $n - 2t$ good nodes must each participate in the reliable broadcast of these messages. As such, there must be at least one good node which receives the coin flip and coin list messages from both cn_1 *and* cn_2, along with $n - t - 2$ coin lists from good nodes.
8: Once the chosen good nodes r-receive cn_1 xor cn_2's coin flips and complete the updated x-sync, they will conclude that the blackboard's result is in the direction of C or $\neg C$, as the adversary intended. Good nodes that r-receive both cn_1 and cn_2's coin flips will behave as if the blackboard is exactly balanced.

This behavior fails if the good nodes' result is sufficiently biased that the adversary's attempts to center the blackboard are unsuccessful. Typically, this will take the form of the adversary using up all its corrupted nodes' flips during the centering process.

4.2.3 A Full Adversary Attack: Deadlock

In this section, we describe an example of how the behaviors listed above can be combined into a full attack on King-Saia. In this attack, the adversary attempts to arrange matters so that each iteration of Modified-Bracha and Global-Coin ends with at least one good node changing their value to C, and at least one other good node changing their value to $\neg C$, indefinitely (creating a deadlock and preventing Modified-Bracha from completing).

Modified-Bracha: Split Results

In the first section of the deadlock attack, the adversary runs a variation of the "Modified-Bracha – Force Coin Toss, Adversary Chooses Value" behavior (Algorithm 11). However, instead of using network scheduling to ensure that every good node r-receives $t+1$ to $2t$ messages of the type $(C, decide)$, the adversary uses it to ensure that some good nodes r-receive between $t+1$ and $2t$ $(C, decide)$ Wave 3 messages, and other good nodes r-receive only t or fewer of such messages. This splits the body of good nodes into two populations: both populations will run Global-Coin, but only the population that r-received few $(C, decide)$ messages will accept the shared coin flip's value. This gives the adversary its chance to force the round to end inconclusively.

Global-Coin: "Acting Against Its Own Interests"

In the second section of the deadlock attack, the adversary attempts to bias the result of Global-Coin as per the "Global-Coin: Biased Coin" behavior (Algorithm 12). However, the adversary tries to push the shared coinflip in the direction of $\neg C$, *not* C.

If the adversary succeeds at affecting the coinflip, the good nodes will end up with a mix of C and $\neg C$ for initial values at the start of the next set of Modified-Bracha

waves. This lets the adversary repeat the attack in that round.

This continues indefinitely until Process-Epoch runs (and the adversary risks being discovered, and its corrupted nodes shunned), or until the adversary can't successfully influence a particular instance of Global-Coin.

4.3 Results

4.3.1 Modified-Bracha

Bracha's Byzantine agreement has the property that, if all good nodes are unanimous in their value at the start of a set of waves, the adversary will not be able to prevent all nodes from deciding on a value [5]. After testing, it is clear that Modified-Bracha shares this property.

In the problem as stated, the adversary cannot prevent delivery of a message between two good nodes. In our implementation, messages that the adversary decides to delay are put into a hold queue, to be released when the experimenter decides to do so (typically after other activity between nodes has died down).

When Modified-Bracha was tested with good nodes with unanimous initial values (100 runs, with a mix of adversary behaviors), the result was always the same; either all good nodes decided right away, or the algorithm paused while messages were delayed, then all good nodes decided on the correct initial value once the queue was cleared.

In order to test the adversarial behaviors, we also tested Modified-Bracha (100 runs per behavior) with nodes that were not initially unanimous in their initial values. Here, the adversary was successfully able to bring about their desired results in each set of three waves. This behavior matches how Bracha's Byzantine agreement responds to adversarial interference.

We conclude that Modified-Bracha retains the same resistance to an adversary that Bracha's Byzantine agreement does.

4.3.2 Global-Coin

The properties required of an x-sync algorithm are that each node emits values to the "blackboard" in order, that the blackboard has $n - t$ full columns of values on completion, and that at most one value in each non-full column of values is ambiguous, with the values before it valid and nonambiguous and the values after it null. The updated x-sync fulfills all of these properties; there was no time during testing when the adversary (or random chance) was able to cause any of these properties to not hold.

However, no such property exists to prevent the adversary interfering with the shared coin flip created through the x-sync. When we tested the "Biased Coin" behavior, it was successfully able to alter the resulting shared coin flip every time Global-Coin was run (100 runs total). The "Split Coin" behavior was not tested due to time constraints.

Notably, when testing Global-Coin without adversarial interference of the blackboard (a total of 100 runs), Global-Coin was able to yield an *unanimous* shared coin flip (that is, every good node's view of the blackboard yielded a sum with the same sign) in every test run. This suggests that, in the absence of adversarial interference, Global-Coin will typically yield unanimous shared coin flips.

4.3.3 "Deadlock" Attack

After the success of the "Biased Coin" behavior in swaying the shared coin flip, we tested the "Deadlock" attack on Modified-Bracha and Global-Coin. The behavior was successful in putting the algorithm into a loop of indefinite duration; our tests

ran continuously for up to 40 Modified-Bracha iterations before we halted them.

However, it is worth noting that we did not implement the "Process-Epoch" step of the original King-Saia algorithm. This step would have run after each 10-20 iterations in our test environment (King-Saia specifies every $c * n$ iterations, where c is a constant), and (if fully implemented) would have a chance to identify and ignore the adversary's corrupted nodes. Instead, our "stub" version of Process-Epoch did not take any action.

4.3.4 Expected Number of Iterations Before Decision

The Deadlock attack inspired an additional question during testing: how many iterations of Modified-Bracha will run before a decision is reached under typical circumstances? When the adversary does not attempt to affect the process, this number appears to be 1-2 iterations; of 100 test runs without adversarial interference, 92 had all nodes decide after a single iteration. In the remaining 7 runs, all nodes decided after 2 iterations of Modified-Bracha (and after 1 run of Global-Coin, as Global-Coin returned an unanimous shared coin flip in each).

The answer is simpler for certain adversarial behaviors. If the adversary uses the "Force Decide" behavior, all good nodes are forced to decide after a single iteration as long as the precondition for the behavior is met. In the case of the "Force Coin Toss, Accepting Random Value" and "Force Coin Toss, Adversary Chooses Value" behaviors, all good nodes decide after 2 iterations unless the adversary is also deliberately trying to split the result of Global-Coin; in the absence of this, either all good nodes accept Global-Coin's value, or all good nodes run Global-Coin but hold their current value. (The latter case corresponds to Case 8B of Bracha's Reliable Broadcast; Bracha proves that all nodes that enter this case in the same iteration will have the same value. [5]) In either of these outcomes, all good nodes will start with unanimous initial

Iterations to Decision	2	3	4	5	6	7	8	9
# Runs	51	20	10	13	2	2	1	1

Table 4.1: Number of iterations before a decision was reached, in 100 test runs of variant "Force Coin Toss" behavior.

values at the start of the second iteration of Modified-Bracha, forcing them to decide.

This leaves only the "Naive Value Changing" behavior and the variant of "Force Coin Toss" used by the Deadlock attack. In 100 test runs of the "Naive Value Changing" behavior, all good nodes decided after one iteration in 91 runs out of the total. In the remaining 9 runs, all good nodes decided after 2 iterations.

The variant "Force Coin Toss" behavior was tested without any adversarial influence on the blackboard (to prevent the behavior from turning into the 'Deadlock' attack). In the behavior, the chance of a node deciding on any given iteration after the first is 50%; this is the chance that the value returned in the shared coin flip will match the value of nodes that participated in Global-Coin, but held their previous value. Results appear to bear this property out; of 100 test runs, 51 had all good nodes decide in 2 iterations, and 20 had all good nodes decide in 3 iterations. Full results for these tests are shown in Table 4.1.

Of all the conditions tested, only the variant "Force Coin Toss" behavior (which was a prelude to the Deadlock attack) was able to keep all good nodes from deciding within 2 iterations. This suggests that, outside of deliberate and specific adversarial interference, Modified-Bracha and Global-Coin can withstand most types of faults.

4.3.5 Blackboard Simulation: How Likely is the Adversary to Succeed?

After the success of the "Biased Coin" behavior and "Deadlock" attack tests, we created a program that performed a Monte Carlo simulation of a blackboard, being filled as if the adversary were using the "Biased Coin" behavior. The purpose of this was to determine how likely it is for an adversary to be (un)able to bias a shared coin flip. The program tested blackboards of between 10 to 1000 simulated nodes (10^1 to 10^3), doing 1 million simulations of each blackboard size. In addition, separate simulation runs were done for the condition of adversary choosing which nodes to corrupt during the middle of the shared coin flip (which would typically happen after the first iteration of a Modified-Bracha instance), versus the condition of the adversary already having chosen corrupted nodes before the shared coin flip begins (which would usually be the case on second and later iterations of a Modified-Bracha instance).

By default, an adversary that does absolutely nothing will "win" 50% of blackboards, as an unbiased shared coin flip will fall in their favor half the time. However, in the Monte Carlo simulation, the adversary was able to win *all but 1* of the blackboards simulated (approx. 1 failure out of 18 million runs total). Moreover, roughly 50%-65% of the simulated blackboard runs (the exact percentage for each group of 1 million runs depended on blackboard size and whether the adversary got to choose which nodes to corrupt; see Figure 4.1a) did not require the adversary to apply bias with the nodes it chose to corrupt; that is, in those runs, corrupted nodes output sets of randomly generated fair coin flips. Effectively, in these runs, the adversary was successful solely using its network scheduling ability; this result may be useful for the problem of distributed consensus, in which the adversary cannot make nodes emit faulty messages.

(a) **Chance of Adversary Winning a Shared Coin Flip Without Introducing Biased Coin Flips**

(b) **Additional Chance of Adversary Winning a Shared Coin Flip With Biased Coin Flips Summing to 0**

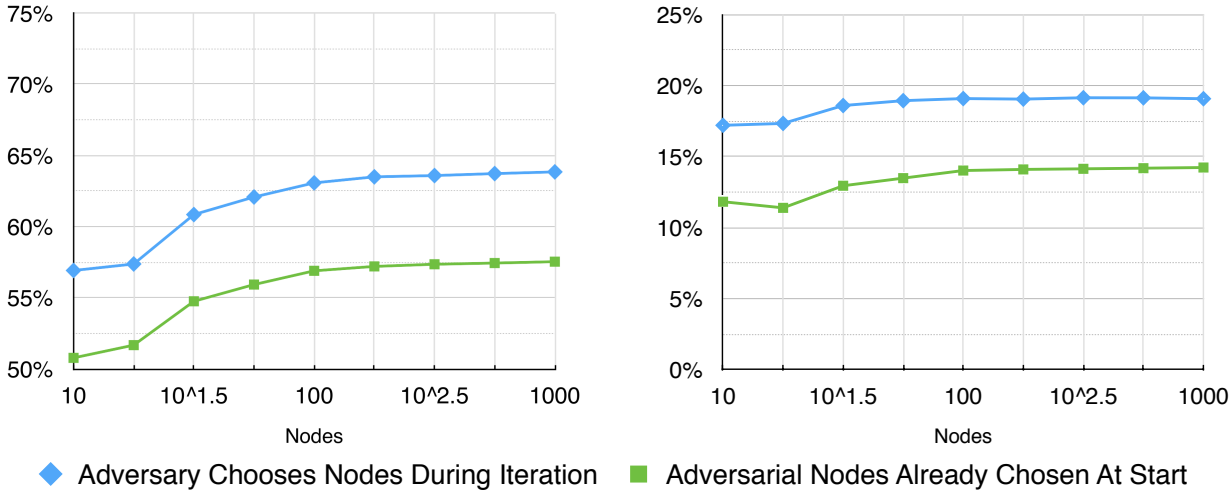

Figure 4.1: Results of Monte Carlo blackboard simulation

As well, in another 10%-20% of the simulated blackboard runs, the adversary was able to succeed by emitting biased coin flips, but in such a way that the sum of the flips of corrupted nodes was 0. Biased coin flips emitted in this way will not be detected by King and Saia's Process-Epoch, which only evaluates the sum of each node's coin flips in each iteration. [16] Figure 4.1b shows the percentage of runs in which the adversary failed to win with fair coin flips, but was able to win with biased flips summing to 0.

At no time during any simulation did any simulated node, good or corrupted, get disqualified from the blackboard for hitting the $5\sqrt{n \ln n}$ boundary noted in Algorithm 4. These results highlight the importance of the "Process-Epoch" step of King-Saia's algorithm, which identifies probably-corrupted nodes and ignores their contributions to future blackboards.

Chapter 5

Future Research & Conclusion

5.1 Implementing Process-Epoch

The results of testing the updated King-Saia algorithm – and in particular, the apparent power of the Deadlock attack to stall the process indefinitely by influencing shared coin flips – clearly show the need to implement the Process-Epoch section of the algorithm, to identify and blacklist probably-malicious nodes.

However, implementing this section is a non-trivial task. King-Saia 2016 lists two potential variants of the Process-Epoch section, one of which runs in expected polynomial time, the other of which runs in exponential time.

While other parts of King-Saia have a resilience of $t < n/3$, the expected-polynomial-time version of Process-Epoch has a resilience of $t \leq 1.14 * 10^{-9}n$ [17], rendering it unfeasible for applications where significant adversarial interference is expected.

The algorithm description of the exponential time version of Process-Epoch, while complete, requires finding a set of processors S_p and a set of iterations S_i (of sufficient size, for the iteration set) where each processor in S_p had a strongly biased result during the shared coin flip section of each iteration in S_i. No subalgorithm is described

in King-Saia for finding these sets, and while a trivial algorithm does exist (trying every possible combination of iterations and nodes), the problem of coming up with an efficient algorithm remains non-trivial.

5.2 Code Improvements

5.2.1 Distributed Implementation

At present, the updated King-Saia algorithm's implementation is set up for simulation on a single machine [15]. However, the network functions of the implementation are designed to be modular; they can be replaced without needing to edit the rest of the code.

In order to create a distributed implementation, both for purposes of more realistic testing and for actual applications of the algorithm, the network functions of the implementation would need to be replaced with versions which connect to other nodes in a distributed system.

5.2.2 Efficiency Enhancements

The current implementation of the updated King-Saia algorithm is written in Python [15]. While this allows for ease of prototyping, as well as access to convenience features such as dynamic typing, it seems possible that the algorithm could be sped up by reimplementing it in a lower-level language, such as C++.

5.3 Performance Testing

Once an fast, distributed implementation of the updated King-Saia algorithm exists, an important avenue of future research would be to test the algorithm's performance

over a network of computers. In particular, one useful potential topic would be to determine if (in realistic conditions) the algorithm is CPU-bound or I/O-bound, especially considering the large number of acknowledgement messages that are created and sent using reliable broadcast (not to mention the overhead of reliable broadcast itself).

5.4 Implementing Multi-Valued Byzantine Agreement Over Single-Valued Byzantine Agreement

Cachin *et al.* and Correia *et al.* have both proposed algorithms for multi-valued ('validated') Byzantine agreement. Both algorithm designs use single-valued (binary) Byzantine agreement as a subalgorithm [7, 12]. However, while few single-valued Byzantine agreement implementions exist, even fewer exist for multi-valued Byzantine agreement; SINTRA's implementation is the only one even slightly well-known [8].

Cachin *et al.* and Correia *et al.*'s algorithms have not been implemented. Given that both use Byzantine agreement as a subalgorithm, implementing either design, using the updated King-Saia algorithm as a single-valued Byzantine agreement subalgorithm, could provide an important contribution to the field of Byzantine agreement implementations.

5.5 Applications: Computing Over Mesh Networks

Byzantine agreement algorithms (especially multi-valued Byzantine agreement algorithms) are well-suited for applications where there are many discrete processors, none are considered trustworthy, and processors communicate directly to one another. One example of an ideal scenario is performing distributed computation over wireless mesh networks, such as a network of mobile devices (cellphones or similar).

To date, only basic research has been performed in this area. Cebe *et al.* created a network of Raspberry Pi devices to test the performance of permissioned blockchain algorithms (which use Byzantine agreement as a component), coming up with a new network protocol suited to the application [10]. As mobile phones and smart devices continue to gain popularity, applications for distributed device-based computing will likely increase in relevance and importance.

5.6 Conclusion

We presented a partial implementation of King and Saia's 2016 algorithm for Byzantine agreement in expected polynomial time. King and Saia's algorithm is based on Bracha's Byzantine agreement, but adds an interactive consistency algorithm, known as Global-Coin, to generate a shared coin flip more quickly, as well as Process-Epoch, a way of identifying and blacklisting probably-adversarial nodes. We also presented an update to Global-Coin which increases its resilience from $t < n/4$ to $t < n/3$. We implemented the majority King-Saia algorithm, as well as an adversary designed to attack it. Then, we tested the implementation, both under normal conditions and against a variety of adversarial behaviors and attacks. Under most conditions, all good nodes would decide within two iterations. For one particularly successful adversarial attack ('Deadlock'), we created a Monte Carlo simulation to determine the adversary's chance of successfully influencing the shared coin flip; the adversary was always successful. This result showed the importance of implementing the Process-Epoch portion of the King-Saia algorithm. Results of testing also led to possible improvements, including reimplementing the algorithm for efficiency and distributed operation, implementing multi-valued Byzantine agreement using King-Saia as a sub-algorithm, and applications involving distributed computation over mesh networks.

Bibliography

[1] Agrawal, S., and Daudjee, K. A performance comparison of algorithms for byzantine agreement in distributed systems. In *2016 12th European Dependable Computing Conference (EDCC)* (Sep. 2016), pp. 249–260. https://doi.org/10.1109/EDCC.2016.17.

[2] Ben-Or, M. Another advantage of free choice (extended abstract): Completely asynchronous agreement protocols. In *Proceedings of the Second Annual ACM Symposium on Principles of Distributed Computing* (New York, NY, USA, 1983), PODC 83, Association for Computing Machinery, pp. 27–30. https://doi.org/10.1145/800221.806707.

[3] Ben-Or, M., and El-Yaniv, R. Resilient-optimal interactive consistency in constant time. *Distributed Computing 16*, 4 (Dec 2003), 249–262. https://doi.org/10.1007/s00446-002-0083-3.

[4] Ben-Or, M., Pavlov, E., and Vaikuntanathan, V. Byzantine agreement in the full-information model in o(log n) rounds. In *Proceedings of the Thirty-Eighth Annual ACM Symposium on Theory of Computing* (New York, NY, USA, 2006), STOC 06, Association for Computing Machinery, pp. 179–186. https://doi.org/10.1145/1132516.1132543.

[5] Bracha, G. Asynchronous byzantine agreement protocols. *Information and Computation 75*, 2 (1987), 130–143. https://doi.org/10.1016/0890-5401(87)90054-X.

[6] Braud-Santoni, N., Guerraoui, R., and Huc, F. Fast byzantine agreement. In *Proceedings of the 2013 ACM Symposium on Principles of Distributed Computing* (New York, NY, USA, 2013), PODC 13, Association for Computing Machinery, pp. 57–64. https://doi.org/10.1145/2484239.2484243.

[7] Cachin, C., Kursawe, K., Petzold, F., and Shoup, V. Secure and efficient asynchronous broadcast protocols. In *Advances in Cryptology — CRYPTO 2001* (Berlin, Heidelberg, 2001), J. Kilian, Ed., Springer Berlin Heidelberg, pp. 524–541. https://doi.org/10.1007/3-540-44647-8_31.

[8] Cachin, C., and Poritz, J. A. Secure INtrusion-Tolerant Replication on the internet. In *Proceedings International Conference on Dependable Systems and Networks* (June 2002), pp. 167–176. https://doi.org/10.1109/DSN.2002.1028897.

[9] Castro, M., Liskov, B., et al. Practical byzantine fault tolerance. In *OSDI* (1999), vol. 99, pp. 173–186. http://pmg.csail.mit.edu/papers/osdi99.pdf.

[10] Cebe, M., Kaplan, B., and Akkaya, K. A network coding based information spreading approach for permissioned blockchain in IoT settings. In *Proceedings of the 15th EAI International Conference on Mobile and Ubiquitous Systems: Computing, Networking and Services* (New York, NY, USA, 2018), MobiQuitous 18, Association for Computing Machinery, p. 470475. https://doi.org/10.1145/3286978.3286984.

[11] Celery Project. Kombu - messaging library for python. https://github.com/celery/kombu.

[12] Correia, M., Neves, N. F., and Verssimo, P. From Consensus to Atomic Broadcast: Time-Free Byzantine-Resistant Protocols without Signatures. *The Computer Journal 49*, 1 (11 2005), 82–96. https://doi.org/10.1093/comjnl/bxh145.

[13] Gilad, Y., Hemo, R., Micali, S., Vlachos, G., and Zeldovich, N. Algorand: Scaling byzantine agreements for cryptocurrencies. In *Proceedings of the 26th Symposium on Operating Systems Principles* (New York, NY, USA, 2017), SOSP 17, Association for Computing Machinery, p. 5168. https://doi.org/10.1145/3132747.3132757.

[14] Gray, J. *A comparison of the Byzantine Agreement problem and the Transaction Commit Problem.* Springer New York, New York, NY, 1990. https://doi.org/10.1007/BFb0042322.

[15] Kimmett, B. Byzantine agreement in expected polynomial time (King-Saia) - implementation and improvement project. https://github.com/bkimmett/polynomial-byzantine.

[16] King, V., and Saia, J. Byzantine agreement in expected polynomial time. *J. ACM 63*, 2 (Mar. 2016). https://doi.org/10.1145/2837019.

[17] King, V., and Saia, J. Correction to byzantine agreement in expected polynomial time, jacm 2016, 2018. https://arxiv.org/abs/1812.10169v2.

[18] King, V., Saia, J., Sanwalani, V., and Vee, E. Towards secure and scalable computation in peer-to-peer networks. In *2006 47th Annual IEEE Symposium on Foundations of Computer Science (FOCS'06)* (Oct 2006), pp. 87–98. https://doi.org/10.1109/FOCS.2006.77.

[19] Kotla, R., Alvisi, L., Dahlin, M., Clement, A., and Wong, E. Zyzzyva: Speculative byzantine fault tolerance. *SIGOPS Oper. Syst. Rev. 41*, 6 (Oct. 2007), 45–58. https://doi.org/10.1145/1323293.1294267.

[20] Kowalski, D. R., and Mostéfaoui, A. Synchronous byzantine agreement with nearly a cubic number of communication bits. In *Proceedings of the 2013 ACM Symposium on Principles of Distributed Computing* (New York, NY, USA, 2013), PODC 13, Association for Computing Machinery, pp. 84–91. https://doi.org/10.1145/2484239.2484271.

[21] Oluwasanmi, O., Saia, J., and King, V. An empirical study of a scalable byzantine agreement algorithm. In *2010 IEEE International Symposium on Parallel Distributed Processing, Workshops and Phd Forum (IPDPSW)* (April 2010), pp. 1–13. https://doi.org/10.1109/IPDPSW.2010.5470874.

[22] Pease, M., Shostak, R., and Lamport, L. Reaching agreement in the presence of faults. *J. ACM 27*, 2 (Apr. 1980), 228–234. https://doi.org/10.1145/322186.322188.

[23] Pivotal Software. RabbitMQ - Open source message broker. https://www.rabbitmq.com/.

[24] Veronese, G. S., Correia, M., Bessani, A. N., and Lung, L. C. EBAWA: efficient byzantine agreement for wide-area networks. In *2010 IEEE 12th International Symposium on High Assurance Systems Engineering* (Nov 2010), pp. 10–19. https://doi.org/10.1109/HASE.2010.19.