Evolving Paxos for the Non-Malicious Arbitrary Model

Enrique S. dos Santos[1] **, Rodrigo R. Barbieri**[1] **, Gustavo M. D. Vieira**[1]

[1]DComp – CCGT – UFSCar
Sorocaba, São Paulo, Brasil

enriquesampaios@gmail.com, rodrigo.barbieri2010@gmail.com

gdvieira@ufscar.br

Abstract. *The non-malicious arbitrary fault model presents a compromise between the weak fault tolerance of the fail-stop fault model and the complexity of the Byzantine fault model. This model describes processes that are capable of using validation codes to detect and drop invalid messages. In this paper we propose a more robust non-malicious version of the distributed consensus algorithm Paxos that can withstand algorithm-level (distributed) faults, while ensuring the application will only see correct states. Our approach to extend the non-malicious model beyond local validations is to define global invariants that must be respected by the algorithm and track them as the algorithm executes. Experimental work shows our solution works for a representative subset of faults and that the performance cost is minimal compared to a local-only non-malicious implementation based on validation codes.*

1. Introduction

Faults are central to distributed systems. Clusters of computers are both a source of a large number of faults due to the many discrete components involved, and a way to achieve fault tolerance due to redundancy in data and computation. It is no surprise that fault-tolerant distributed algorithms are designed according to a specific *fault model*, and that the cost and complexity of the algorithm comes directly from this choice. For instance, the simpler *fail-stop* model leads to considerably less complex algorithms than the more complete *Byzantine* model.

Distributed algorithm complexity derived from the choice of fault model is thus the main driver when one chooses an algorithm for a particular application. The actual faults the algorithm should tolerate are ironically secondary. The reason for this is the gap between the fail-stop model, where processes can only fail by atomically crashing, and the Byzantine model, where processes can fail arbitrarily. There is a large swath of fault types that fall in between these two fault models, many of them interesting for the construction of reliable distributed systems.

One of these models is the *non-malicious arbitrary* fault model [Behrens et al. 2013]. This model describes processes that are *honorable*, meaning that they are capable of using validation codes (CRCs [Correia et al. 2012], arithmetic codes [Behrens et al. 2013]) to detect and drop invalid messages. It is possible to adapt a fail-stop algorithm to the non-malicious model, thus protecting the system from transient hardware faults such as bit-flips and stuck bits [Behrens et al. 2013]. The hardening isn't perfect, as it can only provably protect the *data* using appropriate codes. However, transient hardware faults can

also trigger software errors, where the behavior of the program is changed. Moreover, one can propose that other non-intentional errors such as programming or configuration errors also can be considered non-malicious arbitrary behavior, even if they go beyond the basic definition of honor.

Research on the non-malicious fault models have focused mainly on the locally detectable hardware induced errors [Behrens et al. 2013, Barbieri and Vieira 2015], with some provisions made for code redundancy [Correia et al. 2012, Behrens et al. 2015] and semantic validation of operations [Bhatotia et al. 2010]. In previous work, our group have experimented with an opportunistic approach to extent these protections to the distributed algorithm itself [Barbieiri et al. 2019]. That is, we tried to detect software errors that induce deviations from the distributed algorithm specification, while creating only valid local messages. The results were mixed, the proposal had low overhead, but because of its opportunistic nature it allowed invalid state to corrupt the application before it was detected.

In this paper we propose a more robust non-malicious version of the distributed consensus algorithm Paxos that can withstand algorithm-level (distributed) faults, while ensuring the application will only see correct states. Consensus is a central problem in distributed systems and many problems can be solved using consensus, thus giving generality to our contribution. Our approach to extend the non-malicious model beyond local validations is to define *global* invariants that must be respected by the algorithm and track them as the algorithm executes. Moreover, we restrict the evolution of the internal states of the algorithm and its interaction with other components of the system only when these invariants hold. Experimental work shows our solution works for a representative subset of faults and that the performance cost is minimal compared to a local-only non-malicious implementation based on validation codes.

The main contributions of this paper are: we describe a non-malicious fault model suitable for validation of distributed predicates (Section 2.1), we propose a variant of the Paxos algorithm for this model (Section 3.3), we describe an actual implementation of this algorithm (Section 4) and show experimental validation of our approach (Section 5).

2. Preliminaries

2.1. Non-malicious Arbitrary Fault Model

Fault models abstract the properties a system must satisfy and how a distributed algorithm should tolerate faults. Two very common fault models in which distributed algorithms are designed are the *fail-stop* and *fail-recovery* models. In both models processes only fail by completely crashing. We can call these models *benign* because it is assumed that the tolerated faults will respect a (probably unknown) probability distribution. A stronger fault model that assumes more types of faults is the *Byzantine* or *arbitrary* model in which processes can deviate in any way from the algorithm specification.

In the arbitrary fault model it is impossible for processes to decide whether another process is behaving arbitrarily intentionally or not. This assumption covers virtually any type of fault a system might encounter. We refer to *malicious* faults when a process is behaving arbitrarily intentionally, through manipulation from a malicious agent. These type of faults do not follow a predefined probability distribution and can occur in response to measures taken to tolerate them.

Fault models range from weaker (more strict) to stronger (more general). The stronger the model, the more complex and difficult it is to implement an algorithm. When building a practical distributed system, it is desirable to adopt a fault model that better fits the system and satisfies its requirements for performance and types of faults it must tolerate. However, this is not always the case, since any distributed system that relies on actual computers is prone to arbitrary faults.

Distributed system designers desire to tolerate arbitrary faults, but would prefer a less resource intensive algorithm than a byzantine one [Bhatotia et al. 2010, Correia et al. 2012, Behrens et al. 2013]. While malicious faults are being tolerated using different techniques [Bhatotia et al. 2010, Correia et al. 2012], and based on the premise that any fault model can be hardened to tolerate some arbitrary faults, it is possible to harden the fail-recovery benign model to tolerate non-malicious arbitrary faults, thus achieving a fault model similar to the arbitrary fault model, but where malicious faults are not necessarily tolerated by the algorithm.

An algorithm for the non-malicious arbitrary fault model can be considered to be less complex than an classical arbitrary one because it does not tolerate malicious faults in its implementation. However, the implementation required to tolerate all non-malicious arbitrary faults adds its own complexity to the algorithm. This fault model is more precisely described in terms of the two following properties [Behrens et al. 2013]:

No impersonation: the environment never creates valid messages, except for duplicates. This property assumes that only processes themselves are able to create valid messages, so malicious agents cannot interact with existing processes in a system.

Honor: a process that is considered faulty, by either itself or by another process, cannot ever create a valid message. This property implies that data corruption can be detected using validation codes, such as CRCs or arithmetic codes.

An algorithm for this fault model is expected to tolerate faults caused by data corruption, such as from persistent memory, main memory or network, in addition to benign faults [Bhatotia et al. 2010, Correia et al. 2012, Behrens et al. 2013].

For this paper, we extend this model by replacing the Honor property with a weaker version:

Weak honor: a process that is *knowingly* faulty, by either itself or by another process, cannot ever create a valid message.

This weaker model is required by the fact that individual processes in distributed algorithms do not possess knowledge of the full state of the distributed system. Thus, a process can have corrupted global state that is locally valid. The weaker property allows these corrupted processes to exchange messages, and effectively be able to detect their corruption.

2.2. Consensus and Paxos

Paxos is a distributed algorithm that solves the problem of consensus in a distributed system. Consensus is simple to state, and hard to solve under the presence of faults. Consider a set of N processes that want to decide a single value. Each one of these processes *proposes* a value, and the correct processes in this set must *decide* on a single of these

proposed values. More formally, the decision must respect these properties [Cachin et al. 2011]:

Termination: Every correct process eventually decides a value.
Validity: If a process decides a value, this value was previously proposed by some process.
Integrity: No process decides twice.
Agreement: No two correct processes decide differently.

Paxos implements consensus and is specified in terms of roles and agents; an agent performs a role. Different implementations of Paxos may choose different mappings between agents and the actual processes that execute them. Agents communicate exclusively via message exchanges. The roles agents can play are: a *proposer* that can propose values, an *acceptor* that chooses a single value, or a *learner* that learns what value has been chosen. To solve consensus, Paxos agents execute multiple *rounds*, each round has a *coordinator* and is uniquely identified by a positive integer. Proposers send their proposed value to the coordinator that tries to reach consensus on it in a round. The coordinator is responsible for that round, and is able to decide, by applying a local rule, if any other rounds were successful or not. The local rule of the coordinator is based on quorums of acceptors and requires that at least $\lfloor N/2 \rfloor + 1$ acceptors take part in a round, where N is the total number of acceptors in the system [Lamport 2006]. This way, any two quorums have at least one process in common. Each round progresses through two phases with two steps each:

- In Phase 1a the coordinator sends a message requesting every acceptor to participate in round r. An acceptor accepts the invitation if it has not already accepted to participate in round $s \geq r$, otherwise it declines the invitation by simply ignoring it.
- In Phase 1b, every acceptor that has accepted the invitation answers to the coordinator with a reply that contains the round number and the value of the last vote it has cast for a proposed value, or *null* if it has never voted.
- In Phase 2a, if the coordinator of round r has received answers from a quorum of acceptors, it analyzes the set of values received and picks the single value v with the highest round number. It then asks the acceptors to cast a vote for v in round r, if v is not *null*, otherwise the coordinator is free to pick any value and picks the value proposed by the proposer.
- In Phase 2b, after receiving a request from the coordinator to cast a vote, acceptors can either cast a vote for v in round r, if they have not voted in any round $s \geq r$, otherwise, they ignore the vote request. Votes are cast by sending them and their respective round identifiers to the learners.
- Finally, a learner learns that a value v has been chosen if, for some round r, it receives Phase 2b messages from a quorum of acceptors announcing that they have all voted for v in round r.

This description of the algorithm considers only a single instance of consensus. However, Paxos also defines a way to deliver a set of totally ordered messages. The order of delivery of these messages is determined by a sequence of positive integers, such as each integer maps to a consensus *instance*. Each instance i eventually decides a proposed value, which is the message (or ordered set of messages) to be delivered as the

ith message of the sequence. Each consensus instance is independent from the others and many instances can be in progress at the same time.

3. Non-malicious Arbitrary Paxos

To adapt Paxos for the non-malicious model we create an hierarchy of trustworthiness. At the lower level data integrity checks ensure data has not been corrupted and only correct data is propagated between processes, relying on the honor property of the fault model. These *local* checks have been extensively studied [Correia et al. 2012, Behrens et al. 2013, 2015], including by our group [Barbieri and Vieira 2015]. At the algorithm level, *global* invariants ensure that incorrect processing of the correct data produced by the lower level cannot occur. Crucially, at the top, the client application only sees correct behavior, this time relying on the weak honor property. In this section we describe the components of this hierarchy.

3.1. Data Integrity Checks

Integrity checks verify data by saving at least one redundant copy that can be used to validate against the original data, such as CRCs, duplicate states, timestamps or data size values. This approach is commonly used when reading and writing data from main memory, storage and peer-to-peer network message exchanges. We now describe in more detail these techniques:

Data and state redundancy: each process variable or stored data has a duplicate which can be validated against and used for backup. The duplicates must always be kept in sync and checked for consistency on each read and write operation. This approach clearly uses a significant amount of additional memory and has an increased overhead for keeping both states in sync [Bhatotia et al. 2010, Correia et al. 2012, Chandra et al. 2007];

Validation codes redundancy: the usage of encoded redundancy allows for future detection of undesired corruption. The most common type of redundancy is generating a CRC of data and attaching it to the protocol messages prior to transmitting across the network or saving them in storage. Any read or write operation on data must recalculate the CRC and verify against the one attached to the message, adding a significant performance cost related to the coding algorithm used [Bhatotia et al. 2010, Correia et al. 2012, Chandra et al. 2007];

Encoding and arithmetic codes: the usage of in-place encoding and decoding, like numerical properties of data, can be used to detect undesired corruption in each read and write operation. For instance, if numerical variables are multiplied by a prime number upon writing and divided by the same number when they are read back, the remainder should always be zero. This approach is considered to be very efficient performance-wise, but lacks coverage [Behrens et al. 2013].

3.2. Semantic Checks

Semantic checks validate that after an operation has been applied on data, the newly obtained state is semantically correct according to the applied operation [Bhatotia et al. 2010]. For instance, after adding an element to a list, check if the element is in the list. This approach has the added benefit of testing the system against possible bugs, which was one scenario in the experiment found in [Chandra et al. 2007].

In a sense, semantic checks define invariants of the application that should not be violated. For correct program operating in a classic benign fault model a semantic check should be completely unnecessary. Thus, in the non-malicious model these checks flag deviation of the specification caused by corruption of code, instead of data.

3.3. Distributed Algorithm Checks

Data integrity checks allow a distributed algorithm to check that its data has not been corrupted after it was created. However, corruption in the code of the algorithm can lead to execution of unreachable and invalid states in the algorithm state machine. These occurrences are very rare [Behrens et al. 2015], but are very hard to tolerate without using Byzantine algorithms. However, due to their rarity it suffices to detect them. Sometimes we can even tolerate some of these faults by crashing the affected component.

To detect deviation of the code, we can check if the algorithm works as intended. Please note that checking if a distributed property holds is often much easier than implementing the property. For instance, the consensus problem requires as its *Agreement* property (Section 2.2) that all correct processes that decide a value, decide the same value. The validation of this property is exactly what our distributed validation aims for and the procedure can be summarized as: a Paxos implementation only outputs a decision value v after it has verified that v is the only valid value decided by the other processes.

More formally, we replace the last step of the Paxos algorithm with a new phase, keeping Phases 1 and 2 unchanged:

- In Phase 3a, if a learner receives Phase 2b messages from a quorum of acceptors announcing that they have all voted for v in round r, it generates a validation code and records v as a tentative decision v_t. It then broadcasts v_t and its code to all learners.
- In Phase 3b, if a learner has received Phase 3a messages from a quorum of learners, it analyses the set of values and codes received. If there is only a single value v, this value is the same as v_t and its code is also the same, then v is chosen as the decision value. If not, a corruption was detected and all local agents are crashed.

This additional phase checks if the previous phases worked as intended, by validating the agreement property of consensus. This way, the extended algorithm implements consensus and all its properties, while also respecting the weak honor property, as we now show in this proof sketch.

Theorem. *The Paxos algorithm augmented with the steps of Phase 3 respects the weak honor property.*

Proof (sketch). Consider two different processes p and q that violate the weak honor property and output two different values as the consensus decision. This means that both p and q have received confirmation of their decision from a quorum of processes in Phase 3b, each quorum indicating they decided v or w, respectively. Both these quorums must have an intersection and the processes in the intersection must have responded differently to p and q. However, this is impossible, since the processes are non-malicious and will follow the algorithm and return the single values decided in Phase2b. Thus, its impossible for p and q to exist. □

It is important to note, and the proof sketch makes clear, that Phase 3 of the algorithm is only crash resistant. A deviation of the specification of Phase 3, even if non-malicious in origin, can hinder the simple procedure of validating the consensus decision. However, this phase can have its implementation hardened using the redundancy techniques presented in [Correia et al. 2012] or [Behrens et al. 2015]. Nonetheless, the amount of code in need of duplication is significantly smaller in our proposal.

4. Hardened Treplica

We have implemented the techniques described in previous sections as a hardened variant of the Treplica framework for distributed applications [Vieira and Buzato 2008]. In this section we describe the way we used these checks, and how we combined some of them to increase the protection coverage they provided.

4.1. Treplica

Treplica [Vieira and Buzato 2008, 2010] is a Java framework that allows distributed applications to use Paxos as middleware to manage state replication through its state machine. Its implementation is close to multi-instance Paxos [Lamport 1998] with a few additional optimizations, like Fast Paxos support [Lamport 2006] and broadcast votes, where each learner agent receiving a majority of votes can commit the change immediately.

Applications designed according to the Model-View-Controller standard can easily be modeled to use Treplica. We chose Treplica because its modular architecture allows for improvements to be easily coded and tested. Since it is designed to tolerate benign faults, upon analysis we validated that it is prone to non-malicious arbitrary faults we are interested in. Additionally, Treplica is object-oriented and makes use of immutable objects design, where an object is never changed after being instantiated. This allows for more efficient use of validation codes. State transition semantic checks can also be easily coded by the application due to its integration with the state machine modeling.

4.2. Data Integrity

The approach used for checking data integrity is the use of validation codes. In Hardened Treplica the main unit of data transfer is a *message*, used both to be sent over a network but also to be stored in stable memory. Messages in Hardened Treplica are immutable objects that carry validation codes computed during their first instantiation. Every time a message is processed, the codes are recalculated and the contents of the message are validated. These validations happen mostly when a message is received or when it is retrieved from stable storage, because messages are not stored internally in Treplica after being processed. Hardened Treplica supports many types of validation codes, but the default is a simple and fast CRC32.

Besides messages, the internal state of the algorithm is also validated with codes. Hardened Treplica is a framework that relieves the programmer from caring about state replication and persistence, so the client application's state is also part of the internal state of the algorithm and is under the same protections. This creates a potential performance problem, because the sum of the Paxos implementation state and of the client application's state can be large and take a very long time to compute a validation code. There are many ways to dealing with this problem, but as the client application's state from Treplica's

point of view is opaque we decided to let the application programmer decide between two approaches:

1. Hardened Treplica computes the validation code of the whole internal state. This takes time, but is reliable and can be used for applications with small state.
2. Hardened Treplica asks the application for a *application defined validation code* that can be an abridged validation code, easier to compute, or any other technique appropriate for the application. The degree of protection is up to the application programmer, considering the size of the application and the cost of generating a validation code.

Additionally, to increase the coverage of a simpler application defined validation code, we create a code derived by the combination of the (potentially application defined) state code with the code of the *message* that created the state. This even allows the application defined validation code to be empty, because the message-derived codes allows for a state identifying code useful for global validations as the one described in 3.3, but at the cost of not checking for data corruption in the internal state.

4.3. Data Redundancy

In Hardened Treplica we keep two copies of the internal state, to increase the usefulness of internal state validation codes by creating the possibility of *error recovery*. Each message that changes the internal state is applied to both states, and their codes are calculated as described in the previous section. Both codes must be equal, if the new state was correctly created by the message. Also, at this moment, semantic checks defined by the programmer are implemented. After comparing validation codes, the final state of the both copies run the application specific semantic check to ensure the end result is both unique and correct. In case of divergence of either the validation code or the semantic check, the operation can be rolled back to the previous state.

4.4. Distributed Algorithm Integrity

We have implemented in Hardened Treplica the global validation described in Section 3.3, and integrated it with the state validation code mechanism for greater fault coverage. This integration replaces in Phase 3a the tentative decision value v_t with the combined application and decision (message) *validation code* c_t. In Phase 3b, the validation of the consensus decision is made exclusively based on this code c_t and the codes received by a quorum of processes. This way we can validate that both the decision value is the same for all replicas but also that the state *derived* from this decision is the same and that all replicas advanced consistently together.

Implementing this integration requires that a tentative internal algorithm state be created based on a tentative decision v_t to compute the tentative code c_t. However, for the algorithm to respect the weak honor property, this tentative internal algorithm state can not be "leaked" to the application. To solve this problem we leverage the state redundancy described in the previous section. Consider that we have two copies of the internal state A and B. Without loss of generality, c_t is calculated by applying v_t only on state A, while state B is kept constant. While the Phase 3 of the algorithm runs, all client requests for data are served by state B. Only after state A has been validated by the complete Phase 3 of the algorithm, state B will be updated with the now definitive decision v (and be subject to the redundancy validations described in the previous section).

Another complication is that in Hardened Treplica a single decision value is effectively composed of many messages to increase the throughput of the system, considering the cost of the Paxos algorithm. This is solved by simply considering the *last* message of the bundle as responsible for creating c_t, after applying all preceding messages to state A. Also, to increase the throughput, Phase 3 of one instance of consensus is executed concurrently with Phases 1 and 2 of the next instances. This parallelism is easily achieved because the creation of c_t already happens in parallel with the execution of the algorithm due to the fact that an application provided validation code is required.

5. Experimental Validation

The state integrity techniques implemented in Hardened Treplica have been validated in previous work [Barbieri and Vieira 2015, Barbieiri et al. 2019]. Here we want to assess the improved distributed algorithm validation as a way to protect the consensus algorithm against deviations from its specification. We reach this objective by injecting faults in select operations of the algorithm implementation and observe the occurrence of errors, if any, caused by these injected faults. We also ran a performance test, without the injection of faults, and compared that results with other published works.

It is important to remember the expected source of the faults we are simulating: memory corruption. The occurrence of memory corruption in data can be detected by CRCs and other validation codes, however memory corruption in code can lead to deviation from the implemented algorithm [Correia et al. 2012]. These faults are rare because often the fault naturally does not induce an error [Behrens et al. 2015], however when an error occurs its effect can be catastrophic [Correia et al. 2012].

To ensure our proposed distributed algorithm validation was robust in avoiding such errors, we hand picked and injected faults in key points of the consensus algorithm in a way to increase the potential occurrence of catastrophic errors. In this section we describe our experimental setup, with emphasis in the types of faults injected, the experimental parameters and test load used, and the results obtained.

5.1. Injected Faults

We selected four faults affecting the three central Paxos agents (acceptor, learner and coordinator) and the message exchange infrastructure itself. Each of the agents were selected because of its role in the phases of the Paxos algorithm. The acceptor is essential to Phases 1b and 2b, being responsible for recording the (partial) consensus results in stable storage. The leaner acts as the listener at the end of Phase 2b, being responsible for passing the decision to the client application. The coordinator is crucial for the correct operation of Phases 1a and 2a, being responsible for recovering a decision after an interruption. Additionally, we injected faults in the communication subsystem as a catalyst to the other fault types.

We now describe in more detail each type of fault:

Acceptor forgets last vote (Phase 1b): After an acceptor casts a vote in Phase 2b, it records the voted value in stable storage to support the occasional recovery initiated by a future coordinator. This fault makes an acceptor forget the vote it has previously cast and

instead reply the coordinator request in Phase 1b with the null vote. Once this fault hits, a coordinator can be misled to decide a value different from a previous successful decision.

Learner decides without a quorum (Phase 2b): After Phase 2b the learners can decide as they receive votes from a quorum of acceptors. The requirement of a quorum is very important as it ensures that a future coordinator will always reach the same decision of this learner. This fault makes a learned decide even when it has not yet received enough votes, thus committing to a decision that can be reversed by a future coordinator.

Coordinator ignores acceptors' answers (Phase 2a): Whenever a coordinator needs to start a new consensus round if probes a quorum of acceptors to discover if any value was or could have been decided. This is arguably one of the crucial steps of the algorithm, as it ensures that a decision once taken is always respected. This fault causes the coordinator to ignore the responses received from the acceptors, and to propose the null vote instead. This deviation can make the coordinator to reverse a previously made decision.

Message payload corruption The algorithm fault types just described are not limited to the local insertion of erroneous values, but are instead related to the interaction of the Paxos agents. We can notice that by observing that these faults trigger an error when a coordinator tries to resume an interrupted consensus round. As such, these faults could for most part stay latent if we consider the occurrence of round interruptions to be rare. Paxos is indeed a very resilient algorithm, even when wrong. To consistently trigger errors from the injected faults we also injected simple message payload errors. The fault is simple corruption of bits in the message payload, easily detectable by the message integrity mechanisms [Barbieiri et al. 2019]. The end result is a message loss trigger, also controlled by a fixed probability. Using this type of injected fault we can control the amount of the interrupted consensus rounds and fine tune the triggering of errors as we inject the other type of faults. In the next section we describe the probabilities used.

5.2. Experimental Design and Parameters

5.2.1. Fault Injection Parameters

The fault injecting framework used in an improved version of the framework used in previous work [Barbieiri et al. 2019], and uses AspectJ to insert *pointcuts* in the code to replace correct code with faults. In brief, based on a predefined probability, a correct implementation of a complete method is replaced by a faulty implementation that deviates from the algorithm specification. For example, if a pointcut is associated with a 20% probability, then 20% of the time a call to the annotated method will be replaced with a call to faulty version of the same method.

Each of the four faults described in the previous section is represented by one pointcut, and we composed these pointcuts to define 5 fault scenarios, as shown in Table 1. In Scenarios 1 and 3 and Scenarios 2 and 4 the same fault is injected, respectively. In Scenarios 1 and 2 the fault is injected in only one of the replicas, while in Scenarios 3 and 4 the fault is injected in all replicas. The fault injected in Scenario 5 is restricted

to the coordinator agent of Paxos and thus was injected in *all* replicas to ensure that it was injected in the replica currently hosting the coordinator agent. However, in effect, the configuration setup in Scenario 5 effectively only injects faults in a single replica, the coordinator.

Table 1. Fault Scenarios

Scenario	Injected fault	Failed replicas
1	Acceptor forgets last vote	1 (20%)
2	Learner decides without a quorum	1 (20%)
3	Acceptor forgets last vote	5 (100%)
4	Learner decides without a quorum	5 (100%)
5	Coordinator ignores acceptors' answers	5 (100%)

Besides the number of replicas, we must define the probability of injection of the selected fault for each scenario. If the faults were injected with 100% probability, the algorithm would not work at all and any detection made meaningless. Thus, we have to select a non zero probability that ensures a steady stream of faults. Moreover, as described in the previous section, it is required some level of message loss for these faults to trigger an error. To provide this, we combine the injection of the faults in Table 1 with the injection of the message payload corruption fault.

Due to the nature of each injected fault, the rate of errors triggered are not the same for a given probability of fault injection. We calibrated the probabilities of fault generation to create not only a steady stream of faults, but also a steady stream of errors with the same frequency for *all test scenarios*. We selected an *error* probability of 80% for each execution, regardless of the type of fault injected. The exact fault mix and injection probabilities used for each scenario are listed in Table 2.

Table 2. Fault Probabilities

Scenario	Injected fault	Probability
1	Acceptor forgets last vote – one replica	99%
	Message payload corruption – all replicas	20%
2	Learner decides without a quorum – one replica	80%
	Message payload corruption – all replicas	20%
3	Acceptor forgets last vote – all replicas	90%
	Message payload corruption – all replicas	10%
4	Learner decides without a quorum – all replicas	80%
	Message payload corruption – all replicas	10%
5	Coordinator ignores acceptors' answers – all replicas	30%
	Message payload corruption – all replicas	1%

5.2.2. Experimental Setup

Experimental executions use 5 replicas each running a copy of a simple service, a basic key value database. Each of these replicas runs in a virtual machine with 4 GB RAM, 2 CPUs, 12 GB HD, running CentOS 7 and Java 1.8.0, hosted in a dedicated computer cluster. The execution comprises a 300 s run of about 1000 op/s in each replica, adding

to a total of 5000 op/s load. Each operation generated adds a key to the database, and requires a consensus round to commit.

For each type of injected fault we run 50 executions of the experiment, inspect the errors generated in these runs and derive reliability metrics based on these errors. Considering the desired 80% probability of errors being triggered by the injected faults, we obtain for each experimental run, on average, about 40 executions with errors and 10 error-free executions. For the performance test we use the exact same setup, but change all injection probabilities to 0%.

The data generated was processed by an automated script that provided a breakdown of how many errors were detected. The counting of detected errors is simple enough, being done by simple inspection of the execution log in search of a tell tale sign of a consistency violation. However, we wanted to ensure that all errors, detected or *undetected* by the validation mechanism, were properly recorded by the processing script. This is achieved by dumping, after completing the test run, the complete internal state of the application. This state was then compared between all surviving replicas at the end of the test run. Thus, we can be confident that if any fault would trigger an error, we would know it by the end of the experiment regardless of the distributed algorithm validation ability to detect it.

5.3. Data and Analysis

5.4. Fault Injection Tests

The data for 50 runs of the fault injection tests are in Table 3. We tabulate how many processes were interrupted because the validation mechanism detected a fault. We also recorded, as described in the previous section, if the surviving replicas were consistent at the end of the test run.

Table 3. Fault Injection Data

Scenario	Injected fault	Stopped processes			Errors
		0 proc.	1 proc.	2 proc.	
1	Acceptor forgets last vote – one replica	19	16	15	0
2	Learner decides without a quorum – one replica	8	21	21	0
3	Acceptor forgets last vote – all replicas	12	7	31	0
4	Learner decides without a quorum – all replicas	9	7	34	0
5	Coordinator ignores acceptors' answers – all replicas	13	5	32	0

In our tests the validation mechanism proposed was able to detect and stop all defective processes before their fault could trigger an error in the replication algorithm. Moreover, this fault detection ratio was achieved in an environment where a massive number of benign faults were generated and about 80% of the runs had some type of algorithm-level fault triggered by these "low level" faults.

As described in Section 5.1 the algorithm-level faults require a consensus round to be restarted in a way to create two distinct replica quorums with an inconsistent intersection. Due to the nature of these faults, once the system reaches 2 failed processes and there is only one quorum of 3 processes the faults stop. That's the reason we only count up to 2 faults; it is impossible in our setup for more to occur. This was a happy accident of

our experimental design, allowing even a computation with a high rate of faults to execute to completion and to have its final state checked for consistency.

It is important to note that the behavior observed in the experiments does not rule out the possibility of more faults to happen and *be detected* by the proposed validation in other configurations. In this case, more than a quorum of processes would be stopped and the system would violate the supposition that $N/2 + 1$ processes are correct. As a consequence, the computation would stop as soon as validation quorum could not be found. That is, the algorithm would stop because of the fault, but an erroneous answer would not be generated.

5.5. Performance Test

The data from the 50 runs of the performance tests is in Table 4. The table compares three related implementations of the Paxos algorithm. The classic Paxos implementation is the one found in unmodified Treplica [Vieira and Buzato 2008], the data integrity only implementation is Treplica with the data integrity validations as described in [Barbieiri et al. 2019] and the full distributed validation is the work presented in this paper. We measured average throughput (in operations per second) and latency (in milliseconds) over 50 fault free executions.

Table 4. Performance Data

Version	Throughput		Latency	
	Op/s	**Std Dev**	**ms**	**Std Dev**
Classic Paxos [Vieira and Buzato 2008]	1299	191	615	495
Data integrity only [Barbieiri et al. 2019]	687	90	1245	910
Distributed validation	657	107	1546	1713

Comparing the performance of the three algorithms we can observe that implementing the non-malicious validations has a noticeable cost compared to classic Paxos. The throughput we observed with the hardened implementation was 50% smaller than the one of the classic implementation, and this difference is statistically significant using a independent t-test considering $p < 0.001$ ($t_{(50)} = 20.736$). However, most of this cost is due to the computation of the codes required for data validation, as the comparison with the data integrity only implementation shows. Compared to this implementation the proposed validation has a performance hit of about 4%. This small difference isn't statistically significant using a independent t-test with $p = 0.1332$ ($t_{(50)} = 1.517$).

6. Conclusion

In this paper we propose a more robust non-malicious version of the distributed consensus algorithm Paxos that can withstand algorithm-level (distributed) faults, while ensuring the application will only see correct states. To support this version of the algorithm we have extended the basic non-malicious fault model with a weaker version of the Honor property, suitable for the validation of global predicates. We then proposed a variant of the Paxos algorithm that only makes a final consensus decision after it was able to validate the Agreement property of consensus. We also present Hardened Treplica, an actual implementation of this algorithm and other hardening techniques respecting the non-malicious fault model. The fault detection coverage and performance of this implementation was experimentally shown to be very good. The distributed algorithm validation mechanism

proposed was able to detect and stop all failed processes before their fault could trigger an error in the replication algorithm. While the performance cost of calculating validation codes is substantial, the additional cost of the distributed algorithm validation in isolation was minimal.

References

Barbieiri, R. R., dos Santos, E. S., and Vieira, G. M. D. (2019). Decentralized validation for non-malicious arbitrary fault tolerance in Paxos. In *WTF SBRC '19: Proc. of the 20th Workshop on Tests and Fault Tolerance*, WTF SBRC '19, Gramado, RS. SBC.

Barbieri, R. R. and Vieira, G. M. D. (2015). Hardened Paxos through consistency validation. In *SBESC '15: Proceedings of the V Brazilian Symposium on Computing Systems Engineering*, SBESC '15, pages 13–18, Foz do Iguaçu, Brazil. IEEE Computer Society.

Behrens, D., Serafini, M., Arnautov, S., Junqueira, F. P., and Fetzer, C. (2015). Scalable error isolation for distributed systems. In *Proceedings of the 12th USENIX Conference on Networked Systems Design and Implementation*, NSDI'15, page 605–620, USA. USENIX Association.

Behrens, D., Weigert, S., and Fetzer, C. (2013). Automatically tolerating arbitrary faults in non-malicious settings. In *Dependable Computing (LADC), 2013 Sixth Latin-American Symposium on*, pages 114–123.

Bhatotia, P., Wieder, A., Rodrigues, R., Junqueira, F., and Reed, B. (2010). Reliable data-center scale computations. In *Proceedings of the 4th International Workshop on Large Scale Distributed Systems and Middleware*, LADIS '10, pages 1–6, New York, NY, USA. ACM.

Cachin, C., Guerraoui, R., and Rodrigues, L. (2011). *Introduction to reliable and secure distributed programming*. Springer.

Chandra, T. D., Griesemer, R., and Redstone, J. (2007). Paxos made live: an engineering perspective. In *PODC '07: Proceedings of the twenty-sixth annual ACM symposium on Principles of distributed computing*, pages 398–407, New York, NY, USA. ACM Press.

Correia, M., Ferro, D. G., Junqueira, F. P., and Serafini, M. (2012). Practical hardening of crash-tolerant systems. In *USENIX Annual Technical Conference*, pages 453–466.

Lamport, L. (1998). The part-time parliament. *ACM Trans. Comput. Syst.*, 16(2):133–169.

Lamport, L. (2006). Fast Paxos. *Distrib. Comput.*, 19(2):79–103.

Vieira, G. M. D. and Buzato, L. E. (2008). Treplica: Ubiquitous replication. In *SBRC '08: Proc. of the 26th Brazilian Symposium on Computer Networks and Distributed Systems*, Rio de Janeiro, Brazil.

Vieira, G. M. D. and Buzato, L. E. (2010). Implementation of an object-oriented specification for active replication using consensus. Technical Report IC-10-26, Institute of Computing, University of Campinas.