Automatically Tolerating Arbitrary Faults in Non-malicious Settings

Diogo Behrens, Stefan Weigert, Christof Fetzer
Systems Engineering Group
Technische Universität Dresden
Dresden, Germany
Email: {firstname.lastname}@tu-dresden.de

Abstract—Arbitrary faults such as bit flips have been often observed in commodity-hardware data centers and have disrupted large services. Benign faults, such as crashes and message omissions, are nevertheless the standard assumption in practical fault-tolerant distributed systems. Algorithms tolerant to arbitrary faults are harder to understand and more expensive to deploy (requiring more machines).

In this work, we introduce a non-malicious arbitrary fault model including transient and permanent arbitrary faults, such as bit flips and hardware-design errors, but no malicious faults, typically caused by security breaches. We then present a compiler-based framework that allows benign fault-tolerant algorithms to automatically tolerate arbitrary faults in non-malicious settings. Finally, we experimentally evaluate two fundamental algorithms: Paxos and leader election. At expense of CPU cycles, transformed algorithms use the same number of processes as their benign fault-tolerant counterparts, and have virtually no network overhead, while reducing the probability of failing arbitrarily by two orders of magnitude.

Keywords—arbitrary faults; Byzantine faults; hardware errors; fault tolerance; algorithm transformation

I. INTRODUCTION

Recent studies have shown that transient and permanent hardware errors such as bit flips and stuck-at bits occur surprisingly often in server [1] and customer machines [2]. Large services have been disrupted by arbitrary faults of this nature, resulting in long periods of unavailability, e.g., Amazon S3 [3]. Moreover, the hardware error rate is expected to increase with the upcoming hardware generations [4]. Nevertheless, practical fault-tolerant distributed systems are not built to tolerate arbitrary faults. Examples of such systems are ZooKeeper [5] and Chubby [6], which are used to coordinate critical tasks of Web-scale applications. On the one hand, these systems are deployed on commodity hardware [7] because hardware fault-tolerant computers are expensive and usually an order of magnitude slower than commodity hardware [8]. On the other hand, the distributed algorithms implemented in these systems are designed to tolerate benign faults only, e.g., process crashes, message omissions, etc.

Distributed algorithms designed to tolerate arbitrary faults (Byzantine faults) do exist, for example, PBFT [9], but several factors complicate their adoption in practical systems. First, arbitrary fault-tolerant algorithms are harder to understand and to prove correct than their benign fault-tolerant counterparts [10], [11]. Second, they often require more hardware resources, incurring higher ownership and maintenance costs. For instance, a replicated service requires at least $2f + 1$ replicas to tolerate $f$ crash failures, but at least $3f + 1$ replicas to tolerate $f$ arbitrary process failures [9]. Moreover, the higher number of replicas leads to higher failure frequencies, which can in turn lead to lower availability than in benign fault-tolerant systems [12]. Finally, the arbitrary fault-tolerant systems also tolerate malicious attacks as long as replicas are sufficiently diverse (i.e., a successful attack on one replica does not imply successful attacks on more than $f$ replicas). In many scenarios, however, such tolerance is either redundant or an overkill [13], for instance, in data centers protected by firewalls.

In this paper, we define the non-malicious arbitrary fault model, which covers transient (e.g., bit flips) and permanent hardware faults (e.g., stuck-at bits), but no malicious adversaries (e.g., hackers) as in the classical Byzantine fault model. Our fault model is based on three observations. First, machines are often underutilized and have spare CPU cycles [14]. Second, one can extend the fault coverage of programs by using the spare cycles with redundancy and fault detection mechanisms, for example, arithmetic codes [15]. Third, malicious attacks are often handled by orthogonal mechanisms [13].

In Sect. II, processes transformed with such a fault detection mechanism are called honorable processes. In contrast to native processes, honorable processes add to each sent message an evidence of good behavior, called message validity. Ideally, any faulty transition a process has taken, for example, an error in the arithmetic logic unit, should be reflected on the validity of the messages. We then restrict the system failure semantics with two non-maliciousness assumptions, which have a high coverage1 in settings with no malicious adversaries. Finally, we show that, if correct processes drop invalid messages, executions with arbitrary faults can be mapped into executions with benign faults in non-malicious settings (see Fig. 1).

In Sect. III, we present a proof-of-concept framework for

1The coverage of an assumption is the probability of the assumption holding given that a failure occurred. [12]
building distributed algorithms that automatically transforms benign fault-tolerant algorithms into algorithms that tolerate arbitrary faults in non-malicious settings. Our framework enforces processes to be honorable with an adapted version of the encoding compiler by Schiffel et al. [16], which implements arithmetic codes, a well-known hardware error detection technique [15], [17] to protect memory and computation. In short, the representation of each state variable of a process is extended from $n$ to $n + k$ bits such that only $2^{n}$ of the $2^{n+k}$ code words are considered valid code words. By randomly choosing the valid code words, the likelihood of a faulty transition results in a valid code is $2^{-k}$. One can select $k$ such that the probability of a faulty transition that results in a valid code becomes negligible [18]. Since messages are stored in state variables, they become invalid with a high probability if a fault happens, and can be filtered by correct processes.

Our approach tackles the problems mentioned above by allowing the same well-tested and familiar benign fault-tolerant algorithms to be used to tolerate arbitrary faults with no extra hardware (neither machines nor hardware fault detection components). Hence, replicated services can tolerate $f$ arbitrary process faults with $2f + 1$ replicas in non-malicious settings. In contrast to other transformation-based approaches [10], [11], [19], the message complexity of our transformed algorithms is the same as of the native algorithms. A honorable process sends not only the same number of messages, but also exactly the same messages with exception of the additional bits representing the message validity. (See Sect. V for further related work.)

In Sect.IV, we evaluate our framework with algorithms for two important problems in distributed computing: strong leader election and consensus. Specifically, we implement the Highly-available Leader Election Service by Fetzer and Cristian [20]; and the multi-instance Paxos by Lamport [21], which is widely used in replicated systems [6], [22]. Our results illustrate that one can tolerate a wide variety of arbitrary faults such as bit flips, stuck-at bits, and hardware design failures, at the cost of extra CPU cycles. In particular, a highly fault-tolerant variant of the leader election algorithm can achieve the same election times as with a benign fault-tolerant variant at the expense of at most 11% more CPU utilization. Paxos variants transformed with encoding can reach the same request throughput as the native variant with 5 acceptors. Finally, our fault injection experiments show that our approach reduces the probability of Paxos violating honor by two orders of magnitude.

In summary, the contributions of this paper are:
1) A non-malicious arbitrary fault model suitable for data centers protected by firewalls (Sect. II);
2) A compiler-based framework (Sect. III) that automatically transforms benign fault-tolerant systems into systems that tolerate arbitrary faults;
3) An experimental evaluation (Sect. IV) with two important distributed algorithms showing that arithmetic codes can enforce our honor assumption with reasonable costs.

II. FAULT MODEL AND FAILURE VIRTUALIZATION

In this section, we introduce the non-malicious arbitrary fault model and show how failure virtualization allows benign fault-tolerant algorithms to tolerate arbitrary faults in non-malicious settings. We start with the system and fault models on which we base our new fault model.

A. System Model and Fault Models

A distributed system is divided in two parts: the distributed algorithm and the environment. A distributed algorithm is a set of processes that communicate via message-passing. Processes run on hosts, which send and receive messages on behalf of processes via a network. Hosts and network are together referred to as the environment. Processes and environment are the components of the system. Each component is modeled as a state machine, which consists of a state and state transitions. The system state encompasses the state of all its components, and an execution is a sequence of system states given by component transitions starting at some initial state.

A fault model is typically a set of state transitions, called faults, added to the components of the system. Faults can cause errors, which are abnormal states of components reachable via faults. If an error becomes visible, the component commits a failure. Examples of failures are process
crashes, message corruption, etc. Note that a failure of one component is a fault to another component.

Since no distributed algorithm can tolerate an unbounded number of faults, one typically assumes a limit to the fault extent, e.g., at most \( f \) processes might crash. Under such an assumption, an algorithm is \( F \)-fault-tolerant if every execution of the system with the fault model \( F \) satisfies the algorithm properties.

1) Benign Fault Model: Processes might crash and the environment might lose, duplicate, reorder, or misroute messages in this model. Process \( p \) commits a crash failure by performing a crash transition to some halt state from which no further transition of \( p \) is possible. If not crashed, process \( p \) is said to be correct. Note that misrouting can be considered a benign fault since it can be easily detected if the sender appends to each message sent its process identifier. In the remainder of this paper, we consider only benign fault-tolerant algorithms.

2) Arbitrary Fault Model: In addition to benign faults, this model extends components with arbitrary faults, which are transitions to any representable state of a component, e.g., corruption of a variable in a process, substitution of the operating system image of a host, etc. A component might fail arbitrarily by sending arbitrary messages, also known as commission failures\(^2\). Arbitrary transitions represent not only physical but also malicious faults: a hacker could understand the format of the messages exchanged in the algorithm, and then force transitions in a compromised component to forge algorithm messages, possibly violating algorithm properties. Note that this arbitrary model is more general than PBFT’s fault model\(^9\) since it also allows the environment to fail arbitrarily.

In an execution subject to arbitrary faults, a process \( p \) can be either crashed, faulty, contaminated, or correct (see Table I). A process \( p \) is correct if it does not fail and becomes crashed if it fails by performing a crash transition. Whether a process \( p \) becomes faulty or contaminated depends on which type of fault caused its failure: an arbitrary transition or an arbitrary message. A process \( p \) becomes faulty if it sends arbitrary messages due to an arbitrary transition performed by \( p \). A process \( p \) becomes contaminated if it sends arbitrary messages due to a transition processing a received arbitrary message.

\(^2\)Arbitrary faults can also manifest as omission failures\(^{23}\). We attribute omission failures to the environment.

### B. Non-malicious Arbitrary Fault Model

The non-malicious arbitrary fault model is the arbitrary fault model restricted by the assumption of two properties: no impersonation and honor. Executions in our model (i.e., satisfying these properties) are called non-malicious executions (see Fig. 1); executions not in our model but in the arbitrary fault model are called malicious executions. We argue that the probability of malicious executions is negligible if the system is not subject to malicious adversaries. The non-maliciousness assumptions (i.e., no impersonation and honor) have, therefore, a high assumption coverage.

Intuitively, the no impersonation property states that the environment never behaves like a correct process; and the honor property states that one can detect when a process is faulty. To define these properties more precisely, we introduce the concept of invalid message. An invalid message is a message that is arbitrary, but this fact is trivially detected by inspecting the message itself – Powell similarly defines non-code value errors in\(^{12}\). A message is otherwise a valid message. Note that the validity of a message should be more than a checksum protecting the message against corruption; it represents an evidence of good behavior of the process creating the message. Detection of bad behavior can be achieved by transforming the process with redundancy and fault detection such as arithmetic codes\(^{15, 17}\).

**Property 1 (No Impersonation):** The environment never creates valid messages (except duplicates).

No impersonation has a low assumption coverage in systems subject to malicious adversaries. A hacker could always analyze the program to determine the set of valid messages and then initiate fault transitions in a compromised component, resulting in arbitrary but valid messages, i.e., commission failures. However, if the system is never subject to attacks, which is our focus in this paper, arbitrary faults have a negligible probability of producing a valid message. A message encompasses header fields, checksum, payload, validity bits, etc. Creating an arbitrary but valid message out of the blue would require one or more highly improbable fault transitions in the environment. Note that valid messages might still become duplicated, but duplicates are tolerated since they are benign faults.

**Property 2 (Honor):** A faulty process never creates valid messages.

The honor property holds if arbitrary faults are always reflected in the message validity. We experimentally show that arithmetic codes can enforce honor with a high probability in Sect. IV. In other words, in systems that are not subject to malicious attacks, honor has a high assumption coverage if processes are transformed with the approach described in Sect. III. In the following, an honorable process is a process enhanced with fault detection mechanisms such that it can satisfy the honor property in non-malicious settings.

<table>
<thead>
<tr>
<th>Classification</th>
<th>Failure</th>
<th>Fault</th>
</tr>
</thead>
<tbody>
<tr>
<td>correct</td>
<td>no crash</td>
<td>crash transition</td>
</tr>
<tr>
<td>crashed</td>
<td>arbitrary message</td>
<td>arbitrary message</td>
</tr>
<tr>
<td>faulty</td>
<td>arbitrary message</td>
<td>arbitrary message</td>
</tr>
<tr>
<td>contaminated</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### Table I

**Process Classification According to Failure and Fault**
C. Virtualizing Arbitrary Failures into Benign Failures

We define failure virtualization to be the mapping of failures that are possible in one fault model into failures that are possible in another fault model. Now note that no impersonation and honor properties together imply that any arbitrary message produced by a faulty process or the environment is invalid. Assuming that both properties hold, one can easily virtualize arbitrary failures into benign failures by adding a filter action to each process.

Property 3 (Filter Action): A correct process drops any invalid message without processing it.

Now consider a distributed algorithm that tolerates benign faults. We claim that the same algorithm tolerates arbitrary faults in the non-malicious arbitrary fault model if processes perform the filter action. Remember that a process becomes contaminated when it violates the specification due to the processing of an arbitrary message (see Table I). However, as long as the environment does not impersonate processes (Property 1), and faulty processes are honorable (Property 2), contamination is not possible because correct processes filter invalid messages (Property 3). Hence, correct processes follow their state transitions: they only process and, consequently, create correct messages.

Since contamination is not possible, any non-malicious execution can be mapped into an execution with benign faults in two steps: First, because correct processes ignore all messages from faulty processes, the latter can be mapped as crashed processes. Second, invalid messages can be mapped to message omission failures, which are benign faults as well. It follows from this argument that Paxos [21] can solve consensus with 2f + 1 processes in the non-malicious arbitrary fault model.

D. Extensions of the Model

Our models allow executions where liveness properties are not guaranteed. As in traditional fault models, further assumptions have to be made to satisfy these properties such as a global stabilization time [24] or a timed-asynchronous system model [25]. Moreover, if safety properties of the algorithm rely on a clock, one has to additionally assume that the clock is correct, i.e., has a maximum drift rate. In such a case, the benign fault model also includes process and message performance failures. Note that techniques to build Byzantine fault-tolerant clocks [26] can be used to enforce such assumption.

III. Honor Enforcement with Arithmetic Codes

In this section, we present a framework for building distributed algorithms that automatically transforms processes in honorable processes (Property 2) by means of arithmetic codes, a technique to detect a variety of transient and permanent hardware errors with a high probability [15], [17]. To perform the transformation automatically, we embed and adapt the encoding compiler by Schiffer et al. [16] in our framework. In this way, we extend the use of encoding to support distributed algorithms. We start by describing the framework and the encoding transformation. Next, we explain how failure virtualization is achieved in practice. Finally, we discuss costs and optimizations of our approach.

A. Encoding Framework for Distributed Systems

A process of a distributed algorithm is programmed as a module in C language using the framework’s interface to communicate with the external world. Processes are identified with an integer id. State variables are global variables inside the process module. A message m is sent to a destination dst via send(dst, m) method. Translation between ids and IP addresses and the message transmission itself are handled in the framework’s event loop. When the event loop receives a message m from the network, it calls the recv(m) method implemented by the process. Alarms are scheduled by calling alarm(aid, t) with an alarm number aid and a time t. Once time t is reached, the event loop calls trig(aid) implemented by the process. Time is measured via the clock(t) method.

The process module is compiled and linked together with the framework’s library as the final program. With no source-code modification, honorable processes are created by compiling the process with an encoding compiler, and subsequently linking it with the framework’s event loop and a wrapper module. We use the encoding compiler by Schiffer et al. [16], which can transform programs with the following arithmetic codes: AN-, ANB-, and ANBD-codes. Each arithmetic code can detect a different class of errors at a different runtime cost. We have modified the encoding compiler such that only the algorithm’s process and part of the wrapper are encoded, both together being called encoded process. Figure 2 depicts an honorable process receiving and sending messages in our framework – the subscript c represents an encoded method or value. State and methods inside the shaded area are all encoded. The wrapper intercepts calls to and from the process and en-/decodes arguments allowing the communication between encoded processes. By selecting different arithmetic codes, our framework allows one to trade fault coverage for CPU cycles depending on the system requirements. In Sect. IV we evaluate error detection and performance overhead of algorithms running with honorable processes. We now explain how programs are transformed by giving the example of AN-code.

AN-code Example: The encoding compiler AN-encodes the value of each state variable x of a program by multiplying its original functional value xf with a constant A. The resulting value is called xc. As depicted in Fig. 3, the domain of an encoded variable is divided into a few valid code values (multiples of A) and many non-code values (not multiples). One can easily check whether xc is a code value or not by applying the modulus operation on xc with A. If the result is zero, then the value is
code, otherwise the value is non-code. In case a fault flips bits of \(x_c\), the resulting value \(x'_c\), is not multiple of \(A\) with a high probability, approximately \(1 - \frac{1}{A}\) \[17\]. The encoding compiler additionally transforms each operation of the program into an encoded operation. Encoded operations take encoded values as arguments and produce encoded values as output. Figure 3 illustrates the ideal three properties of an encoded operation: (1) if input arguments are code values and no error occurs, then the operation result is a code value; (2) if at least one argument is non-code, then the operation result is non-code; and, (3) if an operation is faulty, then its result is non-code. As an example of how the code is preserved, consider an encoded addition. If \(A = 7\), then instead of adding the two values 2 and 3 the encoded program adds 14 and 21. The result 35 is again a multiple of \(A\) because \(x_c + y_c = x_f \cdot A + y_f \cdot A = (x_f + y_f) \cdot A\). If the addition is faulty or one of the arguments is non-code, the result is non-code with high probability. By default the encoding compiler uses \(A = 65521\).

B. Failure Virtualization

The key to virtualize arbitrary failures in benign failures is to guarantee that correct processes do not modify their state based on arbitrary messages. In our approach, messages carry an evidence of good behavior of the process that created them. Since messages are nothing else than variables in the encoded process, a valid message simply is a correctly encoded array of values. Hence, errors invalidate outgoing messages with high probability due to the properties in Fig. 3. Messages are protected on transmission as well because they are kept encoded during their whole life-time (see Fig. 2).

Remember that benign failures are crashes, message duplication, message misrouting, and omission failures. The wrapper discards any invalid message received, virtualizing the arbitrary failure into an omission failure. The wrapper also tries to abort before sending arbitrary messages out, virtualizing the arbitrary failure into a crash failure. Note that our approach guarantees that if an arbitrary message is nevertheless sent out, it is invalid with a high probability.

The destination \(dst_c\) of a message is decoded by the wrapper in order to be translated by the event loop into an IP address (\(dst_{IP}\) in Fig. 2). If \(dst_f\) is modified by an arbitrary fault, the message is simply misrouted; a benign failure. Similar virtualization is used for the alarms: the alarm identifier (\(aid_c\)) is kept encoded in the event loop as a 64-bit integer. If an arbitrary fault modifies the alarm identifier, this is detected by the wrapper and the process is aborted. The wrapper decodes however the time \(t_c\) and sets the alarm in the event loop. If the scheduled time \(t\) of an alarm is modified by a fault, the alarm is triggered either too early or too late. Performance failures are benign failures (Sect. II-D). Note that the encoded part of the wrapper can detect out of order alarms by book-keeping the \(id\) of the next alarm to be triggered.

C. Costs and Optimizations

Encoding is known to be computationally expensive \[16\]. In our framework only the algorithm’s processes and part of the wrapper are encoded; the remaining framework code, libraries, network stack, operating system are left untouched. One could further mitigate the cost of encoding by exploiting multiple processors \[27\]. Notwithstanding, we use a single thread above the event loop to be able to precisely measure the performance overhead of our approach.

Encoding incurs a higher network bandwidth utilization. The compiler transforms each 32-bit word in a 64-bit word, blowing up message sizes by at least a factor of two. We propose a bandwidth optimization that can reduce the per-message overhead to a constant size using checksums, \(e.g., 32\)-bit CRC, to protect messages on transmission. Our optimization follows:

1) When \(\text{send}_c\) is called with a message \(m_c\), the wrapper calculates the checksum \(s\) of \(m_c\).

2) The wrapper decodes \(m_c\) and forwards \(m \cdot s\) to the framework’s event loop instead of \(m_c\).

3) The message is sent to the destination via the network.

4) When message \(m \cdot s\) is received from the network, the wrapper encodes \(m\) and calculates the \(s'\) of \(m_c\).

5) The (encoded) wrapper compares whether \(s = s'\), and calls \(\text{recv}_c(m_c)\) case true; otherwise, drops \(m\).

The message is still protected end-to-end because checksums are calculated over encoded data and compared inside the encoded wrapper.
IV. EXPERIMENTAL EVALUATION

In this section, we experimentally evaluate our framework focusing on two questions. (1) how costly is the encoding for distributed algorithms? And (2) what is the probability that honor property is violated, i.e., what is the probability of valid but arbitrary messages being sent by a process? To answer these questions, we experimentally compare the performance of different variants of two important distributed algorithms: Paxos and strong leader election. Next, we inject faults in a process with the EIS fault injector [28] and measure the probability of invalid messages being sent.

A. Algorithms, Metrics, and Setup

1) Paxos: We implemented Multi-Paxos, an extension of the Paxos algorithm [21], which achieves consensus on a sequence of values, and is widely used as building block for replicated systems [6], [22]. In our implementation processes are either clients, proposers, or acceptors. A proposer orders requests from clients, and sends them to acceptors, which in turn log the requests and reply the proposer. If the proposer receives a reply from a majority of acceptors, it sends an ack message to the client containing the identifier of the chosen request.

Acceptors run in exclusive hosts. We experiment with 3 and with 5 acceptors, i.e., maximum number of faulty processes is \( f = 1 \) and \( f = 2 \), respectively. These numbers of acceptors represent common choices in practical systems [5], [6]. We run a single client and a single proposer collocated on the same host. The client generates requests in a fixed interval of time given by a load parameter. We allow the client to send at most 200 concurrent requests. The client batches requests up to 1 MB if the limit of concurrent requests is reached.

2) Strong Leader Election: We implemented the strong leader election algorithm by Fetzer and Cristian [20]. “Strong” refers here to the safety property satisfied by the algorithm: there are never two leaders at the same instant of time. Our implementation follows closely the algorithm described in their paper. The support number is set to a majority, i.e., a process only becomes leader if it receives timely support from a majority of processes. Our experiments run with three processes, each of them on a different host. Once a process becomes leader, the remainder processes are called slaves. We vary how many heart-beats per second are sent via the election period (EP) parameter. The expires parameter, which determines the crash detection time, is set to \( 4 \cdot EP \).

3) Environment Setup and Variants: All of our experiments were performed in 6 workstations with 2 quad-core 2.0 GHz Xeon processors, 8 GB of RAM, and Gigabit Ethernet interface. The measured maximal bandwidth of a host in our cluster is 944 Mbit/s. Our settings are in ideal conditions, i.e., processes do not crash, links are up and timely, and there are no other jobs running on the machines. Our CPU utilization measurements focus on the thread performing the upcalls from the event loop to the algorithm’s process. We use rusage method and consider only the user time.

We experiment the following variants: native is the variant without any encoding; an, anb and anbd are compiled with AN- , ANB- and ANBD-code and bandwidth optimization (see Sect. III-C); and finally, an-naive, anb-naive and anbd-naive are the same variants without bandwidth optimization. ANBD-encoded Paxos could not be correctly compiled. We are still investigating the causes.

B. Paxos: Network Bandwidth

We show that the encoded variants of Paxos can reach the network bound. Figure 4 depicts the maximum goodput each variant can achieve with 3 and 5 acceptors and a request payload size of 1 kB – a typical value, used for example in [5]. Goodput is the throughput at the application level, i.e., 1000 requests per second result in a goodput of 1 MB/s.

Since multicast is performed in software – as in systems such as ZooKeeper [5] – the proposer has one third of its bandwidth with three acceptors, and a fifth with five acceptors. The results in Fig. 4 are consistent with this observation: the maximal goodput native achieves is about 35 MB/s and 20 MB/s with 3 and 5 acceptors respectively.
Our \textit{an}, and \textit{anb} variants use as much bandwidth as \textit{native} except of 4 extra bytes for CRC per message. The results show that \textit{an} reaches the maximum goodput for both 3 and 5 acceptors. \textit{anb} also achieves the maximal goodput with 5 acceptors and reaches about 23 MB/s with 3 acceptors. By increasing batch sizes and the concurrent request limit, \textit{anb} would possibly reach the network limit. \textit{an-naive} and \textit{anb-naive} are network bound, but with half of the \textit{native}'s goodput since they consume twice as much bandwidth with encoded messages.

C. Paxos: Performance under Nominal Load

In practice replicated systems work below the system's limit if responsiveness is critical. We now configure the client with target loads that range from 1000 up to 5000 req/s with 5 acceptors with payload size of 1kB. Figure 5 shows the latency and CPU utilization of our Paxos variants. Under a load of 1000 req/s the mean latency and its standard deviation is 4.38 ± 0.18 ms for \textit{native}, 21.74 ± 3.04 ms for \textit{an}, and 39.87 ± 5.10 ms for \textit{anb}. Now observe (Fig. 5, left side) that for \textit{an} the latency increases slower from 2000 req/s on and for \textit{anb} from 4000 req/s on. At these points the client starts batching its requests. For this load range, the latency of \textit{an} is from 15 ms to 50 ms higher than \textit{native} variant; while the latency of \textit{anb} is from 35 ms up to 100 ms higher.

Encoding incurs a high CPU utilization as expected, as depicted in Fig. 5 (right side). Note that, even though the CPU utilization is close to 100%, the high goodput presented above is achieved because the proposer and acceptors do not process but rather only store the batched payloads. The execution of the requests is outside the scope of the Paxos algorithm.

D. Strong Leader Election Performance

We evaluate the performance of our leader election variants. The results show that encoding incurs no costs on election time and an acceptable cost on CPU utilization. Table II depicts the mean election time ($\mu$) in milliseconds and its standard deviation ($\sigma$) for different EP values and different variants of the algorithm. Each mean is the aggregation of 5 runs. The results show insignificant difference of election time for the selected EP parameters, with variations between 1 and 2 ms for all EP values.

Table III shows the mean CPU utilization for the same experiments. We separate the results between the leader and one slave process. We see the direct correlation between the election period and the CPU utilization. In general, the encoded variants incur in a higher CPU utilization than \textit{native}, which remains close 0% for all parameter values. For example, for $EP = 50$, the mean CPU utilization is 0.3% for \textit{native}, 1.8% for \textit{an}, 3.2% for \textit{anb}, and 11% for \textit{anbd}. The robust \textit{anbd} variant presents measurements from 1.24 up to 10.88% in the leader and 0.41% up to 3.08% in the slave, which are acceptable CPU utilization values for many applications.

Table I

<table>
<thead>
<tr>
<th>EP (ms)</th>
<th>native $\mu$ (ms) $\sigma$</th>
<th>an $\mu$ (ms) $\sigma$</th>
<th>anb $\mu$ (ms) $\sigma$</th>
<th>anbd $\mu$ (ms) $\sigma$</th>
</tr>
</thead>
<tbody>
<tr>
<td>50</td>
<td>64.57 ± 1.56</td>
<td>62.79 ± 0.99</td>
<td>63.45 ± 1.68</td>
<td>64.23 ± 1.11</td>
</tr>
<tr>
<td>200</td>
<td>204.71 ± 1.59</td>
<td>204.30 ± 1.65</td>
<td>205.81 ± 0.72</td>
<td>204.00 ± 1.01</td>
</tr>
<tr>
<td>300</td>
<td>305.73 ± 0.80</td>
<td>305.49 ± 1.36</td>
<td>305.12 ± 0.97</td>
<td>304.26 ± 0.73</td>
</tr>
<tr>
<td>400</td>
<td>404.49 ± 0.74</td>
<td>406.52 ± 1.94</td>
<td>405.65 ± 0.52</td>
<td>404.20 ± 0.44</td>
</tr>
</tbody>
</table>

Table II

<table>
<thead>
<tr>
<th>EP (ms)</th>
<th>leader CPU (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>50</td>
<td>0.28 ± 1.82</td>
</tr>
<tr>
<td>200</td>
<td>0.12 ± 0.62</td>
</tr>
<tr>
<td>300</td>
<td>0.06 ± 0.32</td>
</tr>
<tr>
<td>400</td>
<td>0.04 ± 0.22</td>
</tr>
</tbody>
</table>

E. Fault Injection in a Proposer

We conclude our evaluation with a fault injection experiment. We inject a total of 30,000 different faults in a proposer process of Paxos. For that end we modified the wrapper so that it first stores in disk a 30 s trace of the proposer’s interaction with the framework interface, \textit{i.e.}, its downcalls and upcalls. For each variant we save such a trace and replay it 10,000 times, injecting one fault in a random place per replay. Whenever calls from the process to the framework diverge from the logged trace, the wrapper aborts the process.

Table IV shows, for different fault types, the percentage of arbitrary failures in relation to the total number of failures; and in parenthesis the absolute relation for the 2000 injections. The fault types used in these experiments represent the software-level symptoms caused by hardware errors – see [28], [29] for a more detailed description of these fault types. Arbitrary failures are valid messages sent by the proposer that were not sent in the saved trace. Note that, while only the correctly encoded messages are valid for encoded variants, all messages are valid for the \textit{native} variant. The total number of failures encompasses all failures: crashes, omissions, arbitrary messages, and incorrectly encoded (invalid) messages.

These results give the probability of honor violation given that a failure occurred due to an arbitrary fault. \textit{an} processes violate honor in 1.27 up to 3.32% of the failures; \textit{anb} processes violate honor in about 0.3% of the failures; and finally, \textit{native} processes violate honor in 5 up to 50% of the failures.

F. Discussion

The fault injection results corroborate the hypothesis that arithmetic codes can enforce honor property: One can use
a benign fault-tolerant algorithm, automatically transform it in our framework, reach the same bandwidth utilization, and reduce the risk of violating honor from 16% down to 0.34%. The price of enforcing honor is higher CPU utilization and request latency.

Notwithstanding, one can run a highly fault-tolerant anbd variant with CPU utilization from 1 up to 11% and achieve the same election times as with a native variant. an and anb variants utilize even less CPU: up to 1.8% and up to 3% respectively. Our measurements have shown acceptable latencies with nominal load, varying from 20 to 60 ms for an, and from 40 to 105 ms for anb. In real-world Paxos implementations, requests are written to a physical disk, and replies are first sent once the write operation has finished, which might take from 10 ms up to 70 ms, depending on disk technology and on durability guarantees. We believe this latency would partially hide the encoding overhead, but experiments have to be performed to support this claim.

In this paper we focused on the injecting faults in a proposer of the Paxos algorithm, which is by far the most intricate process we have implemented. Our fault injection results are consistent with previous work [16], indicating that these results can also be extrapolated to other algorithms.

### V. Related Work

Bhatotia et al. [13] also consider arbitrary-fault tolerance an overkill for systems running in data centers: Such fault tolerance mechanisms assume malicious adversaries, although data centers are already protected by firewalls. The non-malicious arbitrary fault model proposed here captures the arbitrary faults reported to occur in commodity-hardware data centers [1].

Our work uses the encoding compiler by Schiffel et al. [16] to enforce honor. Their work focus on hardware errors in safety-critical systems, while our work focus on distributed systems. We not only integrated their compiler in our framework, but also extended it with wrappers to perform communication between processes, to trigger alarms, and to virtualize arbitrary failures (see Sect. III).

Pattabiraman et al. [30] defined a memory model that allows programmers to manually specify critical variables, which are then protected against corruptions (i.e., hardware errors) with forward correction. Our approach, in contrast, automatically transforms distributed algorithms, detecting data corruption and control-flow errors without correcting them. The overhead of encoding could be reduced by using programmer knowledge in a similar fashion as Pattabiraman et al. have done.

In the context of distributed systems, Bhatotia et al. [13] also focus on arbitrary faults in non-malicious settings. Their approach is, however, restricted to a specific set of distributed applications, namely, MapReduce jobs programmed in the Pig framework. More recently, Correia et al. have presented the ASC fault model and the PASC library [31], which locally duplicates each process and executes each message on both replicas. Similarly to our approach, theirs also allows faulty processes to send invalid messages. Nevertheless, while PASC assumes a single transient fault during the processing of a message, our approach detects multiple faults, including permanent faults, which are reported to occur often [1], [2].

Automatic transformations for fault tolerance was first developed by Coan [10] and later by Neiger and Toueg [11]. In contrast to the former our approach is not restricted to algorithms working in a “particular standard form”, i.e., round-based algorithms. In contrast to the latter, our approach is not restricted to synchronous systems. More recently, Ho et al. [19] have transformed algorithms by assigning guard hosts to each process. Besides extra communication, their approach requires that each guard runs a copy of the hosts it observes. Our approach neither requires extra messages nor simulates other processes.

Finally, researches have reduced the number of replicas necessary to solve consensus in the arbitrary fault model by using customized trusted hardware, e.g., a trusted counter, and adapting the algorithms to use such extensions [32], [33]. Our approach neither requires extra hardware components nor modifies the algorithm. A benign fault-tolerant consensus algorithm, when implemented in our framework, can still solve consensus with $2f + 1$ replicas.

### VI. Conclusion

This work presented the non-malicious arbitrary fault model (Sect. II), a fault model specially suitable for environments protected by firewalls. Our fault model captures a wide subset of arbitrary faults, such as transient and permanent hardware errors, which often occur in commodity-

### Table IV

<table>
<thead>
<tr>
<th>fault (error)</th>
<th>arbitrary failures/total failures</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>native</td>
</tr>
<tr>
<td>faulty operation</td>
<td>5.82%</td>
</tr>
<tr>
<td>modified operand</td>
<td>5.66%</td>
</tr>
<tr>
<td>exchanged operand</td>
<td>10.85%</td>
</tr>
<tr>
<td>lost store</td>
<td>49.25%</td>
</tr>
<tr>
<td>total</td>
<td>16.35%</td>
</tr>
</tbody>
</table>
hardware data centers. In our model benign fault-tolerant distributed algorithms can tolerate arbitrary faults as long as processes are honorable and filter invalid messages. We developed a compiler-based framework (Sect. III) to demonstrate that these conditions are achievable with little effort, i.e., requiring no source-code modification.

Our experiments illustrate the possible trade-off between fault coverage and CPU utilization with two important distributed algorithms. Encoded Paxos variants show virtually no network overhead: they reach the same request throughput as the native variant with 5 acceptors at the cost of extra CPU cycles. Under nominal load, encoded variants of Paxos also provide reasonable request latencies, varying from 20 to 105 ms depending on the desired coverage and load. The encoded variants of the leader election provide excellent election times, while consuming at most 11% more CPU cycles. Finally, our fault injection results suggest that arithmetic codes can successfully enforce the honor property: An encoded Paxos proposer has its probability of violating honor (given that a failure occurred) decreased two orders of magnitude, from 16% to about 0.34%.

We are currently working on a new arithmetic code: The code will minimize the performance overhead while maintaining the same or even decreasing the probability of honor violation. Moreover, we are designing a modified Paxos algorithm that works with a subset of the processes being encoded, improving in this way the performance even further.

ACKNOWLEDGMENT

The authors would like to thank Martin Nowack and Martin Süßkraut for their helpful comments and André Schmitt for his support with the encoding compiler. This work was partially supported by the German Excellence Initiative, Center for Advancing Electronics Dresden (cfAED), Resilience Path.

REFERENCES


