ZERMIA - A Fault Injector framework for testing Byzantine Fault Tolerant protocols*

João Soares[1,2][0000−0002−0607−636X], Ricardo Fernandez[1], Miguel Silva[1], Tadeu Freitas[1][0000−0001−5573−2434], and Rolando Martins[1,2][0000−0002−1838−1417]

[1] Faculdade de Ciências, Universidade do Porto, 4169-007 Porto, Portugal
[2] CRACS - INESC TEC, 4200-465 Porto, Portugal {joao.soares,rmartins}@fc.up.pt

Abstract. Byzantine fault tolerant (BFT) protocols are designed to increase system dependability and security. They guarantee *liveness* and *correctness* even in the presence of arbitrary faults. However, testing and validating BFT systems is not an easy task. As is the case for most concurrent and distributed applications, the correctness of these systems is not solely dependant on algorithm and protocol correctness. Ensuring the correct behaviour of BFT systems requires exhaustive testing under real-world scenarios. An approach is to use fault injection tools that deliberate introduce faults into a target system to observe its behaviour. However, existing tools tend to be designed for specific applications and systems, thus cannot be used generically.
We argue that more advanced and powerful tools and frameworks are needed for testing the security and safety of distributed applications in general, and BFT systems in particular. Specifically, a fault injection framework that can be integrated into both client and server side applications, for testing them exhaustively.
We present ZERMIA, a modular and extensible fault injection framework, designed for testing and validating concurrent and distributed applications. We validate ZERMIA's principles by conduction a series of experiments on a distributed applications and a state of the art BFT library, to show the benefits of ZERMIA for testing and validating applications.

Keywords: Fault Injection · System Validation · System Testing · Testing · Fault · Byzantine Fault Tolerance · Fault Tolerance Systems · Distributed Systems

1 Introduction

Byzantine fault tolerant (BFT) protocols [5, 6, 10, 2, 3, 27, 33] are designed to increase system dependability and security. They guarantee liveness and correctness even in the presence of arbitrary faults, including not only hardware/software faults, but also malicious actors that try to compromise systems/applications by subverting their protocols.

BFT consensus protocols have also been adopted as the consensus mechanism used by different private blockchain systems [40, 45]. These protocols replace the original and resource intensive proof-of-work based ones, with more efficient and high-performing ones. However, any compromises to their design and implementations can lead to a compromise of the underlying blockchain and application.

BFT systems are commonly theoretically proofed on their correction. Still, for these systems to be efficiently implemented in practice, some level of compromise is commonly needed. These compromises may invalidate the protocol correctness if not thoroughly tested and validated, and could be taken advantage of by highly skilled opponents. For example, many assumptions on liveness are based on predefined timeouts that, when misconfigured, can compromise the protocol's efficiency [31].

However, testing and validating BFT systems is not an easy task. As is the case for most concurrent and distributed applications, the correctness of these systems is not solely dependant on algorithm and protocol correctness but is also influenced by the reliability and latency of communications. This is aggravated when the protocol itself needs to deal with the possibility of agents trying to subvert the protocol. Thus, ensuring the correct behaviour of BFT systems requires exhaustive testing under real-world scenarios.

* This work was partially funded by POCI-01-0247-FEDER-041435 (SafeCities), POCI-01-0247-FEDER-047264 (Theia) and POCI-01-0247-FEDER-039598 (COP) financed by Fundo Europeu de Desenvolvimento Regional (FEDER), through COMPETE 2020 and Portugal 2020. Rolando Martins was partially supported by project EU H2020-SU-ICT-03-2018 No. 830929 CyberSec4Europe.

1.1 Fault injection

An approach to test these systems in real-world scenarios is to use fault injection tools [20, 4, 22, 41]. Fault injection is the deliberate introduction of faults into a system to observe its behaviour. This is a valuable approach to validate the dependability properties of systems and applications. [1, 30].

Fault injection tools typically fall into hardware-based, software-based, or hybrid hardware/software approaches. Software fault injection tools are typically designed to introduce faults either during pre-runtime (e.g., compile -time, executable image manipulation [20])or at runtime [22, 41].

However, existing *Fault Injection* (FI) tools tend to be designed and implemented for specific applications and systems, thus cannot be used generically by any application. In fact, most FI tools are not designed for concurrent applications and systems, as such, are unable to deal with the requirements presented by these applications. Namely, inject faults on specific threads or processes, coordinate faults between threads and processes, and/or deal with non-deterministic execution flows that may occur in this class of applications. We argue that more advanced and powerful tools and frameworks need to be designed, for testing the security and safety of distributed applications in general, and BFT systems in particular.

To this end we present a brief study on one of the most influential BFT protocols, PBFT [5, 6], to identify the requirements imposed by such protocols when designing a fault injector framework. Additionally, we present the design of a modular and extensible fault injection framework (called ZERMIA), that allows developers to use generic and/or design/implement tailor-made fault for testing their systems/application, without requiring modifications to the original systems.

The remainder of this paper is organized as follows: Sections 2 and 3 presents relevant fault injection tools and frameworks, and PBFT's agreement protocol. Section 4, 5, and 6 present the design, implementation and evaluation of ZERMIA; and, Section 7 concludes this paper by presenting some concluding remarks, lessons learned and future work.

2 Related Work

Software fault injection (SFI) [35] is a technique for testing and assessing the dependability of software systems. Ferrari [24] uses software traps to inject CPU, memory, and bus faults. It uses a parallel daemon process running on the host processor to control the application in which the faults need to be injected. It is capable of injecting faults in the address, data or the control lines of the processor.

In Ftape [42] bit-flip faults are injected into user-accessible registers in CPU modules, memory locations, and the disk subsystem. DOCTOR [20] allows injection of CPU faults, memory faults, and network communication faults. It uses three triggering methods to indicate the start of fault injection. These methods are time-out, trap, and code modification. Xception [4] takes advantage of advanced debugging and performance monitoring features present in modern processors to inject faults. In particular, it uses built-in hardware exception triggers to perform fault injections. This requires modification of the interrupt hardware vector. In FIAT [39] an approach to inject single bit flips in the instruction memory of the application was proposed.

The PROPANE (PROPagation ANalysis Environment) [21] tool injects faults in C code executing on desktop computers. The code is instrumented with fault injection mechanisms and probes for logging traces of data values and memory areas. PROPANE supports the injection of both software faults (by mutation of source code) and data errors (by manipulating variable and memory contents). Another tool that takes advantage of processor debugging facilities is MAFALDA [14]. This tool has aimed to evaluate the use of COTS micro-kernels in safety-critical systems.

Loki [7] is a fault injector for distributed systems that injects faults based on a partial view of the global system state. Loki allows the user to specify a state machine and a fault injection campaign in which faults are triggered by state changes.

FIRE [30] and JACA [29] use reflective programming to inject and monitor C++ and Java applications respectively can change the values of parameters, attributes and returned variables. J-SWFIT [36] is a framework for Java programs that provide two fault types.

PIN [23] dynamically inserts code at runtime allowing a fault to be injected at a specific code location which allows recreating faults that result from common coding errors. LLFI [28] and Relyzer [37] use compiler based techniques to inject code into the compiler for intermediate representation and to determine the equivalence of different fault sites in a target application.

Duraes et al. [12] study on bug fixing in open-source and commercial software systems has found that many software faults involve several program statements. More recently, study [11] on software bugs in OpenStack found that the fault model proposed by Duraes et al. [12] cannot cover some frequent bug types in distributed systems.

Gunawi et al. [19] describe a framework for systematically injecting sequences of faults in a system under test. The authors present Failure IDs as a means of identifying injection points and describe a method for systematically generating injection sequences. A take from this work is that relevant and efficient testing requires users to know the target application' specification for estimating relevant testing cases and if its behaviour is correct even under injection.

3 BFT Protocol Overview

Byzantine Fault Tolerant (BFT) protocols are designed to guarantee the correct system/application behaviour, even in the presence of arbitrary faults (i.e., Byzantine faults). Typically, BFT protocols use a state machine replication (SMR) approach [25, 38], where multiples instances of the same state machine replicate the application's state. SMR guarantees that if all state machines (replicas) start in the same consistent state and execute the same set of operations in the same order, all correct machines will reach the same final state, ensuring state consistency among replicas (as long as state changes are deterministic).

BFT protocols are designed to ensure both *safety* and *liveness*. *Safety* ensures that correct replicas that execute the same set of operations in the same order reach the same final state and produce the same result, while *liveness* guarantees that correct replicas eventually execute all requests from correct clients.

Nodes (i.e., clients or replicas) communicate with each other by sending messages through a network that might drop, corrupt, or delay messages. No assumptions are made concerning the latency for delivering messages, as these systems are designed to ensure safety even in the absence of an upper bound on network and processing delays. However, to overcome the FLP impossibility [15] synchronous phases are required to guarantee liveness [8, 13]. Nodes authenticate all messages they send either by using signatures (e.g., RSA [34]) or message authentication codes (MACs) [43, 5].

A correct consensus protocol guarantees the following properties: *i*) Termination - correct replicas eventually learn about some decided value; *ii*) Agreement - correct replicas must agree on the same value; *iii*) Validity - if correct replicas propose a value, then all correct replicas eventually decide on that value; *iv*) Integrity - a correct replica can only decide on one value. If a correct replica decides on a value, then some other correct replicas must have proposed that value.

Faulty nodes (clients or replicas) are assumed to fail arbitrarily, possibly by: failing to send messages to each other; sending incorrect values; or, sending conflicting messages with diverging contents. Typically, no conjectures are made on the fault origins. Thus, there is no distinction between faulty replica behaviour caused by software bugs or replicas failing as the result of a successful malicious intrusion. In addition, nodes cannot determine whether a node is faulty or simply advances at a slower pace since they have no knowledge of the occurrence and duration of network synchrony. Finally, multiple compromised nodes are expected to potentially collude with each other. However, faulty nodes are assumed to not be able to break cryptographic primitives and consequently cannot impersonate correct nodes.

3.1 PBFT Protocol

PBFT [5, 6] is the first practical BFT solution designed to work in asynchronous environments (i.e., the Internet). It is a leader-based SMR-BFT protocol that requires $N = 3f + 1$ replicas for guaranteeing *liveness* and *safety* in the presence of up to f faults. Assumes a partial synchronous network communication model and ensures node and message authentication based on either MACs or digital signatures.

Agreement protocol Figure 1 presents the PBFT's agreement protocol, consisting of the following steps [1]:

1. Clients send REQUEST messages to all replicas. These messages contain the operation to be executed, a timestamp (used for once-only semantics), and the client's identifier.

[1] All messages exchanged during this protocol are signed by the respective sender for authentication purposes. Validating messages includes validating the message signature.

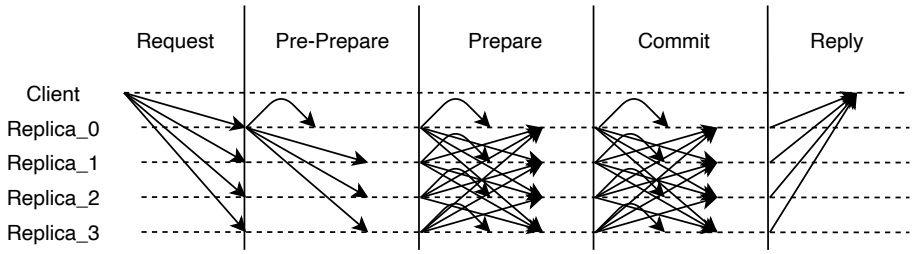


Fig. 1: BFT-SMaRt agreement protocol

2. When a primary receives a REQUEST message, it authenticates the message. If valid, a PRE-PREPARE message is sent to all replicas. PRE-PREPARE messages include the current view number, a sequence number (assigned by the primary), and the client's message digest.
3. When replicas receive a PRE-PREPARE message, they validate it by checking if it has the expected view and sequence number. If valid, a PREPARE message is multicasted to all replicas. PREPARE messages include the current view number, the sequence number(formerly assigned by the primary), a digest of the client's request, and the replica's id.
4. When a replica receives at least $2f$ valid and matching PREPARE messages (i.e., with the same view, sequence number, and digest from different replicas), they multicast a COMMIT message to all replicas. COMMIT messages consist of the view and sequence number from the respective quorum of PREPARE messages and the replica's id.
5. When a replica receives $2f + 1$ valid and matching COMMIT (i.e., with the same view and sequence number, from different replicas) it can then execute the respective operation in the order defined by the view and sequence number (used for totally ordering REQUESTS), and send a REPLY message to the client. REPLY messages include the view number, the request's timestamp, the client's and replica's identifier, and the operation's result.

Clients that receive at least $2f + 1$ matching REPLY messages from different replicas have the guarantee that the operation executed successfully.

3.2 Faulty behaviour

While PBFT has additional sub-protocols (including view-change, checkpoint, and recovery), their descriptions are outside of the article's scope. However, for the agreement protocol, it is possible to identify several ways in which a faulty node can behave, or how a malicious actor could try to subvert the protocol, by compromising one of the system's properties, namely: Termination, Agreement, Validity or Integrity.

Besides the typical communication-related faults (such as delaying, dropping, or corrupting messages), faulty nodes may send different messages to each replica, or a subset of replicas, to try and prevent termination from occurring. These include REQUEST messages with different operations and/or timestamps to each replica; PRE-PREPARE and PREPARE messages with different view and sequence numbers to each replica. All these faults prevent the agreement protocol from terminating. Note that these faults are in no way exhaustive. However, these allow to identify requirements for our fault injection tool, specifically for validation and evaluation purposes.

4 ZERMIA Design

ZERMIA is a fault injector designed for concurrent and distributed applications, with flexibility, extensibility, and efficiency in mind. It is based on a client-server model, where an independent *Coordinator* manages one or more *Agents* that run alongside the target application, as presented in Figure 2.

The *Coordinator* is responsible for: managing Agents; distribute fault schedules among them; synchronise and coordinate Agents and faults; gather and display metrics and relevant information, during and at the end of each test run.

An *Agent* runs alongside the target application by sharing that process's runtime. When starting the target application, Agents execute their bootstrap, where they register with the Coordinator and request their respective fault scheduler. Then, they advance alongside the application triggering faults according to their fault schedule, gathering and sharing metrics with the Coordinator.

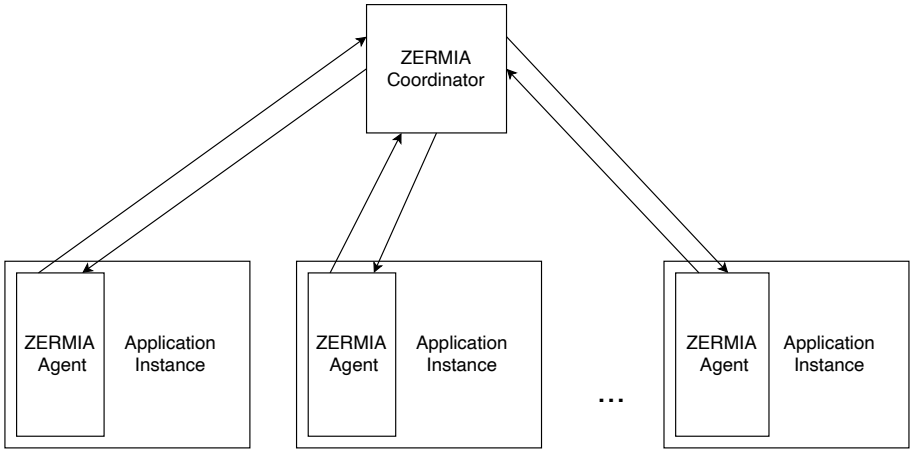


Fig. 2: Overview of the ZERMIA architecture

Agents attach to specific points of the target program, as defined in the fault schedule. They can alter the target's execution state and flow, by accessing method calls, arguments, return values and application state. Agents are also capable of maintaining additional state, independent of the application, such as the number of iterations of the application's main loop, or, in our case, the number of consensus rounds. This increases flexibility since ZERMIA can use this additional information for triggering faults.

4.1 Faults

In ZERMIA, faults are characterised by the following information: *what* is the fault (i.e., the code to be injected); *where* to inject the fault (i.e., the target function or method); *when* to inject the fault (i.e., the condition required to trigger the fault); and, *how* to trigger the fault (i.e., if the fault is to be triggered before, after, and if it prevents the target from executing). In addition, each fault may have additional parameters that allow users to customise or adjust their behaviour.

Since ZERMIA is designed for distributed applications in mind, it provides users with a set of predefined faults that can be parameterised and scheduled without modifications. These include communications and network-related faults (like message delay, message drop, message modifications, etc.) that are typical in these contexts. ZERMIA also defines a generic *Fault Interface* that offers users the means to design and build custom faults for testing their target applications.

4.2 Schedules

Schedules are a sequence of zero or more faults that will be injected into an application during a test run. These include information about the Agent that will apply the schedule, the set of faults that are to be injected by the Agent and the corresponding execution model.

ZERMIA provides scheduling flexibility, offering support for:

- independent fault schedules for each Agent, including fault free schedules (i.e., schedules with zero faults);
- fault dependencies within the same schedule, i.e., dependencies between faults triggered by a single Agent. For instance, prevent a fault from triggering if a previous (possibly different) fault has already been triggered;
- fault dependencies between schedules, i.e., dependencies between fault triggered by different Agents. For instance, only inject a fault after some other Agent has injected its own;
- a priority-based system, that orders faults according to their priority, for resolving possible fault scheduling conflicts deterministically. For example, when two or more faults can be triggered at the same instant.

Both the fault dependency and priority based mechanisms are designed for ZERMIA to deal with possible non-deterministic behaviour of concurrent and distributed applications.

Consider, for example, fault $f1$ that is scheduled based on a time instant (e.g., after application runs for 60 seconds), and fault $f2$ that is scheduled based on a message receive event (e.g., when receiving a

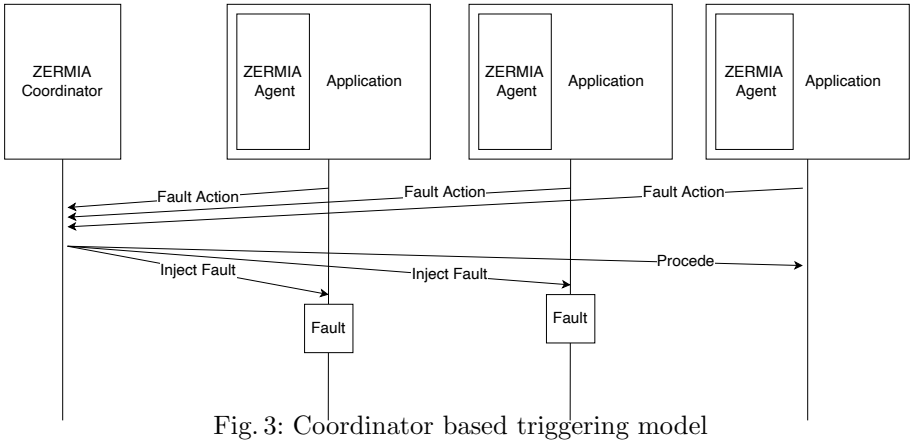


Fig. 3: Coordinator based triggering model

message from another node). Depending on a node's performance or network latency, in some test runs, fault $f1$ may be scheduled before fault $f2$, or $f2$ may be scheduled before $f1$, or even in some cases, both faults can be scheduled at the same instant. Both mechanisms can be used to ensuring a deterministic behaviour (e.g., $f1$ is always triggered after $f2$ or vice-versa), and to allow testing applications with greater reliability, by testing all possible fault interleavings.

4.3 Triggers

While several frameworks and tools provide mechanisms for defining faults based on *What*, *Where* and *When* criteria [32], we consider these criteria to be insufficient for testing concurrent and distributed applications.

Typically, *What* allows users to specify the fault to be injected (i.e., the code of the fault), *Where* allows users to specify the location in which the fault is to be injected (i.e., method signature, memory address, et cetera.), and *When* allows users to specify an instance for injecting the fault (i.e., time or event dependency).

However, concurrent and distributed applications require faults to be triggered by specific threads, processes, or nodes. Additionally, the replicated nature of BFT protocols introduces additional problems, such as how to differentiate between primary and secondary replicas, since their codebase is the same.

For these reasons, ZERMIA uses a hybrid trigger approach, where faults are created and added to the target application's code at compile time and triggered *dynamically* at runtime.

For added flexibility, ZERMIA includes different trigger models, including: *i*) managed by the Coordinator; *ii*) managed by the Agent; and, *iii*) a hybrid approach managed by both the Agent and the Coordinator [2].

Furthermore, fault triggers can be state or event-based, i.e., dependant of target application's state or dependant on some event. For example, a fault can be triggered after a predefined number of consensus rounds, or whenever a message is received (independently of the message type, sender, contents, etc.).

Coordinator-Based Triggering Model In the Coordinator-based approach, the Coordinator is responsible for managing and triggering faults. Agents query the Coordinator before each fault to determine if the fault is to be triggered or not, as presented in Figure 3.

This allows the Coordinator to have a global view of the application execution history and state, and offers the possibility of synchronising and coordinating Agents before triggering faults.

However, this synchronization may compromise testing, since it influences the target application's execution flow due to the added synchronization.

Agent-Based Triggering Model The agent-based approach does not require the Agent to periodically contact the Coordinator, leaving the Agent responsible for triggering faults. This technique reduces overhead, compared with the previous model, as well as prevents the synchronization from compromising

[2] Note that Agents do not exchange information directly with each other, all synchronisation and coordination is managed by the Coordinator.

the target application's execution flows, since no communications are needed between Agents and the Coordinator

However, this model requires Agents to maintain the application execution history and/or state, for validating if fault triggering conditions are satisfied, resulting in an increase of complexity for the Agent implementation. It also prevents external fault dependencies (i.e., fault dependencies between schedules) from triggering, since no synchronisation is performed with the Coordinator.

Hybrid Triggering Model A Hybrid triggering model is also available, where Agents can trigger faults autonomously or synchronise with the Coordinator for triggering faults with external dependencies.

Like in the Agent-based model, this increases complexity in the Agent since it is responsible for managing target application history and/or state. However, it reduces overhead, compared with the Coordinator-based model, and increases flexibility, compared with the Agent-based model, since fault may have external dependencies and only these faults require synchronisation with the Coordinator.

Reducing the overhead is important, since an intrusive fault injector can compromise the target application's execution flow, which in turn compromises test run validity. ZERMIA tries to reach a balance between overhead and flexibility while being able to simulate a wide range of execution scenarios possible.

4.4 Dealing with non-determinism

Concurrent and distributed applications typically suffer from non-deterministic execution traces due to different thread/process interleaving. This is known to result in concurrency errors, which are difficult to detect and reproduce [17, 16, 44, 26].

In order to reduce non-determinism and increase test coverage, ZERMIA provides different mechanisms, including the Coordinator-based and Hybrid triggering models, and the priority-based fault system.

Both triggering models reduce non-determinism and allows testing different fault interleavings by synchronising Agents with the Coordinator before triggering faults. This allows users to test all possible fault interleaving for their schedules based on these dependencies.

The fault priority system, that allows users to assign priorities to faults, can be used to order faults, when two or more faults can be triggered concurrently (i.e., at the same instant) [3]. Changing fault priorities allows users to test different fault orders.

4.5 Discussion

The client-server model of ZERMIA decouples the coordination and management from the Agents processes. This contributes to the extensibility of ZERMIA since Agents can be implemented in different programming languages without requiring any modifications to the Coordinator component. Furthermore, our fault model allows users to design and develop custom faults by implementing the Fault interface, as discussed in Section 5.

The fault and schedule models provided by ZERMIA offer a flexible platform for testing different kinds of applications, independently of the concurrency model. For instance, in a non-concurrent application, ZERMIA can be used with a single Agent that is responsible for managing fault injection, while in a concurrent application, each process or thread can have its own Agent. This allows faults to be triggered independently on a per-process/thread basis, or to coordinate faults between processed/threads. The same principles apply to distributed and replicated applications, where each node/replica can have its own Agent.

Figure 4 represents an example of a fault schedule where Agent *1* executes a Delay fault (of 100 ms) between consensus rounds 5000 and 10000, before method *ServersCommunicationLayer.send*, and a Crash fault at round 20000 before method *ServersConnection.sendBytes*. Agent *2* executes a Crash fault before method *ServersConnection.sendBytes* only after Agent 1 has "crashed".

[3] Note that this can occur when fault triggering conditions are based on different factors, e.g., after a number of rounds and after receiving a specific message.

```
[
   Agent {
      id: 1,
      faults: [
         DelayFault_0 {
            what: zermia.fault.predefined.DelayFault,
            params: {
               duration: 100,
               unit: ms,
            },
            when: {
               start: 5000,
               end: 10000,
               unit: round,
            },
            where: bftsmart.communication.
              server.ServersCommunicationLayer.
                send(..),
            how: before,
         },
         CrashFault_0 {
            fault: zermia.fault.predefined.Crash,
            when: {
               start: 20000,
               unit: round,
            },
            where: bftsmart.communication.
              server.ServersConnection.
               sendBytes(..)
            how: before,
         },
      ]
   },
   Agent {
      id: 2,
      faults: [
         CrashFault_1 {
            fault: zermia.fault.predefined.Crash,
            when: {
               after: CrashFault_0,
            },
            where: bftsmart.communitation.
              server.ServersConnection.
                sendBytes(..)
            how: before,
         },
      ]
   }
]
```

Fig. 4: Example of a fault schedule

5 Implementation details

ZERMIA is currently implemented in Java and uses AspectJ (an aspect-oriented programming extension) for integrating Agents with the target application. Communications between Coordinator and Agents use gRPC [18], decoupling both components and allowing users to extend the framework with support for additional programming languages without having to modify the Coordinator.

5.1 Coordinator

The Coordinator is responsible for managing and coordinating Agents and their respective fault schedulers. During startup, the fault schedule, defined by the users, is read from a file. Then the Coordinator initializes the gRPC communication channels and waits for Agents to connect.

gRPC Interface Agents interact with the Coordinator using the gRPC interface presented in Figure 5. This interface allows Agents to register, request their respective fault schedulers, validate fault triggering conditions, notify about fault execution, and update test metrics.

```
service ZermiaCoordinatorServices{
   rpc RegisterAgent {..}
   rpc RequestScheduller {..}
   rpc FaultTriggerValidation {..}
   rpc NotifyFaultExecution {..}
   rpc UpdateMetrics {..}
}
```

Fig. 5: Coordinator service interface (gRPC methods)

5.2 Agents

Agents are responsible for registering with the Coordinator and injecting faults into the target application. They integrate with the target application through the use of *advices*. *Advices* are blocks of code that are inserted into the target application's code at compile-time by the use of *join-points*. *Join-points* are the locations on the target application's code where the *advices* are inserted. These have an associated semantics for specifying if an advice executes before, after or around the respective join-point, referred to as *pointcuts*. The around semantics can also be used to prevent executing the corresponding join-point (i.e., target method). In our current implementation, an Agent is instantiated when a replica starts by using an advice associated with the replica's *main* function.

During bootstrap, the Agent registers with the Coordinator, requests its fault schedule (using the gRPC services presented in Figure 5), and initialises the necessary data structures for maintaining its state. This includes the value of the current consensus round and the fault schedule (represented as a list of Fault objects). Figure 6 presents a simplified view of the Agent code.

During the application execution, whenever a join-point is called, the execution context is passed to the Agent that verifies if any fault in its schedule is to execute at that join-point, and if the fault's trigger conditions are met (using the *canTrigger* method). If both are true, then the fault is executed according to its semantics.

```
public class ZermiaAgent {
    // state variables
    List<Fault> schedule;
    int currentConsensusRound;
    (...)
    @Around ("execution method_signature1")
    public void execute(JointPoint jp) {
        for f in schedule {
            if ( f.canTrigger(jp, currentConsensusRound) ) {
                if (f.how() == Fault.AFTER)
                    jp.proceed();
                f.triggerFault();
                if (f.how() == Fault.INSTEAD)
                    return;
                else //Fault.BEFORE
                    jp.proceed();
            }
        }
    }
    @After ("execution main_app_loop")
    public void new_iteration() {
        currentConsensusRound += 1;
    }
    (...)
}
```

Fig. 6: Agent implementation example (JAVA)

Additional join-points Agents may use additional join-points for managing their state. In our case, we identified the application's main loop to be associated with the consensus round. When a new PRE-PREPARE message is received, a new consensus round starts, as presented in Figure 1. When the join-point is called, the Agent updates its current consensus round (Figure 6). This allows an Agent to decide to trigger faults based on this information.

5.3 Faults

Rather than being mapped as advices and inserted into joint-points at compile-time, faults are instantiated during an Agent bootstrap and base on the description received from the Coordinator. Each fault in the schedule is instantiated using the corresponding class loader, configured according to its parameters, and added to the respective schedule.

When an Agent calls the *canTrigger* method, it validates if its join-point is the same as the one passed by the Agent and if the associated conditions are met. Figure 7 presents an example of a delay fault that delays the execution of the respective joint-point for some period of time.Its preconditions are only related to the joint-point signature, and the value of consensus round passed as argument. If all conditions are true, the fault can execute, being bypassed otherwise.

```
public class DelayFault implements Fault {
    //parameter variables
    String methodName;
    int start;
    int end;
    int duration;
    int how;
    (...)
    public boolean canTrigger(JointPoint jp, int currentRound) {
        if (jp.getName().equals(methodName))
            return this.start <= currentRound && currentRound <= end;
        return false;
    }
    public FaultResult triggerFault() {
        try {
            Thread.currentThread().sleep(duration);
        } catch (...) {..}
    }
}
```

Fig. 7: Delay Fault implementation example (JAVA)

Fault Interface For custom building fault, ZERMIA provides users with a generic Fault interface, as presented in Figure 8. This interface contains three methods: *canTrigger*, *triggerFault*, and *how*. The first two methods are used by the Agent to validate the preconditions of the fault (i.e., if it can be triggered) and to execute the fault if the preconditions are satisfied, while the last is used to determine the fault's semantics (i.e., fault's pointcuts)

```
public interface Fault {
    public boolean canTrigger(JointPoint jp);
    public FaultResult triggerFault();
    public int how();
}
```

Fig. 8: Generic Fault interface (JAVA)

5.4 Predefined Faults and Extensibility

In its current implementation, ZERMIA includes a set of common faults for testing distributed systems. Although these are generic faults, they can be configured according to the user's requirements. These include:

- **Delay Thread** - Delays execution flow through the use of *Thread.sleep()* method. The user can configure the delay in milliseconds.
- **Message Dropping** - Prevents messages from being sent by preventing the target method call.
- **Modify Message** - Replaces messages contents by modifying argument values passed to the target join-point. Message contents can be defined by the user and may be empty.
- **Message Flood** - Sends a variable number of messages by enclosing the target method in a loop. The user can define the number of loop iterations (i.e., messages to send).
- **Crash** - Crashes the application by calling the *System.exit()* method.

6 Evaluation

In this section we present the evaluation of ZERMIA by showing its flexibility and efficiency. To this end, we designed different fault schedules that target BFT-SMaRt[3], a Byzantine Fault tolerant library based on PBFT's agreement protocol, and applying these schedules when that library is used by an independent application, and under different configurations.

We used YCSB [9] as the application that used the BFT-SMaRt library, as it presents metrics that allows us to view the influence in performance each schedule has on the target library.

6.1 Experimental Setup

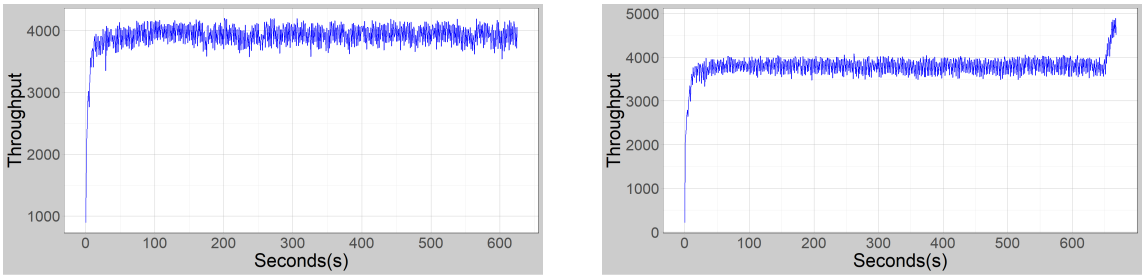
Each experiment consisted in running YCSB with 50 clients, for a total of 2.5 million operations (approximately 350000 consensus rounds) and measuring the system throghput (in operations per second).

Fault triggering started in consensus round number 50000 for $f = 1$, and rounds 100000 and 150000 for $f = 3$. Period of fault injection lasted for a variable number of rounds (depending on the type of fault). BFT-SMaRt was configured with $f = 1$ and $f = 3$, $N = 4$ and $N = 10$ replicas respectively. Specific fault schedules where designed for each leader and non-leader replicas depending on the experiment, always taking into consideration the protocol's invariant of $f \leq \frac{N-1}{3}$.

A total of twelve virtual machines from Google Cloud services where used: 3-10 for the BFT-SMaRt replicas, 1 for the YCSB clients, and 1 for the Zermia Coordinator. Each machine ran Ubuntu 18.04.5 LTS operative system, with 16GB of RAM, 4-core AMD EPYC 7B12 clocked at 2.25GHz, and 10GB of HDD. The presented results are the average from the results of 5 consecutive runs.

6.2 Baseline results

Figures 9a and 9b present the results obtained from fault-free schedules, for $f = 1$ and $f = 3$ respectively. Similar throughput is achieved despite the increase in the number of replicas. These serve as a baseline for comparison when injecting faults.



(a) $f = 1$ (b) $f = 3$

Fig. 9: Results from a fault-free schedule

6.3 Delaying replicas

This experiment consisted in executing a schedule that delayed replicas before sending messages (independent of message type). Faults were triggered for 10000 consecutive consensus rounds, starting at consensus round 50000, 100000, and 150000, with delay intervals of 20 milliseconds, 50 milliseconds, and 100 milliseconds respectively.

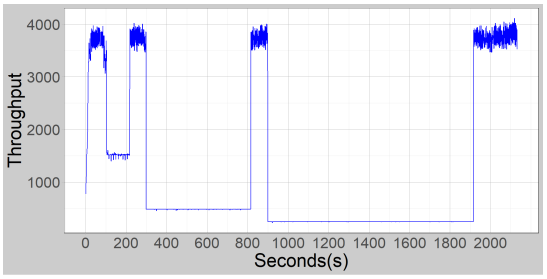
Figure 10 presents the results when targeting different replicas with this schedule: the leader (Figure 10a), the leader and 2 non-leader replicas (Figure 10b), and 3 non-leader replicas (Figure 10c).

It is possible to observe a degradation in performance whenever a delay fault targets the primary replica. Additionally, collusion between a primary and secondary replicas yields a similar outcome (Figure 10b), as system performance is directly related to the rate of *PRE-PROPOSE*. Since these are only transmitted by the primary replicas, delaying secondary replicas does not influence the system performance since the remaining non-faulty replicas are sufficient to guarantee *liveness*. This is corroborated by the results obtained when targeting only secondary replicas (Figure 10c).

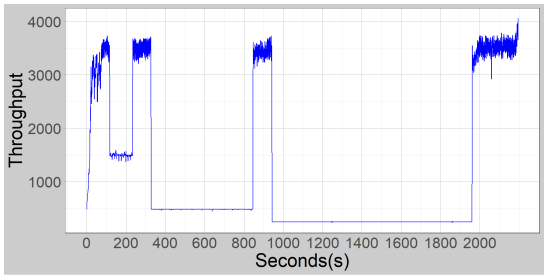
6.4 Crashing replicas

This experiment consisted in executing a schedule that crashes replicas, triggered at consensus round 50000 for $f = 1$, and 100000 and 150000 for $f = 3$. The schedules targeted both leader and non-leader replicas.

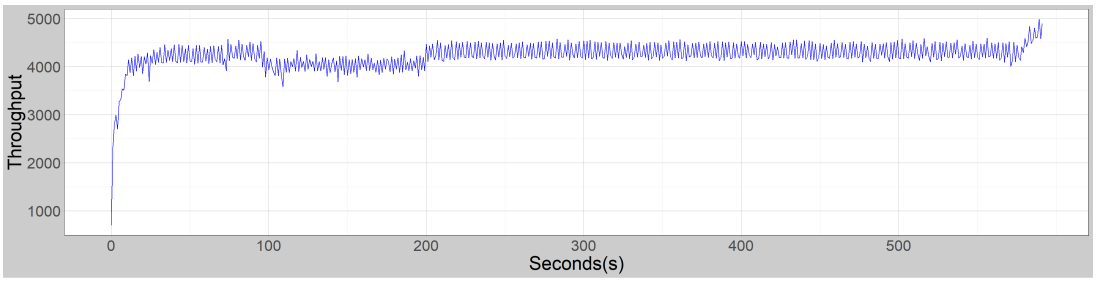
Figure 11 presents the results when targeting only leader replicas for $f = 1$ (Figure 11a), $f = 3$ (Figure 11b, and non-leader replicas (Figure 11c). These schedules result in a performance drop when the fault is injected, i.e., when the leader fails. This is consequence of the protocol adjusting to the crashing replica by electing a new primary. Additionally, it is possible to see a drop in system performance when the system reaches the minimum number of replicas needed to guarantee *safety*.



(a) Target leader with $f = 1$

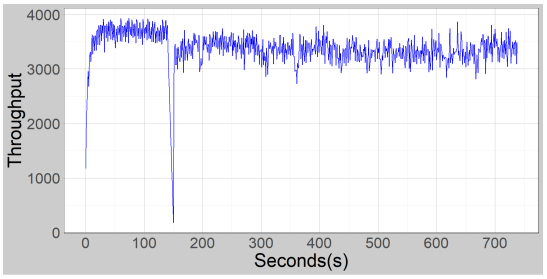


(b) Target leader and 2 non-leader replicas with $f = 3$

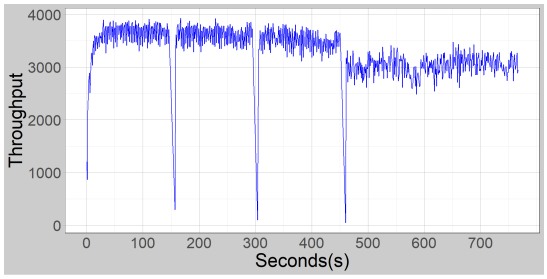


(c) Target 3 non-leader replicas with $f = 3$

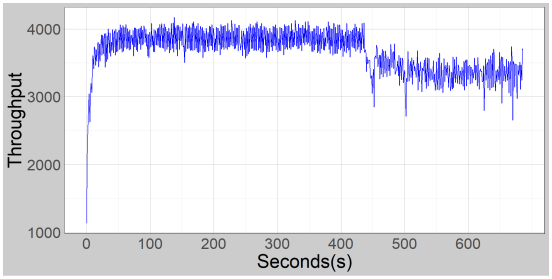
Fig. 10: Results from a the delay fault schedule



(a) Target leader with $f = 1$



(b) Target leader with $f = 3$



(c) Target non-leaders with $f = 3$

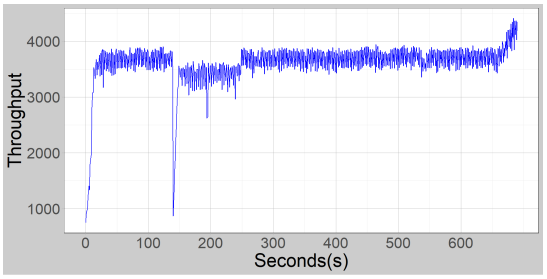
Fig. 11: Results for a crash fault schedule

6.5 Message dropping

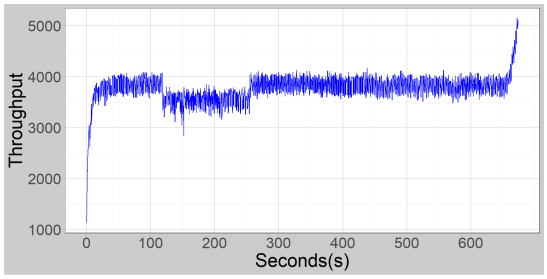
This experiment consisted in executing a schedule that drops messages, by bypassing message sending methods, triggering once replicas reached the consensus round 50000, for a duration of 50000 rounds.

Figure 12 presents the obtained results when targeting: the leader with $f = 1$ (Figure 12a), a non-leader with $f = 1$ (Figure 12b), and when targeting non-leader replicas with $f = 3$ (Figure 12c).

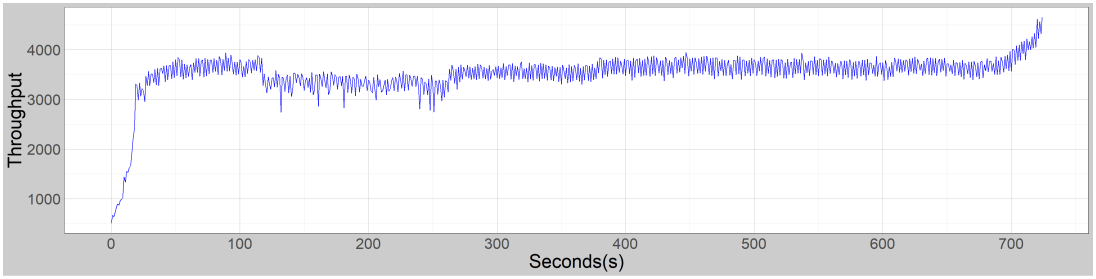
These schedules result in a reduction of application performance during the fault period. This reduction in performance is similar to previous results when the number of non-faulty replicas reaches the minimum required for the system to guarantee safety.



(a) Target leader with $f = 1$



(b) Target non-leader with $f = 1$



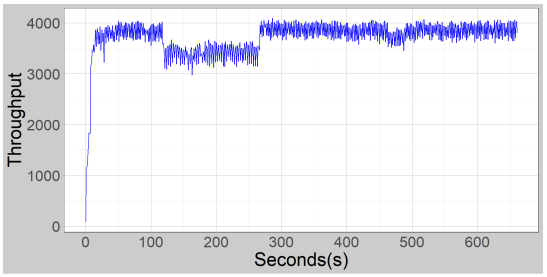
(c) Target non-leaders with $f = 3$

Fig. 12: Results for a message dropper schedule

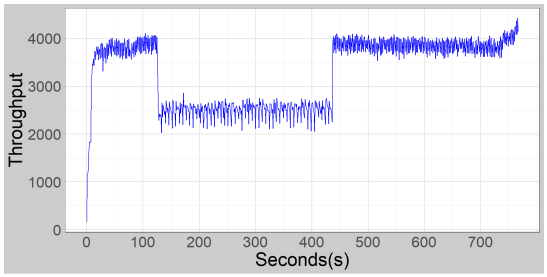
6.6 Message Flooding

This experiment consisted in executing different schedules for flooding the system with messages. Faults were triggered at consensus round 50000, for a period 50000 rounds, unless stated otherwise. The flooding consisted in unicasting or multicasting 5000 additional messages for every legitimate message sent, depending on the schedule.

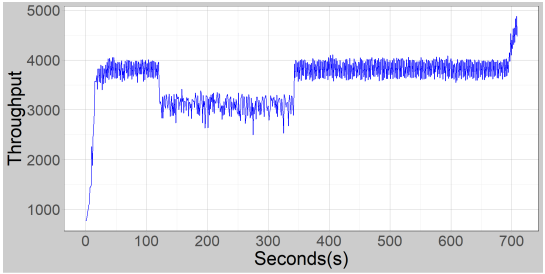
Four different schedules were used, including: a non-leader replica flooding all replicas; a non-leader replica flooding the leader replica; and the leader replica flooding all replicas. For the latter schedule, the period was reduced to 10000 rounds for experiment duration purposes.



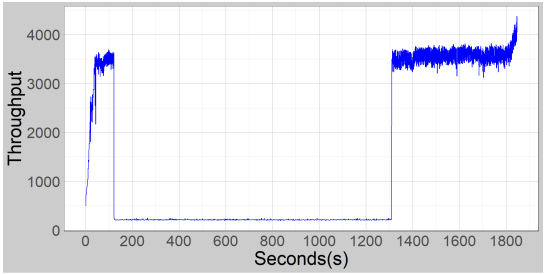
(a) Non-leader replica floods all replica, $f = 1$



(b) Non-leader replicas flood all replica, $f = 3$



(c) Non-leader replicas flood the leader, $f = 3$



(d) Leader replica floods to all replicas, $f = 3$

Fig. 13: Results for a flood schedule

Figure 13 presents the results from this experiment when a non-leader flood all replicas in a $f = 1$ and $f = 3$ configuration (Figures 13a and 13b respectively), a non-leader floods the leader replica (Figure 13c), and when the primary floods the system (Figure 13d).

The presented results show that, increasing the number of replicas flooding the system directly influences the performance impact of the fault, as witnessed by the drop in performance of Figure 13b compared to Figure 13a, which result from the increase in the total number of messages that flood the system. And also, that flooding the entire system has greater impact in system performance than targeting solely the leader, as put in evidence by the reduction in performance when comparing Figures 13c and 13b.

However, the greatest impact in system performance results from the a leader replica flooding the system, as observed in Figure 13d. This results from the influence the leader has in performance due to its responsibilities within the system (proposing the order of operations). Finally, note that the number of flood messages is sufficiently high to impact system performance without triggering a leader election, since some client requests are still answered before their respective timeouts are triggered.

7 Conclusions and Future Work

In this work we presented ZERMIA. A software fault injector designed for testing and validating concurrent and distributed applications, with special focus on flexibility and extensibility. Its design allows users and developers to test their applications, without requiring any modifications to the target application code, and provides users with different fault and trigger models that allow them to test application is realistic scenarios, including monitoring and managing different thread/process interleaving for testing concurrency associated bugs.

We evaluated ZERMIA using a concurrent bench-marking tool, YCSB, and a state of the art Byzantine Fault Tolerance library, BFT-SMaRt. This combination allowed us to present and validate the capabilities of ZERMIA in testing concurrent and distributed applications, and its flexibility by designing different fault schedules that allowed us to run different experiments without any modifications to either the applications and the BFT library. These results also allowed us to show the performance impact communication related faults can have in these systems. We intend to further use ZERMIA to exhaustively test this and other BFT systems, specifically by studying the influence faulty clients may have in these systems.

Additionally, we are working on a generalization of the system so it can be easily ported to other programming languages, and used to test a wider variety of applications. Also, to abstract users from possibly complex Agent implementations, we are working on a Domain Specific Language (DSL) that will allow developers to design and build custom Agents based on fault and schedule descriptions.

References

1. Arlat, J., Aguera, M., Amat, L., Crouzet, Y., Fabre, J.C., Laprie, J.C., Martins, E., Powell, D.: Fault injection for dependability validation: a methodology and some applications. IEEE Transactions on Software Engineering **16**(2), 166–182 (1990). https://doi.org/10.1109/32.44380
2. Aublin, P.L., Mokhtar, S.B., Quéma, V.: Rbft: Redundant byzantine fault tolerance. In: 2013 IEEE 33rd International Conference on Distributed Computing Systems. pp. 297–306 (2013). https://doi.org/10.1109/ICDCS.2013.53
3. Bessani, A., Sousa, J., Alchieri, E.E.: State machine replication for the masses with bft-smart. In: 2014 44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks. pp. 355–362 (2014). https://doi.org/10.1109/DSN.2014.43
4. Carreira, J., Madeira, H., Silva, J.: Xception: a technique for the experimental evaluation of dependability in modern computers. IEEE Transactions on Software Engineering **24**(2), 125–136 (1998). https://doi.org/10.1109/32.666826
5. Castro, M., Liskov, B.: Practical byzantine fault tolerance. In: Proceedings of the Third Symposium on Operating Systems Design and Implementation. p. 173–186. OSDI '99, USENIX Association, USA (1999)
6. Castro, M., Liskov, B.: Practical byzantine fault tolerance and proactive recovery. ACM Trans. Comput. Syst. **20**(4), 398–461 (Nov 2002). https://doi.org/10.1145/571637.571640, https://doi.org/10.1145/571637.571640
7. Chandra, R., Lefever, R., Cukier, M., Sanders, W.: Loki: a state-driven fault injector for distributed systems. In: Proceeding International Conference on Dependable Systems and Networks. DSN 2000. pp. 237–242 (2000). https://doi.org/10.1109/ICDSN.2000.857544

8. Chandra, T.D., Toueg, S.: Unreliable failure detectors for reliable distributed systems. J. ACM **43**(2), 225–267 (Mar 1996). https://doi.org/10.1145/226643.226647, https://doi.org/10.1145/226643.226647
9. Cooper, B.F., Silberstein, A., Tam, E., Ramakrishnan, R., Sears, R.: Benchmarking cloud serving systems with ycsb. In: Proceedings of the 1st ACM Symposium on Cloud Computing. p. 143–154. SoCC '10, Association for Computing Machinery, New York, NY, USA (2010). https://doi.org/10.1145/1807128.1807152, https://doi.org/10.1145/1807128.1807152
10. Correia, M., Veronese, G.S., Neves, N.F., Verissimo, P.: Byzantine consensus in asynchronous message-passing systems: A survey. Int. J. Crit. Comput.-Based Syst. **2**(2), 141–161 (Jul 2011)
11. Cotroneo, D., De Simone, L., Liguori, P., Natella, R., Bidokhti, N.: How bad can a bug get? an empirical analysis of software failures in the openstack cloud computing platform. In: Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering. p. 200–211. ESEC/FSE 2019, Association for Computing Machinery, New York, NY, USA (2019). https://doi.org/10.1145/3338906.3338916, https://doi.org/10.1145/3338906.3338916
12. Duraes, J.A., Madeira, H.S.: Emulation of software faults: A field data study and a practical approach. IEEE Transactions on Software Engineering **32**(11), 849–867 (2006). https://doi.org/10.1109/TSE.2006.113
13. Dwork, C., Lynch, N., Stockmeyer, L.: Consensus in the presence of partial synchrony. J. ACM **35**(2), 288–323 (Apr 1988). https://doi.org/10.1145/42282.42283, https://doi.org/10.1145/42282.42283
14. Fabre, J.C., Salles, F., Moreno, M., Arlat, J.: Assessment of cots microkernels by fault injection. In: Dependable Computing for Critical Applications 7. pp. 25–44 (1999). https://doi.org/10.1109/DCFTS.1999.814288
15. Fischer, M.J., Lynch, N.A., Paterson, M.S.: Impossibility of distributed consensus with one faulty process. J. ACM **32**(2), 374–382 (Apr 1985). https://doi.org/10.1145/3149.214121, https://doi.org/10.1145/3149.214121
16. Fonseca, P., Li, C., Rodrigues, R.: Finding complex concurrency bugs in large multi-threaded applications. In: Proceedings of the Sixth Conference on Computer Systems. p. 215–228. EuroSys '11, Association for Computing Machinery, New York, NY, USA (2011). https://doi.org/10.1145/1966445.1966465, https://doi.org/10.1145/1966445.1966465
17. Fonseca, P., Li, C., Singhal, V., Rodrigues, R.: A study of the internal and external effects of concurrency bugs. In: 2010 IEEE/IFIP International Conference on Dependable Systems Networks (DSN). pp. 221–230 (2010). https://doi.org/10.1109/DSN.2010.5544315
18. Google: gRPC - A High-Performance Open-Source Universal RPC Framework (2015), http://www.grpc.io/, Accessed: 2021-07-01
19. Gunawi, H.S., Do, T., Joshi, P., Alvaro, P., Hellerstein, J.M., Arpaci-Dusseau, A.C., Arpaci-Dusseau, R.H., Sen, K., Borthakur, D.: Fate and destini: A framework for cloud recovery testing. In: Proceedings of the 8th USENIX Conference on Networked Systems Design and Implementation. p. 238–252. NSDI'11, USENIX Association, USA (2011)
20. Han, S., Shin, K., Rosenberg, H.: Doctor: an integrated software fault injection environment for distributed real-time systems. In: Proceedings of 1995 IEEE International Computer Performance and Dependability Symposium. pp. 204–213 (1995). https://doi.org/10.1109/IPDS.1995.395831
21. Hiller, M., Jhumka, A., Suri, N.: Propane: An environment for examining the propagation of errors in software. In: Proceedings of the 2002 ACM SIGSOFT International Symposium on Software Testing and Analysis. p. 81–85. ISSTA '02, Association for Computing Machinery, New York, NY, USA (2002). https://doi.org/10.1145/566172.566184, https://doi.org/10.1145/566172.566184
22. Hsueh, M.C., Tsai, T., Iyer, R.: Fault injection techniques and tools. Computer **30**(4), 75–82 (1997). https://doi.org/10.1109/2.585157
23. Jin, A., Jiang, J., Hu, J., Lou, J.: A pin-based dynamic software fault injection system. In: 2008 The 9th International Conference for Young Computer Scientists. pp. 2160–2167 (2008). https://doi.org/10.1109/ICYCS.2008.329
24. Kanawati, G., Kanawati, N., Abraham, J.: Ferrari: a flexible software-based fault and error injection system. IEEE Transactions on Computers **44**(2), 248–260 (1995). https://doi.org/10.1109/12.364536
25. Lamport, L.: Time, clocks, and the ordering of events in a distributed system. Commun. ACM **21**(7), 558–565 (Jul 1978). https://doi.org/10.1145/359545.359563, https://doi.org/10.1145/359545.359563
26. Li, G., Lu, S., Musuvathi, M., Nath, S., Padhye, R.: Efficient scalable thread-safety-violation detection: Finding thousands of concurrency bugs during testing. In: Proceedings of the 27th ACM Symposium on Operating Systems Principles. p. 162–180. SOSP '19, Association for Computing Machinery, New York, NY, USA (2019). https://doi.org/10.1145/3341301.3359638, https://doi.org/10.1145/3341301.3359638
27. Liu, S., Viotti, P., Cachin, C., Quéma, V., Vukolic, M.: Xft: Practical fault tolerance beyond crashes. In: Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation. p. 485–500. OSDI'16, USENIX Association, USA (2016)
28. Lu, Q., Farahani, M., Wei, J., Thomas, A., Pattabiraman, K.: Llfi: An intermediate code-level fault injection tool for hardware faults. In: 2015 IEEE International Conference on Software Quality, Reliability and Security. pp. 11–16 (2015). https://doi.org/10.1109/QRS.2015.13

29. Martins, E., Rubira, C., Leme, N.: Jaca: a reflective fault injection tool based on patterns. In: Proceedings International Conference on Dependable Systems and Networks. pp. 483–487 (2002). https://doi.org/10.1109/DSN.2002.1028934
30. Martins, M., Rosa, A.: A fault injection approach based on reflective programming. In: Proceeding International Conference on Dependable Systems and Networks. DSN 2000. pp. 407–416 (2000). https://doi.org/10.1109/ICDSN.2000.857569
31. Martins, R., Gandhi, R., Narasimhan, P., Pertet, S., Casimiro, A., Kreutz, D., Veríssimo, P.: Experiences with fault-injection in a byzantine fault-tolerant protocol. In: Eyers, D., Schwan, K. (eds.) Middleware 2013. pp. 41–61. Springer Berlin Heidelberg, Berlin, Heidelberg (2013)
32. Natella, R., Cotroneo, D., Madeira, H.S.: Assessing dependability with software fault injection: A survey. ACM Comput. Surv. **48**(3) (Feb 2016). https://doi.org/10.1145/2841425, https://doi.org/10.1145/2841425
33. Platania, M., Obenshain, D., Tantillo, T., Amir, Y., Suri, N.: On choosing server- or client-side solutions for bft. ACM Comput. Surv. **48**(4) (Mar 2016). https://doi.org/10.1145/2886780, https://doi.org/10.1145/2886780
34. Rivest, R.L., Shamir, A., Adleman, L.: A method for obtaining digital signatures and public-key cryptosystems. Commun. ACM **21**(2), 120–126 (Feb 1978). https://doi.org/10.1145/359340.359342, https://doi.org/10.1145/359340.359342
35. Rosenberg, H., Shin, K.: Software fault injection and its application in distributed systems. In: FTCS-23 The Twenty-Third International Symposium on Fault-Tolerant Computing. pp. 208–217 (1993). https://doi.org/10.1109/FTCS.1993.627324
36. Sanches, B.P., Basso, T., Moraes, R.: J-swfit: A java software fault injection tool. In: 2011 5th Latin-American Symposium on Dependable Computing. pp. 106–115 (2011). https://doi.org/10.1109/LADC.2011.20
37. Sastry Hari, S.K., Adve, S.V., Naeimi, H., Ramachandran, P.: Relyzer: Application resiliency analyzer for transient faults. IEEE Micro **33**(3), 58–66 (2013). https://doi.org/10.1109/MM.2013.30
38. Schneider, F.B.: Implementing fault-tolerant services using the state machine approach: A tutorial. ACM Comput. Surv. **22**(4), 299–319 (Dec 1990). https://doi.org/10.1145/98163.98167, https://doi.org/10.1145/98163.98167
39. Segall, Z., Vrsalovic, D., Siewiorek, D., Ysskin, D., Kownacki, J., Barton, J., Dancey, R., Robinson, A., Lin, T.: Fiat - fault injection based automated testing environment. In: Twenty-Fifth International Symposium on Fault-Tolerant Computing, 1995, ' Highlights from Twenty-Five Years'. pp. 394– (1995). https://doi.org/10.1109/FTCSH.1995.532663
40. Sousa, J., Bessani, A., Vukolic, M.: A byzantine fault-tolerant ordering service for the hyperledger fabric blockchain platform. In: 2018 48th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN). pp. 51–58 (2018). https://doi.org/10.1109/DSN.2018.00018
41. Svenningsson, R., Vinter, J., Eriksson, H., Törngren, M.: Modifi: A model-implemented fault injection tool. In: Schoitsch, E. (ed.) Computer Safety, Reliability, and Security. pp. 210–222. Springer Berlin Heidelberg, Berlin, Heidelberg (2010)
42. Tsai, T.K., Iyer, R.K.: Measuring fault tolerance with the ftape fault injection tool. In: Beilner, H., Bause, F. (eds.) Quantitative Evaluation of Computing and Communication Systems. pp. 26–40. Springer Berlin Heidelberg, Berlin, Heidelberg (1995)
43. Tsudik, G.: Message authentication with one-way hash functions. SIGCOMM Comput. Commun. Rev. **22**(5), 29–38 (Oct 1992). https://doi.org/10.1145/141809.141812, https://doi.org/10.1145/141809.141812
44. Wang, J., Dou, W., Gao, Y., Gao, C., Qin, F., Yin, K., Wei, J.: A comprehensive study on real world concurrency bugs in node.js. In: 2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE). pp. 520–531 (2017). https://doi.org/10.1109/ASE.2017.8115663
45. Yin, M., Malkhi, D., Reiter, M.K., Gueta, G.G., Abraham, I.: Hotstuff: Bft consensus with linearity and responsiveness. In: Proceedings of the 2019 ACM Symposium on Principles of Distributed Computing. p. 347–356. PODC '19, Association for Computing Machinery, New York, NY, USA (2019). https://doi.org/10.1145/3293611.3331591, https://doi.org/10.1145/3293611.3331591