Experiences with Fault-Injection in a Byzantine Fault-Tolerant Protocol

Rolando Martins[1], Rajeev Gandhi[1], Priya Narasimhan[1], Soila Pertet[1], António Casimiro[2], Diego Kreutz[2], and Paulo Veríssimo[2]

[1] Department of Electrical & Computer Engineering, Carnegie Mellon University
rolandomartins@cmu.edu, rgandhi@ece.cmu.edu, priya@cs.cmu.edu, spertet@ece.cmu.edu

[2] Departamento de Informática, Universidade de Lisboa, Faculdade de Ciências
casim@di.fc.ul.pt, kreutz@lasige.di.fc.ul.pt, pjv@di.fc.ul.pt

Abstract. The overall performance improvement in Byzantine fault-tolerant state machine replication algorithms has made them a viable option for critical high-performance systems. However, the construction of the proofs necessary to support these algorithms are complex and often make assumptions that may or may not be true in a particular implementation. Furthermore, the transition from theory to practice is difficult and can lead to the introduction of subtle bugs that may break the assumptions that support these algorithms. To address these issues we have developed Hermes, a fault-injector framework that provides an infrastructure for injecting faults in a Byzantine fault-tolerant state machine. Our main goal with Hermes is to help practitioners in the complex process of debugging their implementations of these algorithms, and at the same time increase the confidence of possible adopters, e.g., systems researchers, industry, by allowing them to test the implementations. In this paper, we discuss our experiences with Hermes to inject faults in BFT-SMaRt, a high-performance Byzantine fault-tolerant state machine replication library.

Keywords: Byzantine fault-injector, failure diagnosis, cloud-computing, Byzantine fault-tolerance, intrusion-tolerance.

1 Introduction

Recent improvements in the performance of Byzantine Fault-Tolerant (BFT) protocols have made such protocols feasible for building fault- and intrusion-tolerant systems. Presently, there are multiple implementations of BFT protocols at disposal of system developers to make their own system fault/intrusion-tolerant without worrying about having to implement the functionality by themselves. However, as other research projects [1] have observed, while current state-of-the-art BFT protocol implementations have considerably improved the performance of the fault-free path, they often fail to properly handle all of the corner cases. The end result is that while many BFT implementations can efficiently handle the complexity of Byzantine failures, they often suffer from multiple orders of magnitude reductions in throughput and long periods of unavailability when in the presence of non-independent faults, such as colluding malicious nodes. This often poses a dilemma for system programmers – on the one hand the use of a publicly available implementation of BFT protocol allows programmers to develop fault- and intrusion-tolerant systems without worrying about implementing these complicated aspects themselves while, on the other hand, there is a question about the ability of an implementation to actually handle complex as well as simple failures in an efficient manner.

System developers often need answers to multiple questions about a particular BFT protocol before they are able to select it and be confident that the implementation will actually meet all of their requirements. Questions can vary from performance to robustness and trustworthiness, such as the following examples. What kind of faults does the system tolerates? Does it really tolerates arbitrary faults? Or only more common faults, e.g. crash faults? What is the degradation of system throughput in the presence of faults? Are there thresholds for fault-arrival rates, beyond which the system breaks down? How does a BFT protocol compare with others?

Software fault injection is often used in software testing to quantitatively assess the impact of faults/bugs in the software. We use a similar approach to assess the performance of a BFT protocol and answer questions like the aforementioned ones that developers may have about the behavior of a protocol in the presence of faults. In this paper, we describe Hermes, our fault injection framework created to help BFT protocol developers in the strenuous task of testing the behavior of a BFT implementation under a diverse and broad range variety of faults. To show its usability, we used Hermes to assess the behavior of BFT-SMaRt [2], a well-known BFT protocol implementation.

Hermes allows system developers to get insight into the performance of a BFT protocol implementation by allowing them to inject faults and observe the behavior of the system. Hermes's fault injection architecture is flexible and allows protocol independent faults (like crash faults, network faults) as well as protocol dependent faults (like corrupt headers, forged signatures) to be injected into a BFT protocol. Our approach is clearly distinguished from existing ones by the fact that we provide a way to simultaneously inject faults across multiple nodes, allowing different type of faults to be injected in different nodes. Furthermore, by simply selecting the appropriate set of faults, the user can enforce a specific fault model, e.g., if collusion is outside the fault model, then no collusion faults can be used.

We built Hermes using AspectJ [3], for the JAVA runtime, and AspectC++ [4], for the C++ runtime, which allows it to seamlessly weave the fault-injecting infrastructure into the target protocol. The use of Aspect-Oriented Programming (AOP) was to avoid source-code modifications on the target system, especially in context-free faults, i.e., faults that do not modify or access any internal state of the protocol. We decided not to use dynamic weaving because it would introduce further complexity into the infrastructure. As such, the injection points are statically weaved and compiled into the target source code.

Hermes does not require a system developer to be familiar with the BFT protocol or its implementation for injecting protocol independent faults. On the other hand, protocol dependent faults, such as payload size corruption, require that the developer performs some modifications to the source code. A minimal amount of adaptation is unavoidable because it depends on the specific details of protocol in use. In our experience with BFT-SMaRt, it took us about two hours to inject protocol dependent faults.

2 Related Work

There has been considerable work done in developing fault injection systems [5–8] and analyzing the dependability of fault-tolerant systems [9–11]. Loki [5] 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. DBench Project [6] aimed to develop standards for creating dependability benchmarks for computer systems. This joint cooperation characterized benchmarks as representing an agreement that is widely accepted both by the computer industry and/or by the user community. Orchestra [9] is a fault injector that uses an interception approach similar to ours to inject communication faults into any layer in the protocol stack. The fault injection core provides the ability to filter, manipulate, and inject new messages. Ferrari [12] (Fault and Error Automatic Real-Time Injection) uses software traps to inject CPU, memory, and bus faults. The Fault Tolerance and Performance Evaluator (Ftape) [13] allows developers to inject faults into user mode registers in CPUs, memory locations, and the disk subsystem. Doctor [7] (Integrated Software Fault Injection Environment) allows developers to inject CPU faults, memory faults, and network communication faults in a system. Xception [14] takes advantage of the advanced debugging and performance monitoring features present in many modern processors to inject more realistic faults. Ballista [15] is a “black box” software testing tool that uses combinational tests of valid and invalid parameter values for subroutine calls, methods and functions. A good survey of fault injection techniques and tools for testing software dependability is provided in [16].

Some of the recent research has looked at the inefficiencies of BFT protocol implementations to handle Byzantine as well as benign faults. In [1], the authors provide a comparative in-depth analysis of several protocols, namely [17–20], in their pursuit to build Aardvark. Their assessment is based mainly in the use of flooding and packet delay (in both primary and non-primary nodes). Similarly, in Prime [21], the authors provide an evaluation of PBFT [17], a leader-based Byzantine fault-tolerant replication protocol, but mention that their approach should work well with all BFT protocols that are leader based, such as [19,22–26]. The experiments conducted in Prime were based on two attacks. The first involved delaying ”pre-prepare” messages, while the second consisted in timeout manipulation, where the system would become stalled until large timeouts occurred.

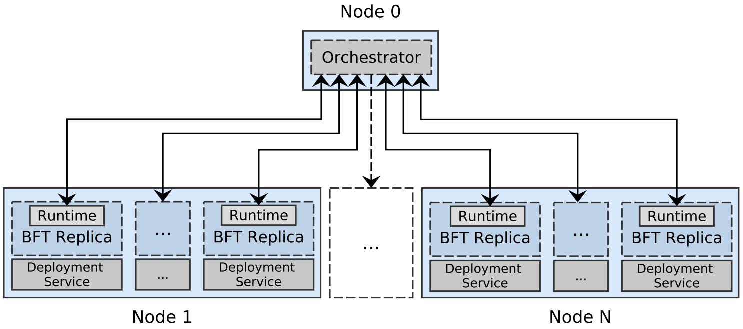

Fig. 1: Overview of the Hermes's architecture.

3 Hermes's Overview

Our fault-injection platform, shown in Figure 1, is governed by a fault-injection orchestrator that enables the injection of multiple faults, simultaneous or not, across multiple remote nodes. The actual fault injection is performed by Hermes's runtime, which is incorporated into the BFT replica (through code-weaving). The initial deployment, i.e., the initial view, of both replicas and client is accomplished through the use of the deployment service. However, a BFT protcol can still reconfigure, e.g., adding a new replica, using its usual facilities without having to interact with the deployment service, because the runtime is able to transparently connect to Hermes's orchestrator.

We start by characterizing the types of faults we consider in this work, followed by the description of the orchestrator, runtime and deployment service.

3.1 Faults

We consider two types of faults, *context-free* faults and *context-dependent* faults. The first relates to faults that can be injected without any context information. For instance, for injecting CPU load it is not necessary to access any information on the protocol. The second, the context-dependent faults need to access data of the protocol being injected in the system. For example, corrupting a specific packet type for a given consensus instance, we need to access the data contained in the protocol header.

Context-Free Faults

- **CPU Load** - injects a specified amount of CPU load.
- **Crash** - crashes the runtime and associated BFT replica.
- **Sleep** - delays processing for a specified amount of time.
- **Drop packet** - induces a packet drop following a pre-defined policy (one-time or percentage-wise).

Context-Dependent Faults

- **Corrupt header** - corrupts a packet header with the main goal of breaking low-level protocol buffering, i.e., underflow and overflow.
- **Corrupt payload** - corrupts the payload of a packet with erroneous and random information.
- **Forge signatures** - substitutes part of the signature set with forged signatures, in an effort to convince correct nodes of an erroneous value.
- **DDOS** - causes the malicious nodes to start multicast messages to all correct nodes.

3.2 Orchestrator

The orchestrator is the main component of Hermes and its goal is to provide the orchestration between the various runtimes, that are built-in into the replicas and client, and act as a front-end to the developer. In Section 4, we provide an overview of the implementation and an example of its usage.

The interactions between the orchestrator and runtime are built on top of three communication primitives: *RemoteAction*, *Action* and *Notification*.

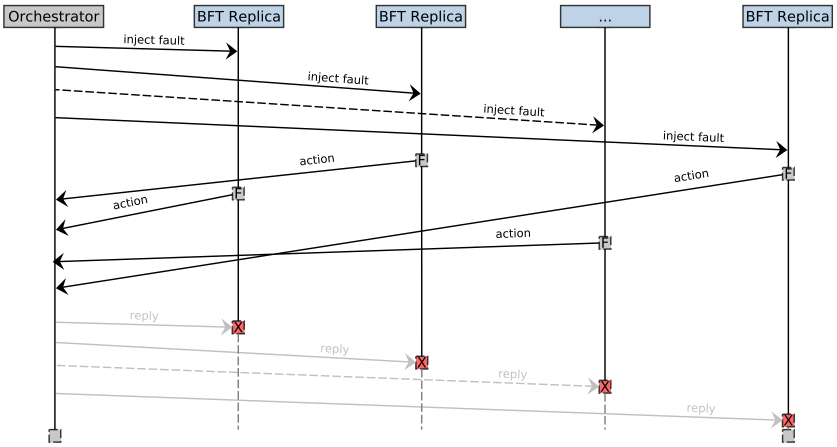

Fig. 2: Action operation overview.

Action

An `Action` is used by the runtime to verify if the fault that is about to be injected into the replica is enabled and ready, or alternatively, if the fault is disabled. While this operation could be used to retrieve information from the overall execution of the injection protocol, its main purpose is to serve as synchronization barrier (shown in Figure 2) for the injection of simultaneous faults. For example, a fault is injected in the packet `send` procedure but it could only be run (injected) when all the malicious nodes reach the same fault. This allows us to test colluding among malicious nodes and also target specific test cases in order to explore specific and tricky/uncommon faults.

For achieving this, each active runtime calls an `Action` when it is about to inject the fault. Because it is a synchronous operation, the runtime waits for the reply from the orchestrator. This reply is only sent by the orchestrator when all the malicious nodes reach the synchronization point, that is, when all the runtimes have called the same `Action`. After receiving the reply, each runtime proceeds and injects the fault.

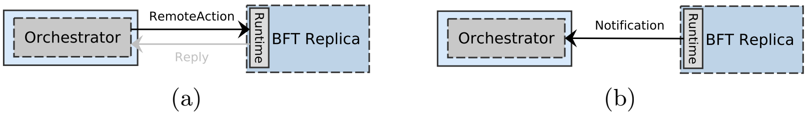

Fig. 3: RemoteAction (left) and Notification (right) operations.

RemoteAction

The `RemoteAction`, shown in Figure 3a, is used by the orchestrator to perform remote procedure calls in the replicas's runtime, and it can be used to manipulate the state of the runtime, or to retrieve some portion of state from the replica.

Notification

The notification mechanism, shown in Figure 3b, provides an asynchronous message passing interface from a runtime to the orchestrator. Its main purpose is to avoid the overhead associated with synchronous operations, i.e. an `Action`, and is used to inform the orchestrator of the progress of the algorithm.

3.3 Runtime

The runtime's main responsibility is to inject faults accordingly to the indications of the orchestrator. The actual fault injection is achieved through the use of Aspect-Oriented Programming (AOP) [27]. We use aspects as a way to seamlessly introduce fault injection points, as well as all the necessary networking infrastructure needed to interact with the orchestrator. We provide more details in Section 4.

3.4 Deployment Service

The deployment service was built to allow remote bootstrap and closure of applications. While it is a general purpose deployment service, its main purpose is to launch replicas and clients to construct the initial view of the system. It should be noted that this does not represent an obstacle for any possible built-in reconfiguration mechanisms, e.g., adding or removing replicas, within the BFT protocol. All the necessary infrastructure to use Hermes is encapsulated inside the runtime, that in turn, is weaved into the replica's (and client) code. Thus, independently of how a replica is deployed, the runtime will perform all the necessary logistics.

4 Implementation

We have implemented the orchestrator in JAVA, while the runtime has implementations in both JAVA and C++. To avoid the complexity/overhead introduced by JAVA existing serialization mechanisms, we custom built a binary protocol that uses little endian encoding without padding. This also allowed us to easily extend Hermes runtime to C++, and in the future will allow to extend to other programming languages, such as python. The implementations are freely available as open-source projects (under an Apache license version 2) at:

- `https://github.com/rolandomar/hermes` (JAVA runtime and orchestration)
- `https://github.com/rolandomar/hermesCPP` (C++ runtime)

Bootstrap Process

It is assumed that each of the hosting nodes has the deployment service running. The orchestrator upon start-up uses the deployment service to create and bootstrap the target protocol nodes, which in this paper are BFT-SMaRt [2] replicas and clients. These nodes were previously weaved, with our runtime and faults, through the use of aspects (shown in Listings 1.3 and 1.4).

After its creation by the deployment service, the runtime within each protocol node bootstraps and connects back to the orchestrator. In turn, the orchestrator creates a barrier for synchronizing the start of all the nodes.

4.1 Orchestrator

An overview of the API provided by the orchestrator is shown in the appendix `http://www.contrib.andrew.cmu.edu/~martinsr/middleware13/apA.pdf`. The `faultInjection()` and `simultaneousFaultInjection()` are the two most important operations offered by the orchestrator. They allow for a single and simultaneous fault injection, respectively. In order to provide further control over the activation of the faults, on the remote runtimes, we use the `Action` primitive. When a runtime reaches a fault, it then uses the `Action` primitive to check if it should proceed with the injection. For example, we use it to only allow the faults to become active after the 500th invocation has taken place (shown in Listing 1.1). For now this value is fixed but can be easily ported as a parameter.

```
public ActionResult onAction(
    OrchestrationNodeServerClient client, Action action)
 switch (action.getSerialID()) {
  case CheckFaultInjectionAction.SERIAL_ID: {
   CheckFaultInjectionAction cfa =
     (CheckFaultInjectionAction) action;
   String faultID = cfa.getFaultID();
   /* omitted code */
   return new CheckFaultInjectionActionResult(
     (cfa.getFaultContext().getRun() < 500));
  }
  /* omitted code */
}
```

Listing 1.1: Orchestration code for fine control over fault injection.

Listing 1.2 shows the code associated with *Attack 1* presented in our evaluation (Section 5). The attack simulates a simultaneous crash of a set of malicious nodes. It starts with the creation of a `CrashFault` associated with the specific injection point, given by `faultID`. In line 5, the orchestrator generates a simultaneous fault, in this case a crash fault. At this point, the orchestrator sends the fault information to the malicious nodes. The test run starts with the creation and bootstrap of a client (line 6-8), identified by `"1001"`, that with perform 1000 invocations, with each invocation incrementing the service counter by 1.

```
1 public void attack1(String[] malicious) {
2  String faultID =
     "2B4FA20ED54E4DA9B6B2A917D1FA723F";
3  HermesFault fault = new CrashFault(faultID);
4  try {
  HermesFuture<Boolean> future =
    simultaneousFaultInjection(malicious,fault);
  String command = HermesConfig.
    getClientCommandLaunch();
  String[] args =
    new String[]command, "1001", "1", "1000";
  launchHermesClient(command, args, 5000);
  boolean ret = future.get(
    TIMEOUT, TimeUnit.MILLISECONDS);
  if(!ret){
    /* handle error */
  }
 } catch (Exception ex) {
    /* handle error */
 }
}
```

Listing 1.2: Orchestrator-side code for Attack 1.

4.2 Runtime Code-Weaving

One of our goals was to make our approach as little intrusive as possible. For that purpose, and as previously explained we used AOP, through the use of AspectJ [3], for the JAVA runtime, and AspectC++ [4] for the C++ runtime. We use two distinct aspects, shown in Listings 1.3 and 1.4.

Runtime Startup Aspect

To seamlessly bootstrap Hermes's runtime into the replica's code, we use aspect `HermesStartupAspect` depicted in Listing 1.3. The aspect is executed before the actual execution of the `main()` procedure. It starts by retrieving the runtime identifier from the list of the application arguments (line 5-6). This identifier is then used to instantiate the singleton's instance (line 7-10). The bootstrap of the runtime is followed by the call to the actual application's `main()` (line 11).

```
 1 @Aspect
 2 public class HermesStartupAspect {
 3  @Around("execution (* bftsmart.demo.counter.
      CounterServer.main*(..))")
 4   public void advice(ProceedingJoinPoint jp)
      throws Throwable
 5   String[] args = (String[]) jp.getArgs()[0];
 6   String id = args[0];
 7   HermesRuntime.getInstance().setID(id);
 8   try {
 9     HermesRuntime.getInstance().open();
10   } catch (Exception e) { /* handle error */}
11   jp.proceed();
12  }
```

Listing 1.3: Runtime bootstrap aspect for replicas and client.

ServersCommunication Aspect

We needed to access the underlying communication infrastructure to inject low-level faults, such as payload corruption. For that purpose, we created the aspect shown in Listing 1.4. For the sake of simplicity and space we only shown the code associated with the payload corruption attack.

The initial portion of the aspect, lines 7-13, checks if the fault is active, and if so retrieves the information about the Paxos protocol, namely, the execution identifier and packet type, e.g., weak and strong packet types. If this information could not be retrieved, i.e., the message is not related to core Paxos protocol, but belongs to surrounding infrastructure such as state transfer, we bypass the fault injection and execute the target code (line 11). In line 14, we update the execution identifier within the fault's context. This information is later sent to the orchestrator in the fault validation, that is executed in line 19.

Because this is a context-dependent fault, the execution of the fault only triggers the verification of the fault's validity. The actual injection is performed in the `sendBytesFailureInjected()` procedure. This procedure is a duplicate of the original code with the added fault injection mechanisms. It was not possible to weave code around this procedure, because it was necessary to access the underlying infrastructure, i.e., in the packet formation we needed to corrupt payload but leave the header intact.

```
 1 @Aspect
 2 public class ServerConnectionAspect {
 3  static public String faultID =
        "5B4FA20ED54E4DA9B6B2A917D1FA724F";
 4  @Around("execution (* bftsmart.communication.
        server.ServersCommunicationLayer.send*(..))")
 5  public void advice(ProceedingJoinPoint jp)
        throws Throwable {
 6   HermesFault fault = HermesRuntime.getInstance().
        getFaultManager().getFault(faultID);
 7   if (fault != null && fault.isEnabled()) {
 8    byte[] msgData = (byte[]) jp.getArgs()[0];
 9    PaxosInfo info = deserialize(msgData);
10    if (info == null) {
11     jp.proceed();
12     return;
13    }
14    fault.updateCtx("RUN", info.getRun());
15    try {
16     switch (fault.getSerialID()) {
17      case BFTForgePayloadFault.SERIAL_ID: {
18       BFTForgePayloadFault faultImpl =
           (BFTForgePayloadFault) fault;
19       faultImpl.execute();
20       int type = faultImpl.getType();
21       Integer attack =
           checkAttack(type,paxosInfo);
```

```
      ServerConnection obj = (ServerConnection)
        jp.getTarget();
      boolean useMac = (boolean) jp.getArgs()[1];
      obj.sendBytesFailureInjected(
        attack,msgData,useMac);
      return;
     }
     /* other faults omitted */
    }
   } catch (Exception ex) {
     /* handle error case */}
  }
 jp.proceed();
 }
}
```

Listing 1.4: ServerConnection aspect.

5 Evaluation

Hardware Setup

We used a NUMA (Non-Uniform Memory Access) workstation with dual hexa-cores, for a total of 12 physical cores and 24 logical threads, with 32GB of RAM and 512GB of RAID-0 storage, comprising 2 SSDs with 256GB each.

For simulating a distributed environment we created 11 virtual machines (VMs), using QEMU/KVM, 10 of which were dedicated to run BFT replicas and 1 for the client. Each VM was allocated with 2GB of RAM and 2 virtual CPUs. The orchestrator ran on the host operating system. Both host the guests used Ubuntu 12.10 LTS as their operating system. Because the virtualized environment already introduces delay and jitter in the network stack (around 1ms latency while measuring with ICMP pings), we only constrained the total amount of bandwidth available (both in-bounding and out-bounding) in each VM to 100Mbit/s, and thus effectively creating a 100Mbit/s network. For the purpose, we used the *TC* command to manipulate the underlying network stacks, with the following commands:

```
tc qdisc add dev eth0 handle ffff: ingress
tc filter add dev eth0 parent ffff: protocol ip prio 50 /
    u32 match ip src 0.0.0.0/0 police rate 100mbit /
    burst 100mbit drop flowid :1
```

5.1 Experiments

Experimental Setup

In order to assess the resiliency of BFT-SMaRt, we performed 1000 invocations per run and injected the faults midpoint, i.e., in the 500th invocation. For such purpose, we devised the following attacks:

- **Attack 1** - *Simultaneous crash*: simultaneously crash all malicious nodes.

- **Attack 2** - *Payload forged with MAX_INT*: all malicious nodes forge their payload size, setting it to MAX_INT (2,147,483,647) bytes.
- **Attack 3** - *Delay Prepare messages below detection timeout*: all malicious nodes delay propose messages to 90% of the timeout used, e.g., for a timeout of 3s the resulting delay would be of 2.7s.
- **Attack 4** - *Delay Prepare messages above detection timeout*: all malicious nodes delay propose messages by 5 times the value of the timeout, e.g., with a timeout of 3s then the delay would be 15s.

Because of space constrains, we only show the results from attacks 1 to 4. The remaining (5 to 9) are available for consultation in the appendix at `http://www.contrib.andrew.cmu.edu/~martinsr/middleware13/apA.pdf`.

For each attack, we tested it against 12 configurations, shown in Table 1, with *1f*, *2f* and *3f* standing for 1, 2 and 3 faults injected, respectively. N is the total number of replicas needed to enforce the $3f+1$ requirement, while M is the set of identifiers for the malicious nodes used in a particular configuration. For example, $M = \{0\}$ stands for the set of malicious nodes only containing node "0", whereas $M = \{x, y\}$ represents a set with two randomly chosen identifiers.

Configurations					
1f, $\|N\| = 4$		*2f*, $\|N\| = 7$		*3f*, $\|N\| = 10$	
#	M	#	M	#	M
0	$\{0\}$	**2**	$\{0\}$	**6**	$\{0\}$
1	$\{x\}$	**3**	$\{x\}$	**7**	$\{x\}$
		4	$\{0, 1\}$	**8**	$\{0, 1\}$
		5	$\{x, y\}$	**9**	$\{x, y\}$
				10	$\{0, 1, 2\}$
				11	$\{x, y, z\}$

Table 1: Configurations used in the attacks evaluation.

For each of these configurations we ran the attack 16 times, and computed the average and the 95% confidence intervals. Each run (a single test) is comprised of 1000 invocations. The maximum amount of time allowed for each run was 5 minutes. After this time, we considered that the run had failed, even if it was not completely stalled or aborted, but rather progressing very slowly.

In our evaluation we collected the following data:

Failed Runs (FR) - the number of failed runs, including stalled or aborted runs with a running time higher then 300s.

Fault-free Latency (LA) - the invocation latency before the fault was injected (in milliseconds).

Faulty Latency (LB) - the invocation latency after the fault was injected (in milliseconds).

Total Duration (D) - the total duration of the run. If the run timed out then the total duration is 300s (in seconds).

Recovery Time (R) - the invocation latency for the 500th and 501st invocations (in seconds).

Total Faulty Invocations (FI) - the number of successful invocations performed after the fault was injected. We only considered the faulty invocations from runs that produced at least 5 invocations after the 500th invocation (otherwise we considered them stalled without recovery).

The application that we chose to run was the `bftsmart.demo.counter`, a simple counter built on top of the BFT-SMaRt protocol. On each invocation we increment the counter by 1. After each successful invocation, the client sends a notification to the orchestrator to report the invocation number and latency. When the orchestrator receives the 1000th invocation from the client, it ends the test and calculates the duration of that run.

The experiments were run twice. We first performed the experiments using the default values and without any modification to the source code, whereas in the second time we tuned the timeout values and made modifications to the source code, of which we provide a detailed discussion on Section 5.3. The relevant parameters used in this work are shown in Table 2.

Parameters		
Parameter Name	Default (ms)	Tuned (ms)
SHORT_TIMEOUT	3000	2000
TOTAL_ORDER_TIMEOUT	10000	3000
CONNECTION_TIMEOUT	10000	3000
INVOCATION_TIMEOUT	40000	60000

Table 2: Parameters considered in the evaluation.

For certain cases the `SHORT_TIMEOUT` is used to quickly trigger a reconfiguration, such as a voluntary exit from a group. The `TOTAL_ORDER_TIMEOUT` is the main timeout used in the implementation and is used to detect when a request was not processed and subsequently trigger the leader-change protocol. The `CONNECTION_TIMEOUT` controls the timeout associated with the establishment of a new connection to a replica. The last parameter, `INVOCATION_TIMEOUT`, is used by the client, more specifically through the `ServiceProxy`, to control the timeout associated with each invocation.

5.2 Results

In this section we will start to discuss the performance using the default parameters and without any modifications to the source code (left sub-table on the results tables 3 to 6). Later, using the knowledge gained throughout the first round of our evaluation, we show how it helped us to track the underlying issues and partially overcome them by tuning some of the system parameters (Table 2) and by applying a modification to the source code (right sub-table).

Generically, we can see throughout the results that the increase in the number of nodes in the system, from 4 to 10 nodes (*1f* to *3f*) results in a linear increase on the fault-free invocation latency, from 16ms to 20ms (see column **LA** in tables 3 to 6). In some cases, the faulty invocation latency drops after a fault has occurred. Because the number of nodes decreases, the latency (and overhead) associated with the protocol also decreases, except in attacks 3 and 4. The recovery time is higher when the leaders are attacked, going upwards to 30s in attack 3. We provide a discussion on the reasons beyond this high recovery time in Section 5.3.

Attack 1

Our goal with attack 1 was to evaluate the impact of simultaneously crashing multiple nodes in the system (for *2f* and *3f* configurations). The results from the single fault scenario (*1f* configurations) are presented to provide a baseline comparison. The results from our evaluation of attack 1 are shown in Table 3. In our initial evaluation, using the default values and implementation, we encountered some issues with configurations 4, 8 and 10. These issues seemed related with the change-leader protocol.

After a manual inspection of logs, we found that the problem was a composition of several issues. First, we found that 40s invocation timeout (from within the client's `ServiceProxy`) was too short, and it should be at least 60s. The stalls that we checked in the results were a direct result from this. When this timeout is triggered, the client aborts its execution and the run ends. This also hid the true values of the recovery time, that can reach roughly 60s (which we concluded after some additional experimentation with larger timeouts).

Overall, we found that the default timeout values (associated with the replicas) were too high for a LAN (Ethernet based network). But that did not account for the low performance that we detected in the protocol after the injections of the faults. After an inspection to the source code, we found a subtle bug in the timeout management that leads to problems in the change-leader protocol. We provide a better explanation to this problem later in Section 5.3.

Attack 1 (Crash Fault)												
	Default						*Tuned*					
C	FR (%)	LB (ms)	LA (ms)	D (s)	R (s)	FI (#)	FR (%)	LB (ms)	LA (ms)	D (s)	R (s)	FI (#)
0	0	16±2	9±0	37±0	20±0	499±0	0	16±2	9±0	23±0	6±0	499±0
1	0	16±2	9±0	19±3	2±3	499±0	0	16±2	8±0	18±1	1±1	499±0
2	0	18±1	10±0	38±0	20±0	499±0	0	18±1	10±0	24±0	6±0	499±0
3	0	17±1	10±0	22±3	3±3	499±0	0	17±1	10±0	18±0	0±0	499±0
4	100	18±1	*N/A*	300±0	*N/A*	*N/A*	0	18±1	11±0	30±0	11±0	499±0
5	6	17±1	11±0	38±33	2±3	499±0	0	17±1	11±0	21±1	2±1	499±0
6	0	20±0	12±0	41±0	20±0	499±0	0	20±0	12±0	27±0	6±0	499±0
7	0	20±0	12±0	21±2	1±2	499±0	0	20±0	12±0	20±0	0±0	499±0
8	100	20±0	*N/A*	300±0	*N/A*	*N/A*	0	20±0	12±0	32±0	12±0	499±0
9	0	20±0	12±0	26±4	6±4	499±0	0	20±0	12±0	21±1	0±0	499±0
10	100	20±0	*N/A*	300±0	*N/A*	*N/A*	0	20±0	14±1	36±0	14±0	499±0
11	13	20±0	13±0	55±45	0±0	499±0	0	20±0	13±0	22±1	1±1	499±0

Table 3: *Attack 1* with the default (left) and the tuned (right) configurations.

Attack 2

We designed attack 2 for assessing the impact of overflowing in the protocol. This was achieved by modifying the length field on the `PaxosMessage` packet. The results from our evaluation are show in Table 4.

Attack 2 (Value/Corruption Fault)												
	Default						*Tuned*					
C	FR (%)	LB (ms)	LA (ms)	D (s)	R (s)	FI (#)	FR (%)	LB (ms)	LA (ms)	D (s)	R (s)	FI (#)
0	0	16±2	8±0	36±0	20±0	499±0	0	16±2	8±0	23±0	6±0	499±0
1	0	16±2	8±0	21±4	5±4	499±0	0	16±2	8±0	17±0	0±0	499±0
2	0	18±1	10±0	38±0	20±0	499±0	0	17±1	10±0	24±0	6±0	499±0
3	0	17±1	10±0	23±4	5±4	499±0	0	17±1	10±0	18±0	0±0	499±0
4	100	18±1	*N/A*	300±0	*N/A*	*N/A*	0	18±1	11±0	27±0	9±0	499±0
5	13	17±1	11±0	63±43	11±5	499±0	0	17±1	11±0	20±1	1±1	499±0
6	0	20±0	13±0	40±0	20±0	499±0	0	20±0	13±0	26±0	6±0	499±0
7	0	20±0	13±0	22±2	1±2	499±0	0	20±0	13±0	21±0	0±0	499±0
8	100	20±0	*N/A*	300±0	*N/A*	*N/A*	0	20±0	13±0	30±0	9±0	499±0
9	6	20±0	14±0	42±32	4±4	499±0	6	20±0	13±0	39±32	1±1	499±0
10	100	20±0	*N/A*	300±0	*N/A*	*N/A*	0	20±0	15±0	34±0	12±0	499±0
11	0	20±0	15±0	31±4	10±4	499±0	0	20±0	15±0	24±1	2±1	499±0

Table 4: *Attack 2* with the default (left) and the tuned (right) configurations.

The presence of failures indicates that the protocol is susceptible to the overflow attack (also to underflow and payload corruption, c.f. in appendix under attacks 6 and 7). Furthermore, the failure pattern from attack 2 closely resembles the pattern of attack 1, which indicates that the underlying causes should be the same or similar.

Using this insight, we were able to successfully track down the root of this behavior within the source code. We found that the crash (caused by the overflow or underflow) of the deliver thread in the low-level communication infrastructure is omitted from the overall leader module management. The same applies when we corrupt the payload of messages. The deliver thread detects the mismatch between the payload and the signature but silently ignores it. The protocol is able to recover because the timeout associated with the request is triggered forcing the leader-change sub-protocol to change the leader. It is important to note that BFT-SMaRt is implemented using the principle of decoupling the total ordering of requests from the actual consensus primitive [28]. The client sends its requests to every replica, not only the leader. This is done to prevent a malicious leader to stall the protocol. When a replica (non-leader) detects that the leader did not propose the request, it forwards the request to the current leader and activates a new timeout. If this fails, then a new regency is activated through the leader-change sub-protocol. Later in Section 5.3, we discuss the impact of this decision on the overall protocol performance.

Possibly it would be feasible to use information from the low-level communication layer to provide further knowledge to the leader module in order to speedup the recovery process when malicious nodes are in the role of the leader. We discuss a possible implementation of this approach later in Section 5.3.

Attack 3 and 4

The use of a leader in BFT protocols creates a potential attack point. This subject was previously explored by Prime [21], which shows the impact of the presence of a malicious leader. We devised attacks 3 and 4 for assessing the resilience of BFT-SMaRt to this kind of attack.

In certain cases, a reconfiguration can be triggered with a short timeout, that is about one third of the regular timeout (3s in the default configuration and 2s in our tuned configuration). To avoid detection, we delay conservatively the sending of prepare messages by 90% of the value of this timeout (2.7s for the default configuration and 1.8s for the tuned configuration). The results from attack 3 (Table 5) show that delay of the prepare messages by leader causes a increase of invocation latency to around 2.7s, as expected, causing the runs to fail as they take more than 300s to finish.

Alternatively, in attack 4 (Table 6) we used 5 times the value of the short timeout, resulting in a timeout of 15s, for the default configuration, and 10s for the tuned version. This clearly triggers the change-leader sub-protocol but eventually the protocol itself stalls when about 15 faulty invocations have been processed. Again, the failure pattern also shows some correlation with the failure pattern of attack 1.

Attack 3 (Timing Fault, Delay Below Timeout)												
	Default						*Tuned*					
C	FR (%)	LB (ms)	LA (ms)	D (s)	R (s)	FI (#)	FR (%)	LB (ms)	LA (ms)	D (s)	R (s)	FI (#)
0	100	16±2	2723±0	300±0	5±0	104±0	100	16±2	1822±0	300±0	3±0	157±0
1	13	16±2	86±10	51±46	0±0	449±63	19	16±2	130±10	69±54	0±0	434±65
2	100	18±1	2729±0	300±0	5±0	104±0	100	18±1	1828±0	300±0	3±0	156±0
3	13	17±1	88±10	53±45	0±0	449±64	6	17±1	46±5	35±33	0±0	477±40
4	100	18±1	2730±0	300±0	5±0	104±0	100	18±1	1828±0	300±0	3±0	156±0
5	25	17±1	186±16	88±59	1±1	400±83	38	17±1	296±16	123±66	1±0	370±81
6	100	20±0	2737±0	300±0	5±0	103±0	100	20±0	1835±0	300±0	3±0	154±0
7	6	20±0	49±7	38±33	0±0	474±46	13	20±0	89±8	55±45	0±0	456±55
8	100	20±0	2736±0	300±0	5±0	103±0	100	20±0	1835±0	300±0	3±0	154±0
9	19	20±0	136±13	73±53	1±1	424±75	19	20±0	134±10	72±53	0±0	434±65
10	100	20±0	2737±0	300±0	5±0	103±0	100	20±0	1836±0	300±0	3±0	154±0
11	38	20±0	312±22	126±66	2±1	350±93	50	20±0	444±21	160±68	1±0	327±84

Table 5: *Attack 3* with the default configuration (left) and the tuned version (right)

5.3 Lessons Learned

Although we are still in the process of analyzing all the data and logs that we have collected, we found some interesting issues within BFT-SMaRt implementation. Although the BFT-SMaRt is a leader-based BFT protocol we were surprised to verify the impact of simple crash faults on the system. While the implementation was able to sustain flawlessly single crash faults or even multiple random crash faults, it was vulnerable, performance wise, to the simultaneous injection of crash faults within the first elements of the nodes, i.e., the initial leaders.

1) After analyzing the code, we detected a misconfiguration of the client, which normally has a default 40s request timeout. When this timeout is triggered within the client, it aborts the execution of the run. This was the reason behind the stalls we detected while using the default timeout values and original implementation.

Attack 4 (Timing Fault, Delay Above Timeout)												
	Default						*Tuned*					
C	FR (%)	LB (ms)	LA (ms)	D (s)	R (s)	FI (#)	FR (%)	LB (ms)	LA (ms)	D (s)	R (s)	FI (#)
0	56	16±2	915±117	252±35	30±0	228±116	0	16±2	8±0	22±0	6±0	499±0
1	13	16±2	102±28	66±46	9±6	438±77	0	16±2	8±0	17±1	0±0	499±0
2	19	17±1	353±55	187±38	30±0	408±91	0	17±1	10±0	24±0	6±0	499±0
3	0	17±1	53±18	45±32	5±5	499±0	0	17±1	10±0	19±1	1±1	499±0
4	100	18±1	16058±574	300±0	30±0	15±1	0	17±1	11±0	30±0	12±0	499±0
5	0	17±1	89±24	64±42	7±6	499±0	0	17±1	10±0	20±1	2±1	499±0
6	38	20±0	357±66	202±39	30±0	336±116	0	20±0	12±0	26±0	6±0	499±0
7	6	20±0	34±15	38±33	1±3	468±58	0	20±0	12±0	21±1	1±1	499±0
8	100	20±0	16196±640	300±0	30±0	13±2	0	20±0	13±0	32±0	12±0	499±0
9	19	20±0	101±31	73±53	5±5	408±92	0	20±0	13±0	23±1	2±1	499±0
10	100	20±0	15968±521	300±0	30±0	14±2	0	20±0	27±5	40±0	12±0	499±0
11	25	20±0	154±38	91±59	7±6	379±101	0	20±0	16±0	25±3	3±2	499±0

Table 6: *Attack 4* with the default configuration (left) and the tuned version (right)

2) Because BFT-SMaRt is leader-based, faults in the leader have a high impact on the overall performance (and recovery) of the protocol. After increasing this timeout we were able to measure a recovery time around 60s. The high recovery time that we measured is only partially explained by the high timeout values. An analysis of the source code revealed that the protocol, in the presence of a request timeout, first tries to forward pending requests to the current leader. This was done to accommodate the possibility that the current leader did not propose the request because it might have been dropped by the underlying network infrastructure. However, in the presence of a single malicious leader, this in fact doubles the recovery time. This is because the request is first forwarded to the leader, and the change-leader sub-protocol only runs after this operation has timed out. On top of this, the leader-change protocol goes sequentially through the processes list to find the next leader (from the lowest to the highest identifier). Because we intentionally injected simultaneous faults in the nodes with the lowest identifiers (that are the first nodes to be elected as leaders), this resulted in the protocol having to go through all the malicious nodes. Furthermore, they immediately assume the roles of leaders, introducing further delays. To minimize this situation, we lowered the timeout values (shown in Table 2).

3) However, this alone did not fully explain the extensive recovery times. A closer inspection of the source code reveled a subtle implementation artifact. The timer used for failure detection (`RequestsTimer`) for all the pending requests present in the system gets its timeout value doubled each time the change-leader protocol is triggered (for example, when receiving a `STOP` and `STOP_DATA` messages from other nodes), but it is never reduced, even in the presence of more favorable system conditions, such as the absence of failures or timeouts. Given this, we introduced a modification to the original source code that consisted on only doubling the timeout within the same regency (i.e., the same leader) on the current view, otherwise the value is reset to the default value. Using our tuned parameters and correction to the source code (right sub-table on the results tables) we were able to almost avoid any failed runs, except in attack 3. Sporadically we got a failed run while using attack 2. We are in the process of analyzing the logs to determine the underlying issues associated with those failed runs.

4) Attack 3 was designed in light to previous work on Prime [21], and was designed to degrade overall performance of the system by delaying the sending of propose messages. The attack was designed so that the leader was not suspected by the other nodes, by limiting the amount of delay just below to the detection timeout value. By lowering the timeout value we were able to minimize the impact of this type of attack, although it is evident that a more proactive and structural approach has to be taken to solve this issue when using leader-based BFT protocols. While Prime provides a way to minimize this situation, it still does not provide a complete solution, because of the limitations derived from the use of Diffserv [29], because a flooding attack would compromise the measurements used to adapt the timeouts. We tried to perform a full evaluation of Prime but we discover after the initial tests that the protocol was not completely implemented. Therefore, we were unable to verify their results. To corroborate our findings, we can see in attack 4 that if a sufficiently small timeout is chosen then the attack is contained, with no apparent loss in performance (except for the recovery time associated with the election of the new leaders).

5) For Attack 2 we were able to manually verify the code and found a missing verification in the creation of the packet by the receiving thread (`ReceiverThread` in the `ServerConnection` class). Because it does not verify the size of the payload, it allows the attacker to crash the thread either with overflow or underflow attacks. While JAVA provides a managed memory system, some vulnerabilities have been found in the past that explore such cases.

6) Taking the knowledge gathered throughout our evaluation we implemented a second modification to the original protocol. As stated in the discussion of Attack 2, we make use of the low-level network events such as unexpected network errors (e.g., closing of TCP connections) or packet malformation (such as mismatched signatures) to trigger the change-leader sub-protocol. The results from this second modification are shown in the appendix, under section "Second Round of Corrections". We were able to cut roughly in half the recovery time from our first implementation modification (and timeout tuning). From the original implementation and timeout values we were able to provide up to a 10 fold improvement. Nevertheless, it should be noted that these modifications come at a price. Because we are assuming a more synchronous network layer and assuming that any type of fault we get from the network is malicious, we could be inducing false suspicion on nodes leading to unnecessary leader changes. However, this would only affect performance but not correctness.

BFT Limitations and Future Directions

It seems clear that leader-based BFT protocols have their weakest point in the leader and change-leader sub-protocol. The timeout settings also play an important role in the overall performance.

For situations without the presence of malicious node, the introduction of adaptive timeouts like in Adaptare-FD [30], could improve seamlessly the overall performance. However to efficiently deal with the presence of malicious nodes, current approaches [21] are still not able to deliver degradation-free performance. While making the assumption that the network is totally asynchronous makes a strong case from a standpoint of correctness and safety, we feel that in order to bridge theory and practice, stronger, yet realistic, assumptions must be made about underlying network infrastructure. The use of software-defined networking, for instance by OpenFlow [31], could allows us to improve on the current state-of-the-art, such as avoiding the issues related with flooding attacks [1].

6 Conclusions and Future Work

In this paper we presented a novel fault-injecting framework that enables the assessment of BFT implementations. Furthermore, we demonstrate the importance of providing support for non-independent faults. Using our approach we were able to detect 2 implementation bugs. The first, at the low communication level, allowed overflow and underflow attacks caused by a missing size verification on packet reception, whereas the second bug was related to a high level implementation artifact derived from the ever-increasing timeout values within the leader-change sub-protocol, that in certain cases would effectively stall the protocol for more than 60s in the presence of non-independent faults. Lastly, we proposed a second set of modifications to the source code, where we avoid forwarding messages to a possibly malicious leader prior to a change in regency and enhance it by using low-level networking exceptions/events, such as signature mismatch, to trigger a leader change when in the suspicion of the presence of a malicious leader.

Using our tuned parameters and source code modifications we were able to provide up to a 10 fold improvement over the original implementation and default parameters.

6.1 Future Work

We expect to enhance Hermes by providing the necessary infrastructure to support proof forging, by coordinating and distributing all the necessary keys across the malicious nodes. Furthermore, we want to expand the work accomplished in this paper, by introducing a visualization tool that continuously monitors and traces all the nodes present in the protocol with the goal of providing the initial support for debugging.

Acknowledgments

We thank Alysson Bessani and the conference reviewers for their feedback. This research was sponsored in part by the project CMUPT/RNQ/0015/2009 (TRONE - Trustworthy and Resilient Operations in a Network Environment), and by Intel via the Intel Science and Technology Center for Cloud Computing (ISTC-CC).

References

1. Allen Clement, Edmund Wong, Lorenzo Alvisi, Mike Dahlin, and Mirco Marchetti. Making Byzantine Fault Tolerant Systems Tolerate Byzantine faults. In *Proceedings of the 6th USENIX Symposium on Networked Systems Design and Implementation*, NSDI '09, pages 153–168, Berkeley, CA, USA, 2009. USENIX Association.
2. BFT-SMaRt: High-Performance Byzantine Fault-tolerant State Machine Replication. `http://code.google.com/p/bft-smart/`. [Online; accessed 4-11-2013].
3. Kiczales, Gregor and Hilsdale, Erik. Aspect-Oriented Programming. In *ACM SIGSOFT Software Engineering Notes*, volume 26, page 313. ACM, 2001.
4. Olaf Spinczyk, Andreas Gal, and Wolfgang Schröder-Preikschat. AspectC++: an Aspect-Oriented Extension to the C++ Programming Language. In *Proceedings of the 40th International Conference on Tools Pacific: Objects for Internet, Mobile and Embedded Applications*, pages 53–60. Australian Computer Society, Inc., 2002.
5. R. Chandra, R. M. Levefer, M. Cukier, and W. H. Sanders. Loki: A State-Driven Fault Injector for Distributed Systems. In *International Conference on Dependable Systems and Networks*, pages 237–242, June 2000.
6. DBench Project Final Report, May 2004.
7. S. Han, Rosenberg H. A., and K. G. Shin. Doctor: An integrateD sOftware fault injeCTiOn enviRonment. In *International Computer Performance and Dependability Symposium*, pages 204–213, April 1995.
8. G. A. Alvarez and F. Cristian. Centralized Failure Injection for Distributed, Fault-Tolerant Protocol Testing. In *International Conference on Distributed Computing Systems*, pages 78–85, May 1997.
9. S. Dawson, F. Jahanian, T. Mitton, and T.-L. Tung. Testing of Fault-Tolerant and Real-Time Distributed Systems via Protocol Fault Injection. In *Symposium on Fault Tolerant Computing*, pages 404–414, June 1996.
10. N. Looker and J. Xu. Assessing the Dependability of OGSA Middleware by Fault Injection. In *Proceedings of the 22th IEEE International Symposium on Reliable Distributed Systems*, SRDS' 03, pages 293–302, October 2003.
11. E. Marsden and J.-C. Fabre. Failure Analysis of an ORB in Presence of Faults. Technical report, October 2001.
12. G. A. Kanawati, N. A. Kanawati, and J. A. Abraham. FERRARI: A Flexible Software-Based Fault and Error Injection System. *IEEE Transactions on Computers*, 44(2):248–260, February 1995.
13. Timothy K. Tsai and Ravishankar K. Iyer. Measuring Fault Tolerance with the FTAPE Fault Injection Tool. In *Proceedings of the 8th International Conference on Modelling Techniques and Tools for Computer Performance Evaluation: Quantitative Evaluation of Computing and Communication Systems*, MMB '95, pages 26–40. Springer-Verlag, 1995.
14. J. Carreira, H. Madeira, and J. G. Silva. Xception: Software Fault Injection and Monitoring in Processor Functional Units. In *Proceedings of the 5th Annual IEEE International Working Conference on Dependable Computing for Critical Applications*, DCCA '95, pages 135–149.
15. J. DeVale, P. Koopman, and D. Guttendorf. The Ballista Software Robustness Testing Service. In *Proceedings of Testing Computer Software*, 1999.
16. Mei-Chen Hsueh, Timothy K. Tsai, and Ravishankar K. Iyer. Fault Injection Techniques and Tools. *Computer*, 30(4):75–82, April 1997.
17. Miguel Castro and Barbara Liskov. Practical Byzantine Fault Tolerance and Proactive Recovery. *ACM Transactions on Computer Systems*, 20(4):398–461, February 2002.

18. Michael Abd-El-Malek, Gregory R. Ganger, Garth R. Goodson, Michael K. Reiter, and Jay J. Wylie. Fault-scalable Byzantine Fault-Tolerant Services. *SIGOPS Operating Systems Review*, 39(5):59–74, October 2005.
19. Ramakrishna Kotla, Lorenzo Alvisi, Mike Dahlin, Allen Clement, and Edmund Wong. Zyzzyva: Speculative byzantine fault folerance. In *Proceedings of 21st ACM SIGOPS Symposium on Operating Systems Principles*, SOSP '07, pages 45–58, New York, NY, USA, 2007. ACM.
20. James Cowling, Daniel Myers, Barbara Liskov, Rodrigo Rodrigues, and Liuba Shrira. HQ Replication: A Hybrid Quorum Protocol for Byzantine Fault Tolerance. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation*, SOSDI '06, pages 177–190. USENIX Association, 2006.
21. U. Amir., B. Coan, J. Kirsch, and J. Lane. Prime: Byzantine Replication under Attack. *Dependable and Secure Computing, IEEE Transactions on*, 8(4):564 –577, July-August 2011.
22. Y. Amir, C. Danilov, D. Dolev, J. Kirsch, J. Lane, C. Nita-Rotaru, J. Olsen, and D. Zage. Steward: Scaling Byzantine Fault-Tolerant Replication to Wide Area Networks. *IEEE Transactions on Dependable and Secure Computing*, 7(1):80–93, 2010.
23. Jian Yin, Jean-Philippe Martin, Arun Venkataramani, Lorenzo Alvisi, and Mike Dahlin. Separating Agreement From Execution for Byzantine Fault Tolerant Services. *ACM SIGOPS Operating Systems Review*, 37(5):253–267, 2003.
24. J-P Martin and Lorenzo Alvisi. Fast byzantine consensus. *IEEE Transactions on Dependable and Secure Computing*, 3(3):202–215, 2006.
25. Yair Amir, Brian Coan, Jonathan Kirsch, and John Lane. Customizable Fault Tolerance forWide-Area Replication. In *Proceedings of the 26th IEEE International Symposium on Reliable Distributed Systems*, SRDS '07, pages 65–82. IEEE, 2007.
26. Jinyuan Li and David Mazieres. Beyond One-Third Faulty Replicas in Byzantine Fault Tolerant Systems. In *Proceedings of the 4th USENIX Symposium on Networked Systems Design and Implementation*, NSDI '07, 2007.
27. Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Videira Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-oriented programming, in proceedings of the european conference on object-oriented programming (ecoop), finland. *Springer-Verlag LNCS*, 1241:16, 1997.
28. Joo Sousa and Alysson Bessani. From Byzantine Consensus to BFT State Machine Replication: A Latency-Optimal Transformation. In *Proceedings of the 9th European Dependable Computing Conference*, EDCC '12, pages 37–48, Washington, DC, USA, 2012. IEEE Computer Society.
29. IETF. An Architecture for Differentiated Services. `http://www.ietf.org/rfc/rfc2475.txt`. [Online; accessed 17-October-2011].
30. Mônica Dixit, António Casimiro, Paolo Lollini, Andrea Bondavalli, and Paulo Verissimo. Adaptare: Supporting Automatic and Dependable Adaptation in Dynamic Environments. *ACM Transactions on Autonomous and Adaptive Systems (TAAS)*, 7(2):18, 2012.
31. McKeown, Nick and Anderson, Tom and Balakrishnan, Hari and Parulkar, Guru and Peterson, Larry and Rexford, Jennifer and Shenker, Scott and Turner, Jonathan. OpenFlow: Enabling Innovation in Campus Networks. *ACM SIGCOMM Computer Communication Review*, 38(2):69–74, 2008.