Proceedings of the International Conference on Parallel and Distributed Computing and Systems
Marina del Rey, California, pp. 210-217, November 2010.

USING VIRTUALIZATION TO VALIDATE FAULT-TOLERANT DISTRIBUTED SYSTEMS

Israel Hsu, Andrew Gallagher, Michael Le, Yuval Tamir
Concurrent Systems Laboratory
UCLA Computer Science Department
Los Angeles, California, U.S.A.
{israel,ajcg,mvle,tamir}@cs.ucla.edu

ABSTRACT
Asynchronous events and complex system state distributed across independent nodes make exposure and diagnosis of flaws in distributed systems a challenge. The difficulties are exacerbated when the goal is to validate fault tolerance mechanisms that are activated only by the occurrence of errors, which are, by nature, rare. Validation of fault tolerance mechanisms is often done by injecting faults that emulate the actual faults and ‘‘stress’’ the functionality of the resilience mechanisms. Validation campaigns lasting days and involving thousands of fault injections are often necessary. We present an infrastructure that combines virtualization and software-implemented fault injection to automate validation campaigns and support the analysis of the behavior of a distributed system under test. Virtualization enables: 1) a flexible fault injector capable of emulating a wide variety of faults, and 2) a mechanism for autonomously recovering faulty nodes so that the campaign can continue running on a target system that is fully functional. As a case study we use this infrastructure to validate a Byzantine-fault-tolerant cluster manager. Over 1280 hours of fault injections yielded the exposure of 11 unique flaws in the cluster manager.

KEY WORDS
Fault Injection, Dependability, Validation Tools.

1. Introduction

To maximize reliability and availability, distributed systems often employ fault tolerance mechanisms that allow them to continue to operate correctly despite hardware or software faults. The operation of these mechanisms cannot be validated by field testing since faults are so infrequent that fault tolerance mechanisms are rarely exercised. Instead, software-implemented fault injection (SWIFI) is often used to verify system operation under a variety of fault scenarios at a greatly accelerated rate [10, 8].

Using fault injection, validating distributed systems is still challenging since erroneous behavior may only manifest when a rare ordering of asynchronous events occurs or a specific distributed system state is reached. Hence, a flexible infrastructure for unattended execution of injection campaigns is necessary. There are three key requirements from such an infrastructure: 1) it must allow stressing of corner cases, coordinated across multiple nodes, as well as lengthy randomized testing; 2) even when running many thousands of injections, it must ensure that the initial system state for each test is error-free; 3) fault injections, system responses, and resource usage must be logged for off-line analysis.

This paper presents an infrastructure that meets the requirements above by combining SWIFI with system virtualization. Virtualization [17] is leveraged to allow campaigns to continue autonomously following failures in the system under test, reduce the hardware resources required for testing, and facilitate injection to the OS kernels of system nodes [12]. The infrastructure is optimized for validating closely-coupled distributed systems (clusters). As part of this work, we present a detailed case study of using the infrastructure to validate a Byzantine-fault-tolerant cluster manager, including discovery of several critical flaws. Our results demonstrate the critical importance of the flexibility and autonomous operation provided by our infrastructure.

The use of fault injection to test distributed systems [19, 7, 8, 6] and leveraging virtualization for fault injection [2, 3, 18, 16, 9, 12] have been presented in previous works. However, unique features of our work include: A) a comprehensive presentation of all aspects of a practical testing infrastructure, and B) a detailed case study of the use of the infrastructure to validate a system with multiple advanced fault tolerance features.

Section 2 describes the validation infrastructure based on system-level virtualization. A case study of the use of this infrastructure to validate a Byzantine-fault-tolerant cluster manager is presented in Section 3. Sections 4 and 5 present, respectively, related work and our conclusions.

2. Validation Infrastructure

This section describes our validation infrastructure: 1) the virtualized distributed system; 2) the fault injector; 3) the management of the injection campaigns; and 4) the facility for logging information used for postmortem analysis.

2.1. A Virtualized Distributed System

Since validation requires subjecting the system to a large number of faults, a meaningful validation campaign must execute for long durations. Hence, it is desirable to run multiple campaigns simultaneously on multiple distributed system instances. This may be impractical in many environments if each distributed system is actually composed of multiple physical computers. This issue provides part of the motivation for using system-level virtualization technology to run multiple nodes of the distributed system under test on a single physical computer. In particular, for the experiments we report here, the entire distributed system is consolidated on a single physical host.

Virtualization technology allocates a computer's resources, such as CPUs and I/O devices, to multiple virtual machines (VMs). Each node of the distributed system runs as a VM, and each VM runs its own OS and user-level software. A virtual machine monitor (VMM) enforces isolation among the VMs so that the activities of one VM do not affect other VMs. We use the Xen VMM [1], which gives direct access to the physical computer's hardware devices to one privileged VM. The privileged VM can start, halt, or shut down unprivileged VMs. It also hosts disk images for the virtual disks in each unprivileged VM and routes the network traffic among the VMs and with other computers outside the virtualized environment.

In addition to reducing the required hardware resources, running nodes in VMs provides three main benefits over using physical machines. First, it enables implementing fault injection software partially or completely outside of the VM, minimizing intrusion on the system under test. Second, nodes implemented as VMs can be easily and quickly power-reset without specialized hardware. Finally, the virtualized environment can provide a lightweight communication facility among VMs through shared memory that can be used for coordinating fault injections and for logging system responses.

2.2. Fault Injection in a Virtualized Environment

To be useful, a fault injector must be flexible in the types of faults that can be injected, the times at which faults can be injected, and the targets where faults can be injected. We have developed *Gigan* [12], a flexible SWIFI capable of injecting a variety of faults into OS kernels and user-level processes. *Gigan* can operate in non-virtualized systems but also has capabilities optimized for virtualized systems. *Gigan's* operation is based on *triggers* and *actions*. *Triggers* are set to fire after some threshold has been reached or some event has occurred in the target machine. Triggers can fire based on timers, instruction breakpoints, process creation/termination, and performance monitoring events (e.g., CPU cycle count) [4]. Associated with each trigger is a set of one or more *actions* to be performed at the time the trigger is fired. To maximize flexibility, an action either injects a fault or sets another trigger.

The purpose of a fault injection campaign is to validate system operation under some well-defined fault model. For example, the goal may be to validate correct operation as long as, within a period of **T** seconds, no more than **k** nodes in the system operate erroneously. Hence, for a particular experiment, the injector must not inject

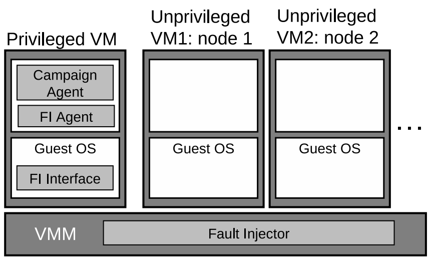

Figure 1: Architecture of VMM-level Injector

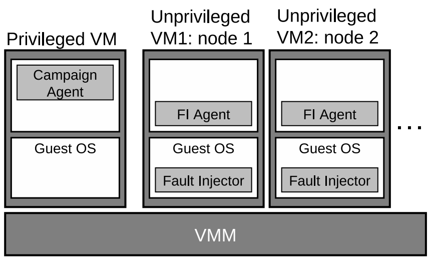

Figure 2: Architecture of OS-level Injector

additional faults once the worst-case scenario being tested is reached.

Gigan implements two approaches to fault injection: the VMM-level injector (Figure 1) and the OS-level injector (Figure 2). With the first approach, the injector is implemented completely outside of the VMs. This implementation has four components: the *Fault Injector* in the VMM, the *Fault Injector Agent (FI Agent)*, the *Fault Injector Interface (FI Interface)*, and the *Campaign Agent*. The Campaign Agent tracks the progress of the campaign, gathers information about the state of the system under test, and instructs the FI Agent to pause or resume fault injections. The FI Agent creates the triggers and actions as specified by the fault injection campaign. At the beginning of each test, the FI Agent sets triggers and actions by sending commands to the Fault Injector via the FI Interface. When a trigger fires, the Fault Injector executes the associated actions.

With any fault injection, it is desirable to minimize *intrusiveness*, i.e., minimize the impact of the fault injection on the behavior of the target system. The VMM-level injector can operate without any knowledge of the internal structure of a VM running as a node in the system under test. Furthermore, it does not require any changes to such VMs and has essentially no impact on their normal operation. However, treating the VM as a ''black box'' does not allow targeting specific user-level processes or OS data structures within the VM. The only exception to this opacity is that the Fault Injector can distinguish between the VM's execution of processes and OS-level code. This exception is due to the fact that the VM OS runs in a higher hardware privilege level than the user-level processes, and the act of switching privilege levels is visible to the VMM. The VMM-level injector can use this visibility to target the VM OS by injecting faults only when the CPU is executing at the higher privilege level.

To target user-level processes, a VMM-level injector would need to be able read and parse data structures of the guest OS, such as process tables. Rather than add such complexity and OS dependency to the VMM-level injector, *Gigan* implements a second approach to fault targeting, the OS-level injector (Figure 2). The OS-level injector consists of an FI Agent running inside each VM to be targeted, the Fault Injector as a module in each VM's guest OS, and the Campaign Agent running in the privileged VM. As a user-level process, the FI Agent can invoke services of the guest OS to gather information on the state of user-level processes and use this information to set triggers and actions. Triggers and actions are set via system calls to the Fault Injector module. When a trigger fires, this module executes the associated actions. As with the VMM-level injector, the Campaign Agent signals the FI Agents to pause and resume fault injections. Because the FI Agent runs only to set triggers and actions and the Fault Injector is active only when triggers are fired, there is negligible intrusion by these components on the operation of the VMs.

OS-level FI Agents must be coordinated so that fault injections into the target VMs occur as specified by the fault injection campaign. With *Gigan's* OS-level FI Agents, coordination is based on synchronized clocks and network communication. The Network Time Protocol (NTP) is used to keep the wall clocks of the VMs synchronized. In some cases, the FI Agents on the different VMs are assigned different time slots during which they can perform injection. In addition, the Campaign Agent can command FI Agents to pause or resume injection using user-level signals sent via SSH connections. The virtualized environment does enable an alternative method of coordination with potentially lower intrusion: processes on different VMs can communicate using memory pages that are shared among VMs. We chose to use time for coordination over the lower-intrusion shared memory design for two reasons. First, NTP and user-level signals use very few system resources such that the gain of even lower intrusion does not justify the added complexity of sharing memory between VMs. Second, the OS-level FI agents as designed are portable to a validation infrastructure that uses physical machines instead of VMs.

2.3. Injection Campaign Management

In order to be able to interpret the results of the injection campaign and use them to correct flaws in the system, each experiment (injection) must be performed starting with a fault-free system. Specifically, latent erroneous state from one injection must not be allowed to ‘‘contaminate’’ the results of the next injection. Thus, in order to maximize the speed of an injection campaign, the validation infrastructure should restore the system to a fault-free state as quickly as possible.

Using SWIFI, the *hardware* of the system under test cannot be permanently affected. Hence, a reboot (or power reset) of a node removes any faulty state in volatile memory (CPU registers and memory). For a distributed system running on physical machines, issuing a reboot command from within the faulty node may not work because the node could be crashed or hung. Using VMs instead of physical machines enables ‘‘power-cycling’’ the nodes without the use of special hardware (e.g. Intelligent Platform Management Interface).

Power-resetting a VM is not sufficient to repair a faulty node because fault injections can cause a node to write erroneous data to disk. The validation infrastructure must isolate and remove such erroneous disk state between experiments. Furthermore, erroneous disk state may need to be saved for later analysis to assist in debugging the tested system. A possible way to meet these requirements is to move the VM's old disk image to a safe location and make a copy of a ‘‘golden’’ uncorrupted disk image for the VM's new disk image. Since VM disk images can be quite large (hundreds of MB), copying an entire disk image would introduce long delays between experiments.

To reduce the delay of restoring disk state, the validation infrastructure uses a union mount inside each VM [15]. For each unprivileged VM, the privileged VM hosts two disk images: a read-only image containing the unprivileged VM's root file system, and a read-write image for the union mount's branch file system. The union mount of the root and branch file systems is performed by the guest OS at boot time. Subsequently, all writes to the root file system are redirected by the guest OS to the branch file system. After halting an unprivileged VM, the Campaign Agent moves the image containing the branch file system to another directory for storage and later analysis, and a new empty image is copied in its place. The Campaign Agent then commands the VMM to power-reset the unprivileged VM.

The use of union mount involves guest OS support and is thus somewhat intrusive. However, the benefit gained by this intrusion is a significant reduction in the time spent restoring disk state before booting a VM. This is due to the fact that the size of the branch file system can be small relative to the size of the root file system. For example, with our case study system (Section 3), the size of the root file system of each node was 2.5GB while the size of the branch file system was limited to 50MB. Copying the 2.5GB image took up to one minute, while copying the 50MB image took at most one second.

2.4. Logging for Postmortem Analysis

A key goal of any fault injection campaign is to identify and correct flaws in the system's fault tolerance mechanisms. In support of this goal, the infrastructure must provide a mechanism for collecting detailed functional and timing information regarding the actions of the injector as well as the resulting actions of all the components of the system under test. As diagnosis of a flaw often requires focusing the information collected to particular components of the system, this mechanism must be configurable to easily include or exclude a variety of information sources.

To meet the above requirements, our validation infrastructure records timestamped information logged by three components: the location of fault injections, logged by the fault injector; information related to detection of and recovery from errors, logged by user-level processes running on the nodes (VMs) of the system under test; and, for each node of the system under test, usage of CPU cycles and memory of all the processes, as well as process creation and termination events, logged by a system *resource monitor* running as a user-level process on each node. The node *resource monitor* is useful for diagnosing problems such as hung processes and memory leaks.

Each VM runs a user-level *log client* process which receives logs from various processes on that node via named pipes and sends them over the network to the *log server* on a remote physical machine. The *log server* writes the logs to its local file system for later analysis.

There is a possibility that the *resource monitor* and logging activities on each node will impact the operation of the system (undesirable *intrusiveness*). With virtualization, less intrusive resource monitoring could be implemented from the VMM. This would require significant additional complexity in the VMM — an ability to access and parse internal data structures of the OS kernel of the VMs. Additionally, instead of using *log clients* to transmit logs to the *log server,* virtualization could be exploited for lower intrusion by using shared memory pages between each of the unprivileged VMs and the privileged VM. During the design and later experimentation with our infrastructure we determined that, for our purposes, the intrusiveness of the implemented mechanisms was negligible and did not justify the added complexity of the alternative implementations. Furthermore, the present resource monitoring and logging mechanisms are portable to a distributed system running on physical machines.

3. Validation of the Ghidrah CMM

As a case study, we used our validation infrastructure with *Ghidrah*, a fault-tolerant cluster manager, developed at UCLA [13, 14]. Cluster management middleware (CMM) performs functions that are critical to the operation of a cluster, including allocating resources to user tasks, scheduling tasks, reporting task status, and coordinating fault handling for tasks. Since failure of the CMM causes the entire cluster to fail, the CMM must be highly fault-tolerant. This section describes the use of our infrastructure to validate the fault tolerance mechanisms of the *Ghidrah* CMM.

To facilitate understanding of the fault injection campaigns we employed, we present a brief overview of *Ghidrah's* system architecture and its error detection and recovery mechanisms. Following this, we describe the fault injection campaigns and the results of fault injections.

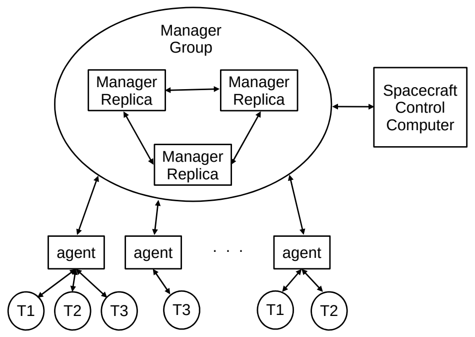

Figure 3: The logical structure of the *Ghidrah* fault-tolerant cluster management middleware.

3.1. Overview of *Ghidrah* System Architecture

With the goal of deployment for data processing in space, the *Ghidrah* CMM includes aggressive fault tolerance capabilities [13, 14]. The overall structure of *Ghidrah* is shown in Figure 3. The system consists of four components: a replicated centralized manager, an agent on each node, a library for user applications, and a ''trusted computer'' called the Spacecraft Control Computer (SCC). *Ghidrah* supports running multiple parallel applications with gang scheduling; in Figure 3 the circles represent processes of three user tasks labeled T1, T2, and T3. The *Manager Group* performs cluster-level decision making, such as scheduling and fault management. An agent on each node reports node status to the *Manager Group,* performs commands at the node on behalf of the *Manager Group,* and provides an interface between application processes and the CMM. The user-level library, linked with every user application, provides the mechanisms for setting up intra-task communication and includes an implementation of the Message Passing Interface (MPI) for user applications. The CMM design and implementation is focused on maintaining the basic cluster functionality despite any single faulty node. The minimum ''basic cluster functionality'' that must be maintained is the ability to submit new tasks, to continue the execution of tasks on operational nodes, and to maintain overall scheduling and monitoring of the system.

The most critical part of the *Ghidrah* CMM is the centralized manager. Hence, the manager employs active replication [13] across three processes (*Manager Replicas,* or simply *Replicas*) running on different nodes. Each manager operation is performed independently on each of the *Manager Replicas.* Messages exchanged among *Replicas,* and between *Replicas* and agents are authenticated (signed [11]) to ensure that faulty nodes cannot forge messages from other nodes, even if the message is forwarded by the faulty node.

Agents act only when receiving identical authenticated commands from at least two *Manager Replicas.* Hence, a *Replica* that stops or generates incorrect commands cannot corrupt the system. If any *Replica* or agent suspects an error in a *Replica,* a message is sent to all *Replicas* to initiate a *self-diagnosis procedure*. Self-diagnosis consists

Table 1: Descriptions of fault injection campaigns and their results. The campaign name indicates the injection target and fault type: the first letter is the target, **P** for user-level **p**rocess or **K** for OS **k**ernel; the second letter is the fault type: **f** for bit-**f**lip or **t** for process **t**ermination. *Target node selection changed only after the target node is power-reset. †Some flaws were exposed by multiple campaigns.

Campaign		Campaign Description			Campaign Results			
		Time slot (s)	Selection of target node	Injection target	Total time (hours)	# injections	# errors	# flaws exposed
General	1Pf	60	Round-robin	One randomly-selected CMM process	422.1	23781	4943	6
	2Kf	60	Random	OS kernel	92.6	4136	3000	2
	3Kf	15	Random*	OS kernel	76.6	7127	1458	0
Ghidrah-specific	4Pf	60	Round-robin	Two randomly-selected CMM processes	320.0	21403	13062	6
	5Pf	60	Round-robin	Two randomly-selected CMM processes	252.2	12574	12574	5
	6Kf	60	Node not running *Manager Replica*	OS kernel	24.9	794	489	1
Flaw-specific	7Pt	30	Random	*Manager Replica*	63.5	7023	7023	1
	8Pt	30	Random	*Agent*	30.3	1058	1058	1
Totals					1282.2	77896	43607	11†

of a Byzantine-fault-tolerant agreement protocol to detect and diagnose a *Replica* with corrupted state. If two *Replicas* agree that a third *Replica* is faulty, they run a *replica recovery procedure* to command the faulty *Replica's* local agent to terminate it and to command an agent on another node to spawn a new *Replica*.

Each "agent" actually consists of three processes, called *Agent*, *Agent-Helper*, and *Agent-Keeper*. The *Agent* communicates directly with the *Manager Group,* including sending periodic heartbeats. The *Agent-Helper* handles messages from application processes. The *Agent-Keeper* is a very small process that monitors heartbeats from the *Agent* and *Agent-Helper,* restarting both if the heartbeats stop.

The SCC controls the entire spacecraft, including handling communication between the cluster and its human operators on Earth. Loss of the SCC implies loss of the spacecraft. Hence, while the entire cluster is built using commercial-off-the-shelf technology, the SCC uses radiation-hard technology and other aggressive fault tolerance techniques to ensure the survival of the spacecraft. The design of the CMM must take into account the need to interact with the SCC and can take advantage of the existence of this "hard core." However, the SCC is not designed for high performance and must not be burdened with routine operation of the cluster.

Ghidrah takes advantage of the SCC by relying on its ability to power-reset the nodes of the cluster. The *Manager Group* commands the SCC to power-reset a node if it stops receiving heartbeats from the node's *Agent*. The *Manager Group* also sends periodic heartbeats to the SCC and a report each time the self-diagnosis procedure is initiated. If the SCC stops receiving consistent heartbeats from at least two *Manager Replicas*, it power-resets all of the cluster nodes (i.e. resets the entire cluster). It is possible that failure of the self-diagnosis procedure will cause *Replicas* to erroneously initiate new rounds of self-diagnosis repeatedly. The SCC maintains a record of recent self-diagnosis initiations and triggers a reset of the entire cluster if the number of self-diagnosis initiations over a specified period of time exceeds a fixed threshold.

3.2. Experimental Setup

We ran the Xen VMM on a dual-socket quad-core Intel Xeon system (a total of eight cores). A cluster of four nodes was run in the virtualized system. The SCC ran on a separate physical computer; this computer also used a log client to send SCC logs to the log server described in Subsection 2.4. Instead of directly performing power-resets of cluster nodes, the SCC logged such actions. The logs were then read and acted upon by the Campaign Agent running on the privileged VM.

3.3. Fault Injection Campaign Design

A *fault injection campaign* is a set of fault injections that have common properties, such as fault injection type, target, and timing. After an injection is performed, the system under test is allowed to run for a defined period of time to observe whether the injected fault caused an error, whether the error was detected by the system, and whether the system recovered from the detected error. The campaign may also direct when and how the system is to be restored to fault-free conditions in preparation for subsequent injections.

To validate *Ghidrah* and its fault tolerance mechanisms we designed three kinds of fault injection campaigns: 1) *general campaigns* that injected faults with minimal regard to the design of *Ghidrah*, 2) *Ghidrah-specific campaigns* that injected faults based on the design of *Ghidrah*, and 3) *flaw-specific campaigns* that injected faults designed to reproduce conditions known to activate flaws that were exposed by the first two kinds of campaigns. Table 1 summarizes all of the campaigns. The general campaigns were useful in exposing a number of flaws in the *Ghidrah* CMM. Because the *Ghidrah*-specific campaigns used injections specific to the design of *Ghidrah*, they were able to expose more flaws that were very unlikely to be exposed by the general campaigns. Many of these flaws were exposed only after tens of hours of fault injections.

For the campaigns involving bit flips, the injected fault was a single bit flip into a randomly selected general purpose register of the x86 architecture. The selection of the bit of the register to flip was also random. Single bit flips were used because they have been shown to best capture the effect of hardware transient faults caused by particle strikes to the system [5]. Since x86 processors have relatively few registers, each of these registers is frequently used. Hence, a large fraction of faults injected into registers are manifested as errors. Thus, fault injection into registers resulted in accelerated stressing of *Ghidrah's* fault tolerance mechanisms. Fault injection into memory was not performed because memory corruption has been shown to have a low error manifestation rate [12, 18].

The timing of fault injections was varied in order to introduce errors at different points of execution of the system. For each campaign, time was divided into periodic slots (see Table 1), with one injection performed per time slot. The injection trigger was scheduled to fire after a random interval from the beginning of the time slot up to half of the time slot duration. For injections of bit flips in registers that targeted a specific process or the OS kernel, the action for the time-based trigger was not the actual injection. Instead, the action was to set a second trigger to fire after a randomly-selected number of instructions, up to 5000 instructions, were executed by the target process or OS kernel. The second trigger's action was to perform the bit flip.

In all of the campaigns, the Campaign Agent read the SCC log so that it could power-reset nodes on behalf of the SCC. *Ghidrah* was designed to handle only a single node failure within the time it takes to detect and recover from such failure. Hence, as discussed in Subsection 2.2, the injection infrastructure had to refrain from additional injections while the node reset was in progress. Therefore, the Campaign Agent paused all fault injection during an SCC-requested node power-reset.

None of the campaigns prevented faulty state in the target node from being propagated to fault-free nodes (e.g. via a corrupt message sent to a fault-free node). However, *Ghidrah* was designed to detect and handle messages with corrupt payloads. Thus, it was not surprising that our experiments did not expose any fault propagation to fault-free nodes. Separate work in [14] performed fault injections on message payloads to validate *Ghidrah's* fault-tolerant communication protocols.

3.3.1. General Campaigns

There were two general campaigns: Campaign 1Pf used the OS-level injector to target CMM processes; Campaign 2Kf used the VMM-level injector to target the guest OS. As discussed in Subsection 2.3, a node would ideally be rebooted after each injection in order to ensure a fault-free state before the next injection. However, in order to accelerate the experiments, this was not done with these campaigns. This shortcut had the potential to produce incorrect results — failure of the system under test due to a latent error from a previous injection coupled with the current injection manifesting as two simultaneous node failures. In our runs of Campaigns 1Pf and 2Kf, this circumstance did not occur because faulty state either caused an immediate error or was overwritten before an error could be caused.

Campaign 3Kf was designed to decrease the possibility of simultaneous errors in multiple nodes while, at the same time, accelerating fault injections into the guest OS. In Campaign 3Kf, a target node was randomly selected, and it remained the target until it failed in such a way that the SCC requested it to be power-reset. Subsequently, a new target node was selected for the next injections. Thus, a single node could be targeted for multiple injections without clearing faulty state between injections, but at no time was there more than one node subject to fault injection.

3.3.2. *Ghidrah*-Specific Campaigns

These campaigns stressed *Ghidrah* based on knowledge of its design. Campaign 4Pf injected bit-flip faults as Campaign 1Pf, except that in each time slot an injection was performed in two randomly selected CMM processes on a single target node. Like Campaign 1Pf, targeted processes were not terminated after each fault injection experiment to remove faulty state. In some cases, this did cause a fault injected in one time slot to be manifested as an error in a later time slot. When this impacted two *Manager Replicas* simultaneously, *Ghidrah* responded correctly, with the SCC resetting the entire cluster.

Demonstrating the flexibility of the infrastructure, Campaign 5Pf was a modification of 4Pf that eliminated the incorrect propagation of the effects of one injection to the following injection, without requiring a time-consuming node reboot. 5Pf was similar to 4Pf except that the targeted processes were explicitly terminated near the end of the time slot. *Ghidrah* detected the terminated processes and automatically started new processes to replace them. Thus, the system was quickly restored to a fault-free state by the time the next time slot began.

Campaign 6Kf stressed *Ghidrah's* replica recovery procedure by attempting to cause the starting of a new *Replica* to fail. In this campaign, a bit-flip fault was injected into the OS of a node *not* running a *Replica*. This could crash or hang the node or potentially introduce latent erroneous state in the node. In the case where the target

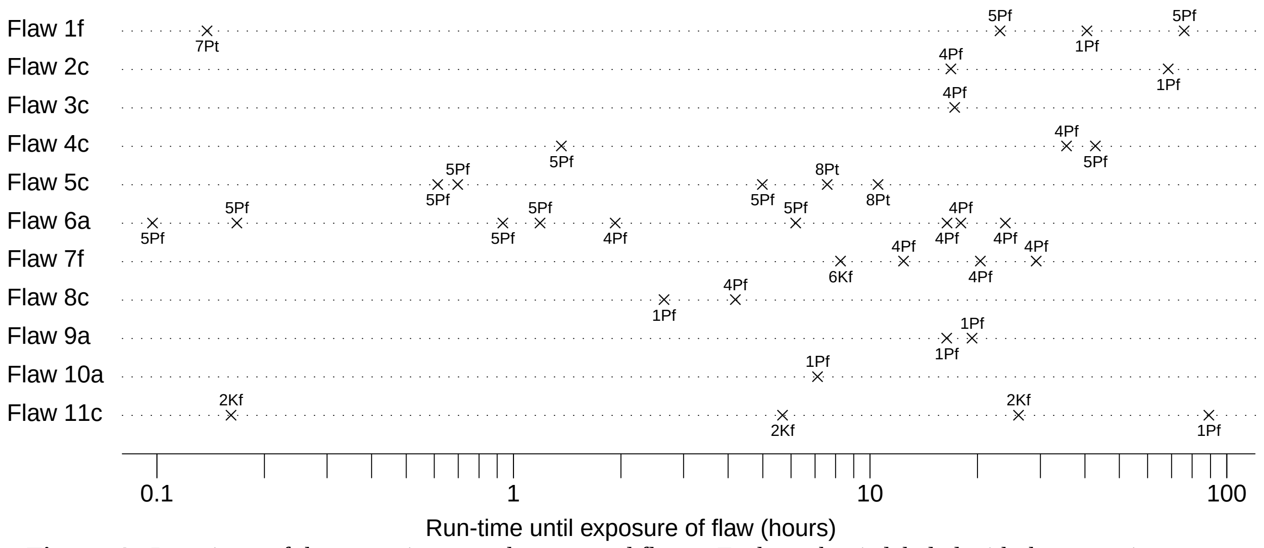

Figure 4: Run-times of the campaign runs that exposed flaws. Each marker is labeled with the campaign name.

node did not hang or crash in the 15 seconds following the injection, the campaign caused the system to attempt to move a *Replica* to the target node. This was done by terminating a *Replica* process on another randomly-selected node, forcing the *Manager Group* to attempt to start a new *Replica* on the target node. If the target node failed to start the *Replica*, the *Manager Group* was expected to attempt to start a new *Replica* on another node.

3.3.3. Flaw-Specific Campaigns

The general and *Ghidrah*-specific campaigns exposed a number of flaws that were activated by CMM processes crashing. Campaigns 7Pt and 8Pt were designed to quickly reproduce these flaws by explicitly terminating CMM processes. Campaign 7Pt's injection was to terminate the *Replica* on the target node; Campaign 8Pt's injection was to terminate the *Agent* and *Agent-Keeper*.

3.4. Results

The results of the campaign runs are summarized in the right half of Table 1. Of the 43,607 fault injections that caused errors, 37 exposed a total of 11 unique flaws in *Ghidrah*. To qualitatively illustrate the number of injections required to expose these flaws, we define the concept of a *campaign run* as a sequence of injections beginning with the cluster starting with all nodes powering up and ending with the exposure of the flaw. The 37 flaw exposures are plotted in Figure 4 against the duration of the campaign run in hours. The large variability in the campaign run times are due to the randomness of injections, demonstrating the necessity of many injections. The figure also illustrates the need for varied fault types in order to cover the various parts of the system.

Flaws 1f, 2c, and 3c resulted in race conditions between normal timer events in the *Manager Group* and local events occurring in individual *Replicas* during the self-diagnosis and replica recovery procedures. Therefore, the campaigns that caused a *Replica* to fail were more likely to expose these flaws. After these flaws were exposed by the *Ghidrah*-specific campaigns, Campaign 7Pt was designed to re-expose these flaws for debugging purposes. The first run of Campaign 7Pt ended with exposure of Flaw 1f in less than 15 minutes. The remaining 63 hours of Campaign 7Pt runs were performed after Flaws 1f, 2c, and 3c were fixed.

The *Ghidrah* implementation failed to take into account the fact that a halted *Agent* does not mean that all the other processes on that node have halted. Flaws 4c, 5c, and 6a resulted in *Ghidrah* processes improperly handling messages from a *Replica* on a node whose *Agent* and *Agent-Keeper* had terminated. With flaws 4c and 5c, this led to *Replica* crashes. Flaws 7f, 8c, 9a, and 10a resulted in errors in handling unexpected orderings of messages or events when CMM processes terminated or were restarted. It should be noted that these flaws were exposed by different sets of campaigns across a wide range of campaign run-times.

Flaw 11c was unique among the 11 flaws because it was exposed primarily by injection into the OS in Campaign 2Kf. Fault injection caused the *Replica* on the target node to stop sending heartbeats to the other *Replicas* but to otherwise operate normally. This caused the other *Replicas* to repeatedly initiate self-diagnosis due to the missing heartbeats and to never detect an inconsistency in the replicated state of the *Replicas*. This flaw demonstrated the utility of injections targeting the OS even when the goal is to expose flaws in user-level programs.

Of all the components of the validation infrastructure, the logging mechanism had potentially the highest intrusion and performance overhead, depending on how much data was logged by the tested system. For most campaign runs, we ran the *Ghidrah* CMM with detailed logging enabled to facilitate debugging. During these runs, the four nodes combined logged an average of 5.8KB/s.

The system logged more data while error detection and recovery procedures were being executed. Injections that caused errors resulted in bursts of logging activity, lasting up to 30 ms, with a peak rate of 58 KB/s.

4. Related Work

Fault injection has been used for validating fault tolerance mechanisms for over four decades. Many SWIFI tools target distributed systems [19, 7, 8, 6] but do not use virtualization. Such tools do not, by themselves, provide a solution to the problem of unattended operation of campaigns that require node reboots. In [6], faults injected into the distributed system are coordinated based on a partial view of the global state of the system. This approach requires instrumentation of the target system to notify the fault injector of the system's state transitions. DOCTOR [8] and NFTAPE [19] are flexible SWIFI tools that use multiple trigger types and fault types for injection into distributed systems.

Virtualization has been used by others for fault injection. This includes tools for single-node systems, where the Linux kernel and applications run inside a single user-level Linux process [2, 3, 18]. In [16], fault injection is performed using a software-implemented emulator of PC hardware. This tool is also focused on single-CPU systems.

In [9] an infrastructure using virtualization is used to evaluate a fault-tolerant overlay network for implementing distributed hash tables. The faults injected included terminating user-level processes and halting VMs. Their methodology and focus most closely resemble our work.

5. Conclusion

It is well understood that validating fault-tolerance properties of complex distributed systems is a difficult task because flaws may remain hidden until a rare confluence of asynchronous events occurs. The flexible validation infrastructure presented in this paper leverages both virtualization and SWIFI to efficiently validate complex distributed systems. It enables unattended, long duration, fault injection campaign runs, consisting of both randomized and tightly focused injections. The infrastructure provides the flexibility to easily design, implement, and execute multiple fault injection campaigns to quickly expose, reproduce, and diagnose flaws in the target system. The validation infrastructure was used in a case study to validate a Byzantine-fault-tolerant cluster manager. The fault injection campaigns exposed several flaws and were instrumental in improving the reliability of the cluster manager.

Acknowledgements

This work was supported, in part, by a grant from the Jet Propulsion Laboratory Director's Research and Development Fund, and by gifts from HP Laboratories and Xerox Foundation University Affairs Committee.

References

[1] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield, "Xen and the Art of Virtualization," *19th ACM Symp. on Operating Systems Principles*, Bolton Landing, NY, pp. 164-177 (October 2003).

[2] K. Buchacker and V. Sieh, "Framework for Testing the Fault-Tolerance of Systems Including OS and Network Aspects," *6th Int. Symp. on High-Assurance Systems Engineering*, Boca Raton, FL, pp. 95-105 (October 2001).

[3] K. Buchacker, M. D. Cin, H. J. Hoxer, V. Sieh, O. Tschache, and M. Waitz, "Hardware Fault Injection with UMLinux," *Int. Conf. on Dependable Systems and Networks, Fast Abstracts*, San Francisco, CA, p. 670 (June 2003).

[4] J. Carreira, H. Madeira, and J. G. Silva, "Xception: A Technique for the Experimental Evaluation of Dependability in Modern Computers," *IEEE Trans. on Software Engineering* **24**(2), pp. 125-136 (February 1998).

[5] H. Cha, E. M. Rudnick, J. H. Patel, R. K. Iyer, and G. S. Choi, "A Gate-Level Simulation Environment for Alpha-Particle-Induced Transient Faults," *IEEE Trans. on Computers* **45**(11), pp. 1248-1256 (November 1996).

[6] M. Cukier, R. Chandra, D. Henke, J. Pistole, and W. H. Sanders, "Fault Injection Based on the Partial Global State of a Distributed System," *18th Symp. on Reliable Distributed Systems*, Lausanne, Switzerland, pp. 168-177 (October 1999).

[7] S. Dawson, F. Jahanian, T. Mitton, and T.-L. Tung, "Testing of Fault-Tolerant and Real-Time Distributed Systems via Protocol Fault Injection," *26th Fault-Tolerant Computing Symposium*, Sendai, Japan, pp. 404-414 (June 1996).

[8] S. Han, K. G. Shin, and H. A. Rosenberg, "DOCTOR: An Integrated Software Fault Injection Environment for Distributed Real-time Systems," *Int. Computer Performance and Dependability Symp.*, Erlangen, Germany, pp. 204-213 (April 1995).

[9] T. Herault, T. Largillier, S. Peyronnet, B. Quetier, F. Cappello, and M. Jan, "High Accuracy Failure Injection in Parallel and Distributed Systems Using Virtualization," *6th ACM Conf. on Computing Frontiers*, Ischia, Italy, pp. 193-196 (May 2009).

[10] M.-C. Hsueh, T. K. Tsai, and R. K. Iyer, "Fault Injection Techniques and Tools," *IEEE Computer* **30**(4), pp. 75-82 (April 1997).

[11] L. Lamport, R. Shostak, and M. Pease, "The Byzantine Generals Problem," *ACM Trans. on Programming Languages and Systems* **4**(3), pp. 382-401 (July 1982).

[12] M. Le, A. Gallagher, and Y. Tamir, "Challenges and Opportunities with Fault Injection in Virtualized Systems," *1st Int. Workshop on Virtualization Performance: Analysis, Characterization, and Tools*, Austin, TX (April 2008).

[13] M. Li, D. Goldberg, W. Tao, and Y. Tamir, "Fault-Tolerant Cluster Management for Reliable High-Performance Computing," *Int. Conf. on Parallel and Distributed Computing and Systems*, Anaheim, CA, pp. 480-485 (August 2001).

[14] M. Li, W. Tao, D. Goldberg, I. Hsu, and Y. Tamir, "Design and Validation of Portable Communication Infrastructure for Fault-Tolerant Cluster Middleware," *IEEE Int. Conf. on Cluster Computing*, Chicago, IL, pp. 266-274 (September 2002).

[15] J. R. Okajima, "aufs—another unionfs," *http://aufs.sourceforge.net/aufs.html*.

[16] S. Potyra, V. Sieh, and M. D. Cin, "Evaluating Fault-Tolerant System Designs Using FAUmachine," *Workshop on Engineering Fault Tolerant Systems*, Dubrovnik, Croatia (September 2007).

[17] M. Rosenblum and T. Garfinkel, "Virtual Machine Monitors: Current Technology and Future Trends," *IEEE Computer* **38**(5), pp. 39-47 (May 2005).

[18] V. Sieh and K. Buchacker, "UMLinux - A Versatile SWIFI Tool," *4th European Dependable Computing Conf.*, Toulouse, France, pp. 159-171 (October 2002).

[19] D. T. Stott, B. Floering, D. Burke, Z. Kalbarcypk, and R. K. Iyer, "NFTAPE: A Framework for Assessing Dependability in Distributed Systems with Lightweight Fault Injectors," *4th Int. Computer Performance and Dependability Symp.*, Chicago, IL, pp. 91-100 (March 2000).