

Parikshan: A Testing Harness for In-Vivo Sandbox Testing

Nipun Arora
NEC Research Labs
Princeton, NJ, USA
nipun@nec-labs.com

Franjo Ivancic
Google
New York, NY, USA
ivancic@google.com

Gail Kaiser
Columbia University
New York, NY, USA
kaiser@cs.columbia.edu

ABSTRACT

One of the biggest problems faced by developers testing large scale systems is replicating the deployed environment to figure out errors. In recent years there has been a lot of work in record-and-replay systems which captures traces from live production systems, and replays them. However, most such record-replay systems have a high recording overhead and are still not practical to be used in production environments without paying a penalty in terms of overhead.

In this work we present a testing harness for production systems which allows the capabilities of running test-cases in a sandbox environment in the wild at any point in the execution of an integrated application. The paper leverages, User-Space Containers(OpenVZ/LXC) to launch test instances in a container cloned and migrated from a running instances of an application. The test-container provides a sandbox environment, for safe execution of test-cases provided by the users without disturbing the execution environment. Test cases are initiated using user-defined probe points which launch test-cases using the execution context of the probe point. Our sandboxes provide a separate namespace for the processes executing the test cases, replicate and copy inputs to the parent application, safely discard all outputs, and manage the file system such that existing and newly created file descriptors are safely managed.

We believe our tool provides a mechanism for practical testing of large scale multi-tier and cloud applications. In our evaluation provide a number of use-cases to show the utility of our tool.

Categories and Subject Descriptors

H.4 [Information Systems Applications]: Miscellaneous;
D.2.8 [Software Engineering]: Metrics—*complexity measures, performance measures*

General Terms

Theory

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

ICSE 2015 Firenze, Italy

Copyright 20XX ACM X-XXXXX-XX-X/XX/XX ...\$15.00.

Keywords

ACM proceedings, L^AT_EX, text tagging

1. INTRODUCTION

As application software grows and gets more complicated, testing large scale applications has become increasingly important. However, it is often impossible to recreate realistic workloads in an offline development environment for large scale multi-tier or cloud based applications. Testing in the development environment can be (1). Un-realistic because it may not be possible to faithfully reconstruct the production environment, (2). The test-cases generated may be incomplete, (3) It is infeasible to test all possible configurations given time and cost constraints of releasing the software to the field.

One of the proposed mechanisms of addressing this problem is to “perpetually test”[?] the application in the field after it has been deployed. This is important since testing in a production system enables us to capture previously “unreachable” system states, which are possible only for a long running production environment. Since we are testing in a real environment, we get real user-input for test-cases, and are able to test a real environment. However, perpetual testing has never gotten much traction because of an obvious flaw : executing such test-cases will adversely effect the user-experience, both in performance and potentially in application logic.

The reason behind any potential state-change, or performance slow-down is because of embedding test-case logic within the context of the application, thereby executing something that would otherwise not be there in an optimized implementation of the application. This is something which is usually unacceptable in user-facing applications. Intuitively, test-cases can be parallelized thereby amortizing the cost of running the test, by running it in parallel [?]. However such approaches suffer from slow-down problems, as the forking must happen within the production system environment, and can lead to slow-down, as well as effect the system state, and hence can only help in speeding up testing in a development environment. An alternate approach is to record running production environments, and replay them offline. Over the years there have been several systems which have explored this direction with promising results. However most record and replay systems have high overheads, and require replication of the running configuration which may not be possible. Additionally the administrator needs to wait for offline analysis, instead of doing real-time diagnosis.

The motivation behind our work is to provide testing

as a service for real-time diagnosis of production applications. We observe that most modern day service oriented applications are hosted on IAAS cloud providers, and can hence be easily scaled up. Leveraging this abundance of resource, and recent advances in user-space container technology (OpenVZ/LXC[1, ?]) we present a testing harness which allows the user to dynamically insert test cases in a production environment (we call this in-vivo testing), enabling real-time diagnosis. Our system called *Parikshan*¹ allows capturing the context of application, and for tests to be run without effecting the sanctity/and performance of the actual user-facing application. This is done by cloning a production server and creating two containers: a production container, and a testing container. We duplicate the incoming traffic to both the production container and the test container using a custom proxy, which ignores the responses from the test-container. The testing on the test-container is done on the fly using dynamic instrumentation, hence any set of test-cases can be turned on whenever required. The user can pre-define probe points for dynamically inserting test-cases (by default the entry and exit of each function is considered a probe point). Since the test is executed in a VM it acts like a sandbox which restricts it from causing any perturbation to the state of the parent process, or effecting the sanity of the responses to the production client. We synchronize the production and test containers using a variant of live migration without suspending the services of the production server, and follow it up with frequent synchronization for a long running tests. Frequent synchronization is necessary because the test container can potentially go out of sync with the production because of non-determinism or because a test-case changes the state of the container, and effects future problems.

@Nipun edit -> consider why use user-space containers instead of VMs? While VM virtualization has existed for several years, recent advances in user-space container technologies, along with support for migration, has created a space for light-weight testing in live environments. Technically our sandbox techniques could also be applied using more traditional Virtual Machines. However, the overhead of using Virtual Machines is considerably higher, and it would technically require double the amount of resources for the target production servers. User-Space containers reduce this overhead considerably by using the resources in the same machine. We believe the availability of resources in IAAS cloud infrastructures combined

The key contributions of this paper are:

- Our tool provides a sandbox environment to execute test cases in the production environment. This allows for a safe and secure test harness which does not effect the production state, and allows the application to proceed in it's execution.
- We allow for dynamic insertion of the test case, and safely capturing the context of the application. Dynamically inserting test-cases is important to avoid re-launching binaries in the test-container with the required test-cases. Restarting binaries is not possible, because it would break active network connections, and destroy the state of the test container

¹Parikshan is the sanskrit word for testing

- Language and Platform agnostic: One of the key advantages of our approach is that it is language and platform agnostic. Since the underlying mechanism takes advantage of containers as a platform to do the cloning, the language or interface does not matter as far as cloning is concerned. Of-course testing mechanisms may differ depending upon different languages.

1.1 Impact

The impact of sandbox testing can be seen in several different ways

- **Sandbox Live Testing** One of the key motivations leading to *Parikshan* is to provide a harness to allow the user to test real, live implementations. We have tried and tested this in particular for unit-testing cases. While unit-testing cases are undoubtedly self contained, and can be designed like assertions such that they have no after effect on the any subsequent executions, any failure in these test-cases can definitely result in an unclean state. Apart from touching the state of the system testing in the production will also lead to a slow-down, which will affect the real system. We believe *Parikshan* can be an effective tool to test and analyze unit tests.
- **Fault Tolerance Testing** A possible implementation of the *Parikshan* test harness is to do Fault Tolerance Testing. As mentioned earlier testing and recreating large-scale configurations is extremely difficult. Additionally testing scalable aspects is costly as a significantly large test-bed is required to replicate loads. Recent large scale fault tolerance testing approaches has been to use fault injection at random places. One such example is Chaos Monkey[?] which has been employed by Netflix [?] video streaming service. Netflix has a highly distributed architecture with a large client base, and has several robustness mechanisms inbuilt to manage for failure. The chaos monkey infrastructure forces random failiure in live Netflix production servers, to test it's fault tolerance. The key intuition behind this approach is, that faults in an ever evolving large-scale environment are inevitable, and in most cases the infrastructure will be able to auto-respond and get its instances back to a live state. However, in the cases when it is unable to do so, Netflix wants to learn from failures, by forcing them in scheduled low-traffic hours. Natrually such in production fault-injection mechanisms will always effect the user. An alternate mechanism proposed by *Parikshan* is to use the test-container to inject faults. As a clone of the production-container any fault-injected should produce a similar effect as the original container, without effecting the user.

- **Verification** Software Verification is the process of exploring all possible
- **Integration Testing**

2. MOTIVATION

2.1 Motivationg Scenario

Take for example user Joe who is an administrator, and IT manager for a multi-tiered system. Much like several

IT systems user Joe has a dashboard which informs him of the health status of all of his applications, and provides him with high level statistical views of all tiers of the system. At time t_0 , Joe observes an unusually high memory usage by tierA for transaction type X or unusually high latencies in fetch operations for user Y. Under usual circumstances, the system would have to go down (depending on the severity of the problem), a ticket would be generated for the developer and the system would be patched once the problem has been diagnosed. However often, it is difficult to find out the configuration of the system, and the user input which is causing this problem, also solving any emergent problems as soon as possible is extremely important.

Joe can now use *Parikshan*, to fork off a clone of tierA as test-tierA. Our proxy balancer sends a copy of the incoming request to test-tierA, while users can continue using tierA. Process in test-tierA follow the same execution paths, as they receive the same input (we discuss non-deterministic related issues later); this allows Joe to initiate deeper test-cases, and observe the test-tierA, without fearing any problems in the user-facing operations.

Time to bug resolution is usually a very important criteria in any user-facing service oriented application. Bearing this in my mind we believe, that online testing will be an important aspect towards modern applications. Additionally the usage of redundant computing for testing in alpha-beta testing (see 3) approaches is a well accepted paradigm in real-world applications. This leads us to believe that using redundant computing should be acceptable for regular testing approaches as well.

2.2 Motivation Questions?

To further motivate our testing paradigm we have come up with a set of motivating questions:

Q1: Is it important to sandbox test-cases?

Q2: Is recreating production environment difficult?

Q3: Is redundant computing available?

Q4: How would executing test-cases in a production server effect user-experience?

2.2.1 **Q1:** *Is it persistent testing important?*

2.2.2 **Q2:** *Is recreating production environment difficult?*

2.2.3 **Q3:** *Can redundant computing be utilized for testing?*

2.2.4 **Q4:** *How would executing test-cases in a production server effect user-experience?*

3. RELATED WORK

There have been several existing approaches that look into testing applications in the wild. The related work can be divided in several categories:

- **Perpetual Testing** We are inspired by the notion of perpetual testing[?] which advocates that software testing should be key part of the deployment phase and not just restricted to the development phase.
- **Record and Replay**
- **Alpha-Beta Testing**

4. DESIGN AND IMPLEMENTATION

In this section we begin with a system overview of *Parikshan*. We then explain how it inserts test cases, into the test harness, and finally we explain how a user can use the *Parikshan* api to insert test cases in the test harness.

4.1 System Overview

Figure 1 shows, a simple 2 tier production model with

4.1.1 How does clone manager work?

While the focus of our work is not to support VM/Container migration, or to make changes to the hypervisor, we need to tweak the way typical hypervisors offer live migration for our purposes. Before moving further we wish to clarify that instead of the typical live migration supported by hypervisors, *Parikshan* requires a clone of the production container. In contrast with live migration, where a container is copied to a target destination, and then the original container is destroyed, the cloning process requires both containers to be actively running, and be still attached to the original network. This cloning requires some tweaking, and modification in both how compute migration is handled, and especially how the network migration is handled.

To understand cloning in our context, one must first understand how live migration works. Live migration refers to the process of moving a running virtual machine, guest os or container from one host node (physical machine) to another, without disconnecting any client or process running within the machine. There are several variants of live migration, some of which require a short suspend time, while others are able to seamlessly transfer without any noticeable down-time by the user. In general the process involves the following steps: (1) Firstly, the copy is initiated by a pre-copy memory migration, where the underlying hypervisor copies all the memory pages from the source to the destination. (2) Secondly,

5. TRIGGERING AND INSERTING ANALYSIS

Once we have forked off a clone, we are now ready to do some deeper analysis. We divide such analysis in two parts based on the time window required for analysis: (1).Statistical

5.1 Statistical Analysis

Analysis which need a long time window to record, and run the status across multiple requests are considered as long running analysis. Such analysis can be considered to be similar to monitoring of live applications, and are usually statistical in nature. Typically tools such as PIN [?], Valgrind [?], Dyninst [?], can do deep analysis without modifying the logic of the application. However, they impose a heavy penalty in terms of performance. Such tools can be easily used in *Parikshan*, without effecting system performance. However, there are a few challenges with such statistics which need a longer window to run.

5.2 Unit Tests

6. NETWORK ISSUES

There are several problems that can effect the execution of sandbox testing.

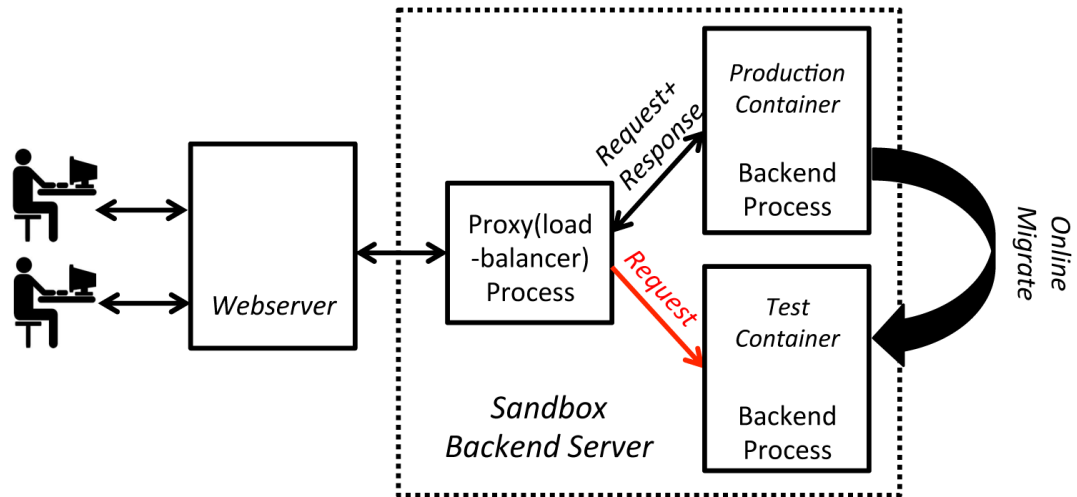


Figure 1: Workflow

- Stateful Connections
- Time Lag

7. IMPLEMENTATION

8. CONCLUSION

9. REFERENCES

- [1] K. Kolyshkin. Virtualization in linux. *White paper*, OpenVZ, 2006.