Parikshan: A Testing Harness for In-Vivo Sandbox Testing

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 levarages, User-Space Containers(OpenVZ/LXCs) 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 seperate namespace for the processes executing the test cases, replicate and copy inputs to the parent application, safetly discard all outputs, and manage the file system such that existing and newly created file descriptors are safetly 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

ICSE 2015 Firenze, Italy

Keywords

ACM proceedings, LaTeX, 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. Companies such as Flickr, Twitter, Facebook and Google have re-ignited the debate towards similar approaches as perpetual testing under the umbrella of DevOps. DevOps stresses on close coupling between software developers and operators, and to merge the operations of both. Most of these companies have very frequent releases and a very short time to a bug fix, test, patch and release in order to realize continuous delivery(Facebook mobile has 2 releases a day, and Flickr has 10 deployment cycles per day). Testing, analyzing and monitoring are an important aspect of the devops cycle, however they have several limitations as only a very small performance bandwidth can be dedicated to QA operations as compared to real user activity(so as to not effect user-perceived delay).

The motivation behind our work is to provide testing as a service for real-time diagnosis of production applications, in order to signifcantly reduce the time towards bug resolution. 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

[1]Parikshan is the sanskrit word for testing

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.

@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 availablity of resources in IAAS cloud infrastructures combined

The key contributions of this paper are:

- A tool which 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 safetly capturing the context of the application. Dynamically inserting test-cases is important to avoid relaunching 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
- 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 mechanims 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 definetely 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
- **Testing Software Updates** Software patches for performance or functional updates are frequently done on backend servers. These may not necessarily change the user-facing input and can be optimizations internal in the back-end server. Such patches can be first tested in the test-container to verify that they are correctly behaving before doing the release.

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 t0, 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 confiugration 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-determinisim 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:

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**
- **DevOps**

4. DESIGN

In this section we begin with an explanation of live cloning to explain to the reader some the key challenges in designing a sandbox. We then give a system overview of `Parikshan`, explaining each of its components. Next, we 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 How does cloning 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 fish 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) Once pre-copy phase is finished, the VM is temporarily suspended, and all memory pages including live memory pages are transferred to the target node. Once all memory pages have been transferred the target container is restarted. To reduce the down-time memory pages are transferred probabiliticly based on priority, and incase a page is not found a network fault is generated which executes a fetch from the original VM. Network migration is managed by the IAAS which publishes the same MAC address for the copied VM. Since the identity of the target container remains the same, the IAAS is able to give it same IP Address, and network traffic is rerouted after the identity is published in the network.

Instead of Live Migration, in Live Cloning, we do not suspend operations of the source container, rather we allow the container to keep executing in both production and test locations. The more tricky aspect is that there are two containers with the same identities in the network and application domain. Hence the same network identifier should map to two seperate addresses. Further a packet level sniffer or mirror port would keep the same buffer and potentially timeout. To allow for some load-balancing and an application aware buffer, we went for a web application level proxy server to duplicate traffic to both containers. Further in this section we explain how we deal with these challenges.

4.2 System Overview

Each instance of *Parikshan* can target only one tier at a time. Multiple instances of *Parikshan* can be orchestrated together especially when it's required for integration testing or cross tier results need to be correlated.

To begin with let us look at a simple example of client webserver with a database server as the backend, where the test harness needs to be applied on the backend. The architecture can be divided into two parts: (1) A Proxy Network Duplicator, (2) container clone manager

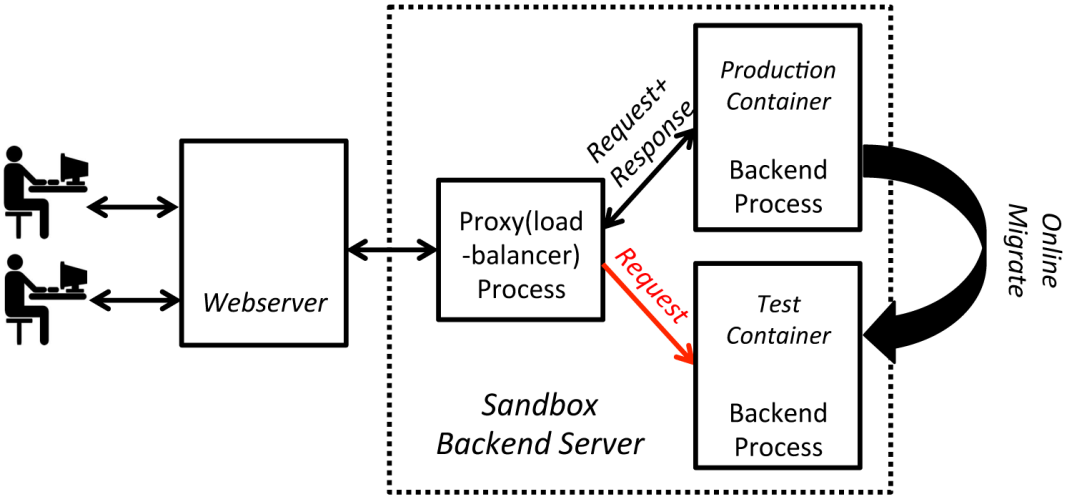

Figure 1: Backend wrapped around with Parakishan Run-time

4.2.1 *Proxy Network Duplicator*

As described earlier an important aspect of live cloning is that we have two replicas which share the same identity. Clearly two containers cannot share the same network identity. While we do not have strong consistency measures, in order for both containers to execute they must receive the same input. This can be achieved in multiple ways, the easiest would be a packet level tap device or a hardware switch which does port mirroring. These are both pretty common, and are blackbox and do not require much configuration. However, such port mirroring solution gives us minimal control on the traffic going to our test container. The production and test-container may execute at varying speeds which will result in them being slightly out of sync. Additionally we need to accept responses from both servers and drop all the traffic coming from the test-container, while still mantaining an active connection with the client. Hence a layer 2 level network solution is not possible as some context of the address and state are required

We have implemented our duplicator in two modes: (1) at TCP level, (2) at application level. The TCP level duplicator is configured with the client facing ip address (hence it becomes a a proxy), and essentially works as a socket reader/writer which reads incoming TCP streams and writes these streams to two different socket connections for the production and test containers.

4.2.2 *Clone Manager*

5. IMPLEMENTATION

Parikshan has been implemented in a small cluster environment, with physical hosts running KVM based virtual machines. Each VM is considered as a host-os for our implementation test bed. User-Space Container virtualization is done using OpenVZ[1], in Centos 6.4 kernels. Each container layout is managed using PLOOP[**?**] devices to enable faster and easier migration.

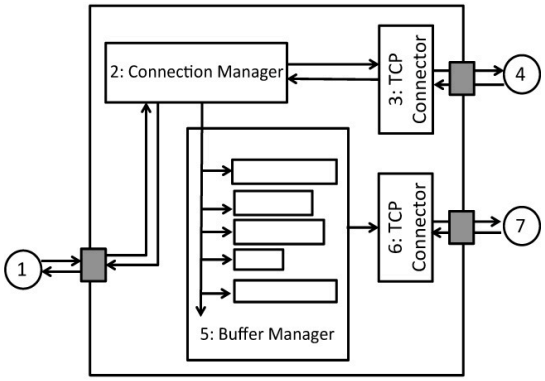

Figure 2: Description of the Network Duplicator

6. 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

6.1 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 challanges with such statistics which need a longer window to run.

6.2 Unit Tests

7. CHALLENGES

7.1 Non-Determinism

While the test container is a clone of production container, and they receive the same input simultaneously. They still suffer from aspects of non-determinism. Non-Determinism, can be triggered by multi-process servers, caching, or in rare circumstances it can happen because of non-determinism in the order of processing of input requests. While the change in system state between the production and test containers, may not be too important initially, we believe that the cumulative effect of non-determinism may change the

7.2 Slowdown

7.3 Consistency Requirements

8. NETWORK ISSUES

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

- **Stateful Connections**
- **Time Lag**

9. IMPLEMENTATION

Parikshan has been implemented in a small cluster environment, with physical hosts running KVM based virtual machines. Each VM is considered as a host-os for our implementation test bed. User-Space Container virtualization is done using OpenVZ[1], in Centos 6.4 kernels. Each container layout is managed using PLOOP[**?**] devices to enable faster and easier migration.

10. CONCLUSION

11. REFERENCES

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