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 us to run test-cases in a sandbox environment in the wild at any point in the execution of a service oriented 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 any perturbation to 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.

1 Introduction

As application software grows and gets more complicated, testing large scale applications has become increasingly important. The recent trend towards DevOps[**?**] by the software engineering industry further compounds this problem by requiring a fast and rapid resolution towards any software bug. 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 hence require 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).

However, it is extremely difficult to meet the quick debugging demands of a devops environment, as it is not feasible to recreate realistic workloads in an offline development environment for large scale multi-tier or cloud based applications. In general, testing in the development environment can be (1). Un-realistic because it may not be possible to faithfully reconstruct the production environment, (2). Incomplete, as it may be impossible to generate all possible input cases (3). Costly, as it is infeasible to test all possible configurations given time and cost constraints of releasing the software to the field. Hence testing is not only difficult because of difficulty to recreate production scenarios, it is also increasingly important to test and catch bugs in a very short period of time.

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 can arise due to various factors such as unpredictible user environment, outdated softwares, an ever increasing list of hardware devices (e.g. mobile phones, embedded devices etc.), or simply because of imperfect network connectivity (wifi, cellular). Some approaches such as Chaos Monkey[**?**] from Netflix, and AB Testing[**?**] already use "testing in the wild" to check for errors and robustness of the software, or to check for new features that have been added. However, despite a clear need, testing in the wild has never gotten much traction in real-world appli-

cations as it consumes too much performance bandwidth and more importantly, it can affect the sanity[1] of real operational state of the software.

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 resources, and recent advances in user-space virtualization technology(OpenVZ/LXC[**?**, **?**]) we present a testing mechanism 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`[2] 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.

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

- **Monitoring Applications/Localized Errors** Most user-end applications have monitoring mechanisms to capture the health of the application built within the system. Such monitoring mechanisms can often indicate problems in the systems showing spikes or slowdown in CPU usage, memory footprint, cache misses etc. Very often problems in stable production systems are either (1) restricted to only a small percentage of transactions, (2) are system wide, but have minimal effect on the user (cumulative effect of slow memory leak etc.). Such problems often do not necessitate taking down the system or lead to a crash *Parikshan* can be used to do a live analysis from the point of time when the clone is done, with much deeper monitoring, and light weight testing to localize the errors without worrying about the slowdown to the production system.
- **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.

[1]The state of the production server may change leading to a crash or wrong output

[2]Parikshan is the sanskrit word for testing

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.

- **Testing Software Updates - AB Testing** 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. A similar approach called A/B Testing[**?**] is commonly used in mobile and web-applications, which randomly forwards a small percentage of user traffic to a backend server B a modified version of the backend server(A is the original server). This gives the developer a controlled experiment scenario where he can see if the updates work without effecting too many users. *Parikshan* can potentially extend this by testing all user input since it does not effect the user experience at all.

- **Verification**

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 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-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 **??**) 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

4.1 System Overview

Each instance of *Parikshan* can target only one tier at a time. However, multiple instances 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 (as shown in Figure **??**), where the test harness needs to be applied on the

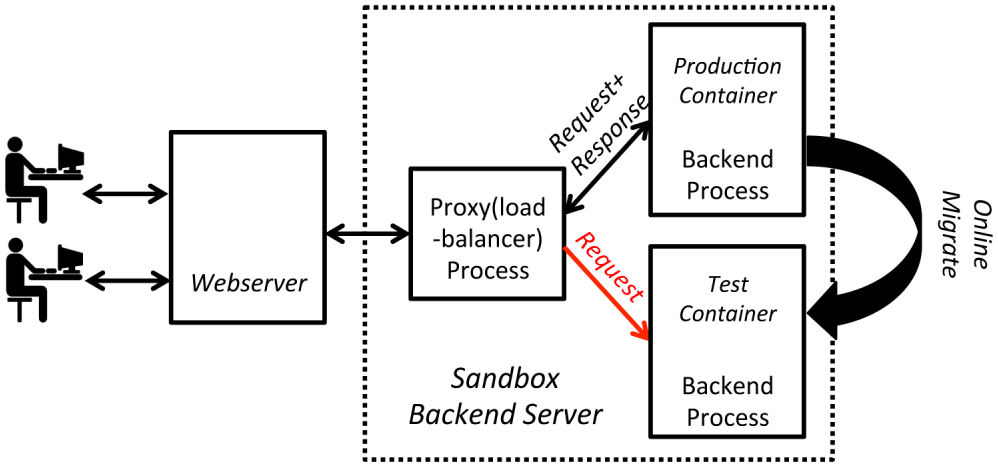

Figure 1: Backend wrapped around with Parakishan Run-time

backend. As explained earlier basic workflow of our system is to duplicate all network requests to the production backend server and a "live cloned" test container. Traffic duplication is managed by our proxy network duplicator (see section **??**), which uses several different strategies to clone user input to our test-container, with minimal impact on the production container. Another core aspect of our design is how to implement "live cloning"; this is the process by which a production container (in this case our backend service), can be cloned to create a test-container which has the same file system and process state. Cloning and syncing between the production container and the test-container is managed by our clone manager.

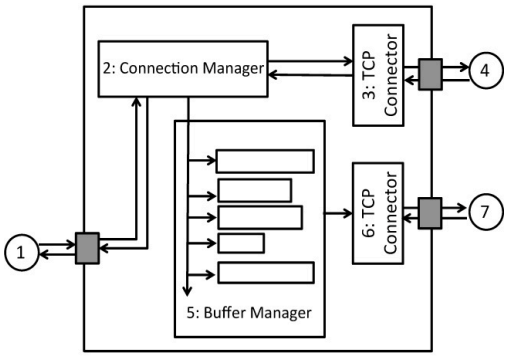

Figure 2: Description of the Network Duplicator

4.2 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 port-mirroring mechanism either using software provided tap devices or in hardware switches (several vendors provide mirroring options). These are both pretty common, and are black-box 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.

The workflow of each component is as follows: Traffic from the client (Node 1 in figure **??**) is forwarded to the Connection Manager(Node 2 in figure **??**). The connection manager essentially is a socket reader which copies, parses the incoming traffic. Based on the type of TCP request, a new connection is created or data is

forwarded/received to/from the TCP Connector(Node 3, figure **??**) which in turn creates a connection to actual production server. In this way the connection manager and TCP connector follow the TCP state machine hence mantaining network packet sanity while forwarding traffic to the production container. Simultaneously, the connection manager creates an internal copy of incoming traffic, and parses and sends it to the Buffer Manager (Node 5, figure **??**). In a naive scenario the connection could be simply forwarded to the test-container, and the traffic would be managed by itself. However, it is likely that the rate of traffic consumption by the test-container is less than the production container. This gives us several startegies to design the duplication of traffic to the test container:

1. **Synchronized**

   A naive startegy is to send TCP requests

2. **Asynchronous**

3. **Synchronized Partial Order Multi-Threaded Queue**

4. **Multi-Duplication**

Since the speed of input from the client is not in sync production-container, we need to buffer incoming traffic and relay it to the the TCP Connector(Node 6, figure **??**) as soon as the test-container is ready for new traffic. Since parallel connections can be initiated by the client on the same TCP port, the BufferManager creates multiple buffers for each connection, and initiates new connections by following the same causal flow of the packets received from the client. Unlike record-replay systems *Parikshan* does not claim to have strong consistency requirements, and does not need to exactly follow the production container as the goal is not reproduce all non-determinism in the production environment, but instead capture input non-determinism, and complete system state from a given point of time. We discuss consistency requirements further in section **??**

4.3 Clone Manager

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 standard 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.

The Clone Manager is responsible for creating a live running "clone" of the production container and launch it as the test-container. In our current setup cloning is done for each target production environment in the same physical host machine (we can clone to a different physical host as well, however for optimization purposes we have assumed that they will always be in the same local network). The same amount of resources as the production container are reserved for the test-container. After doing the pre-copy, we do a pause for syncing the two containers, and start them together. Subsequent clone

operations are optimized as they only require rsync for the change that has happened to the base image and in memory operation. After the intial setup, the frequency of cloning depends on the slowdown experienced by the test-container, and requirement of the test-case (some test cases may not require a long running test-window).

4.3.1 Time Delay and Buffer Management

Incoming traffic may potentially have a cumulative effect which will eventually lead to a buffer overflow in the BufferManager, hence leading to packet drops. This behavior is similar to packet dropping in classical TCP buffers, which rely on retransmission of incoming packets from the client. However in our case if packets are dropped the client cannot retransmit them as it not in sync with the test container. Hence in some cases *Parikshan* analysis, has to be restricted to a time-window dependent on the load of incoming traffic and the slowdown experienced by the test-container.

4.4 Network State Model

Network communication in most applications consists of two core types of protocols: UDP & TCP. The UDP(User Datagram Protocol) allows the applications to send messages(referred to as datagrams) to other hosts in the network without prior communications to set up any transmission channels. UDP uses a simple communication mechanism while minimizing protocls. It has no handshaking dialogues or acknowledgement of package delivery. It is broadly used for network traffic where speed is much more important than reliability (viz. network streaming applications like video etc.) On the other hand TCP(Transmission Control Protocol) is a reliable error checked delivery stream. It involves initially establishing the connection, and allowing for packet re-delivery or re-ordering to allow for reliable and dependable connection. While TCP is slower than UDP it is preferred for most normal connections between clients and server applications.

Since the UDP protocol has no error management mechanism, it automatically follows that machine state in a UDP connection is not important. Hence in our design the duplicator can easily flood packets to the cloned UDP server by simply sending packets to the targeted host and port. Our solution for this is

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 challanges 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.

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

7 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[**?**], in Centos 6.4 kernels. Each container layout is managed using PLOOP[**?**] devices to enable faster and easier migration.

8 Conclusion