Parikshan: Debugging Production Systems in Isolation

Anonymous Submission

ABSTRACT

One of the biggest problems faced by developers debugging 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 user-experience.

In this work we present a harness for production systems which allows users to debug the target system (run test-cases, debug, or profile etc.) in a sandbox environment cloned from a live running system at any point in it's execution. The paper leverages, user-space containers (OpenVZ/ LXCs) to launch a container cloned and migrated from a running instances of an application, thereby launching two containers: production (which provides the real output), test-container (for debugging/testing). This *test-container* provides a sandbox environment, for safe execution of test-cases/debugging done by the users without any perturbation to the execution environment. Our sandboxes provide a have namespace, and resource management for the processes executing the test/debug cases. A customized-network proxy agent replicates inputs from clients to both the production and test-container, as well safely discard all outputs from the test-container. We believe our tool provides a mechanism for practical live-debugging of large scale multi-tier and cloud applications, without requiring any application down-time, and minimal performance impact.

Categories and Subject Descriptors

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

General Terms

Theory

Keywords

ACM proceedings, LaTeX, text tagging

1. INTRODUCTION

As application software grows and gets more complicated, testing large scale applications has become increasingly important. The recent trend towards DevOps[8] 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, debugging in the development environment can be (1). Unrealistic 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 unfeasible to test all possible configurations given time and cost constraints of releasing the software to the field. Hence debugging is not only difficult because of difficulty to recreate production scenarios, it is also increasingly important to localize and fix 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 unpredictable user environment, outdated software, 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[16] 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 applications as it consumes too much performance bandwidth and more importantly, it can affect the sanity[1] of real operational state of the software.

The main reason that debugging in the development environment is easier, is because developers can trace the execution flow of the program using tools such as gdb, valgrind etc. and look at variable values for the given input. This gives them an immediate insight as to whether the application is behaving correctly, and where the bug could be. Unfortu-

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

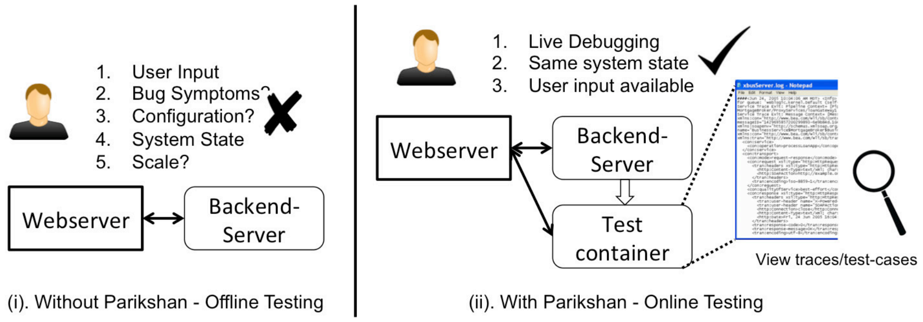

Figure 1: Here (i). shows a workflow of a simple 2 tier system without `Parikshan` where the user has to do offline debugging, and will have to figure out someway to capture the user input/bug symptoms/system state, (ii) is the same system running with `Parikshan` , which allows the user to do online debugging in a parallel test-container cloned from the backend server, which receives the same input

nately, such techniques are not possible in the production environment as they would lead to unacceptable slow-down, alter the application functionality, or worse crash the application. The motivation behind our work is to provide **"bug diagnosis as a service"** for real-time debugging of production applications in order to significantly 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[5, 1]) we present a debugging mechanism which allows the user to dynamically insert probes in a *cloned* production environment wotjpit effecting the actual application: thereby, enabling real-time diagnosis.

Our system called `Parikshan` [2] allows capturing the context of application, 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 debugging on the test-container is done on the fly using dynamic instrumentation, hence any set of test-cases can be turned on whenever required. 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 clone 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 long running tests.

While we discuss several case-studies to debug/test production applications that show how our framework can be used, we wish to stress that **the main advantage of `Parikshan` is a harness/framework for testing/debugging in a live environment rather than a new testing methedology**. The key contributions of this paper are:

- A tool which provides a sandbox environment to debug the production environment. This allows for a safe and secure mechanism to perturb and dianose the application, **without effecting the functionality** of the production container.

- A testing harness, and proxy which ensures non-blocking message forwarding to the test-container, which ensures **no performance impact** on the production container.

- Our tool tracks the **fidelity of the test-container** (if the test-container faithfully represents the production container) and creates a flag whenever the containers are out-of-sync. The time till which the test-container maintains fidelity is called it's *testing-window*(see section 2.4.

- We allow for **dynamic insertion of probes**, and safely capturing the execution trace of the application. Dynamically inserting probes is important to avoid relaunching binaries in the test-container. Restarting binaries would break active network connections, and destroy the in memory state of the test container(note: configuration/file-system state is still preserved).

- 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.

1.1 Motivationg Scenario

In Figure 1, we have shown two workflows of the same system running with `Parikshan` , and without `Parikshan` . To further explain, let us take 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 (Alternatively, a trouble ticket could have been generated by the user). Under usual circumstances, the system would have to go down(depending on the severity of the problem), the problem debugged using offline testing, and the system would be patched once the problem has been diagnosed. However often, it is difficult

[2]Parikshan is the Sanskrit word for testing

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.

One of the key advantages of such an online approach is a reduced time to bug resolution. Time to bug resolution is usually a very important criteria in any user-facing service oriented application, as the longer a bug remains the system, the more it is going to hit the user perception/revenue. 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 A/B testing(see section 9) 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.

1.2 Impact

The main advantage of Parikshan is that it provides an open platform which allows the user to do any kind of debugging/testing on a live clone of the production system which is also receiving the user input. Natrually, this allows the user to run 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[16] which has been employed by Netflix [3] 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.

- **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.

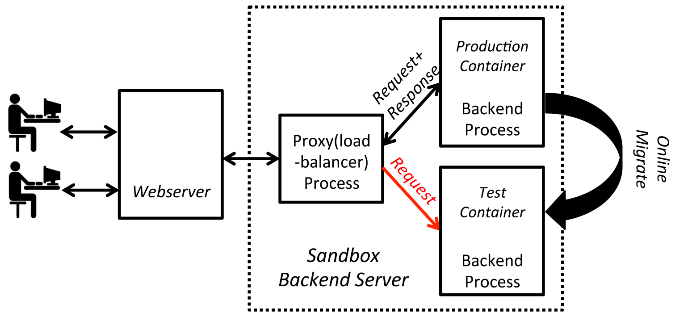

Figure 2: Backend wrapped around with Parakishan Run-time

2. DESIGN

2.1 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 web-server with a database server as the backend (as shown in Figure 2), where the test harness needs to be applied on the 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 2.3), 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, which uses user-space containers OpenVZ and a variant of live migration to manage the cloning. Next, we explain each of the modules in detail.

2.2 Clone Manager

2.2.1 *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 wish to clarify that instead of the standard live migration supported by hypervisors, `Parikshan` requires a cloning functionality. 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, let us 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. Live migration is supported by most well known Hypervisors(vmware, virtualbox, xen, qemu, kvm) with different amount of efficiency. There are several variants of 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 (*rsync*) 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. Obviously, two rsync runs are needed, so the first one moves most of the data, while the container is still up and running, and the second one moves the changes made during the time period between the first rsync run and the VM stop [3].

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. This is important, as the operating system and the process may be configured with the IP Address, or other networking operation, which cannot be changed without leading to a crash of the system. Hence the same network identifier should map to two separate addresses. Clearly the same ipAddress, mac addresses cannot be kept for both the production and test-container as that would lead to conflict in the network. There are two ways to resolve this : (1) Host each container behind their own network namespaces, on the same host machine, and configure packet forwarding to both containers such that the duplicator can communicate to them. Network namespaces(see internal mode, figure.3) is a definite possibility, and have been used by several hosting providers, to launch VM's with same private ipAddress, in a shared network domain [4]. (2) Another approach is to host both containers in different machines with port forwarding setup to forward incoming TCP requests to containers behind a NAT (see external mode, figure.3). This is slightly more scalable and has clear separation of network and compute resources. However, an obvious downside to this approach is that it needs a new VM to be allocated. This leads us to two different modes of implementation, which we discuss in the next section.

[3]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

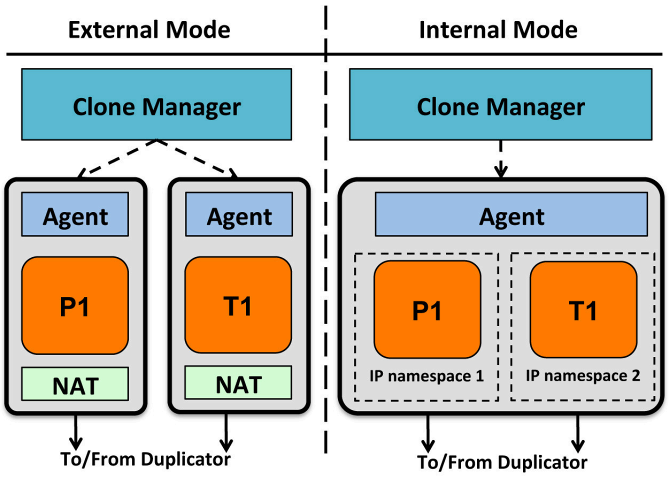

Figure 3: External and Internal Mode for Live Cloning: P1 is the production container, and T1 is the test container, the Clone Manager interacts with an Agent which has drivers to implement live Cloning

2.2.2 *Basic Design & Modes*

The Clone Manager itself is just an interface which interacts with an *agent* installed in each container host. The clone manager dictates the frequency of cloning/syncing operations, as well as coordinates setup operations/orchestration etc. The agent itself is the driver for live cloning, and performs rsync operations, snapshot, transferring and starting the image.

Test and production containers can be allocated in various schemes: we call these schemes *modes*. Broadly speaking the clone manager works in 2 modes (see figure.3) :

- *Internal Mode*: In this mode we allocate the test-container and production containers to the same host node. This would mean less suspend time, as the production container can be locally cloned (instead of streaming over the network). Requires the same amount of resources as the original production container (number of host nodes remain the same), hence could potentially be cost-effective. On the down-side, co-hosting the test and production containers, could have an adverse effect on the performance of the production container, hence effecting the user. As we can see in figure.3 the production container P1 and test container T1 are both hosted within the same physical host, and with the same ip addres, but their network is encapsulated within different network namespaces to sandbox them. The duplicator is then able to communicate to both these containers with no networking conflict.

- *External Mode*: In this mode we provision an extra VM as the host of our test-container (this VM can host more than one test-containers), While this mechanism can have a higher overhead in terms of suspend time (3 seconds, dependent on process state), and it will require provisioning an extra host-node, the advantage of this mechanism is that once cloned, the VM is totally separate and will not effect the performance of the production-container. We believe that such a mode will be more beneficial in comparison to the internal mode, as testing is likely to be transient, and it is often more important to not effect user experience[4]. As we can see in figure.3 the production container P1 and test container T1, are hosted on two different host machines, and are encapsulated behind a NAT[2] (network address translator), hence they each have their ip's in an internal network thereby avoiding any conflict.

2.2.3 *Algorithm*

In our current implementation, we are using OpenVZ[5] as our container engine, and have modified the migration mechanism in vzctl [7] to make it work for live cloning instead. We tested this out on multiple VM's acting as host nodes for OpenVZ containers. To make the cloning easier and faster, we used OpenVZ's *ploop* devices [6] to host the containers. *Ploop* devices are a variant of disk loopback devices where the entire file system of the container is stored as a single file. This makes features such as syncing, moving, snapshots, backups and easy separation of inodes of each container file system.

The algorithm for live cloning is explained below:

Let us imaging we are cloning production container C1 on Host H1 as test container C2 on Host H2. The initial setup requires certain safety checks and pre-setup to ensure easy cloning, these include: ssh-copy-id operation for accessing without password, checking pre-existing container ID's, check version of vzctl etc. These ensure that H1 and H2 are compatible, and ready for live-cloning. Next, we run an

[4]*Scaled Mode*: This can be viewed as a variant of the External Mode, where we can scale out test-containers to several testing containers which can be used to do statistical testing and distribute the instrumentation load to capture the overhead easier. This reduces the frequency with which the container needs to be synced. This is currently out of the scope of this paper, however we aim to show this in a future publication.

Algorithm 1 Algorithm for Live Cloning using OpenVZ

```
1. Safety Checks(Checks that a destination server is
   available via ssh w/o entering a password, and ver-
   sion checking of OpenVZ running in it)
2. Runs rsync of container file system (ploop device) to
   the destination server
3. Checkpoints and suspend the container
4. Runs a second rsync of the ploop device to the des-
   tination
5. Start container locally
6. Set up port forwarding and packet duplication
7. Starts the container on the destination
```

intial rsync of container C1, from Host H1 to Host H2, this step does not require any suspension of C1, and copies the bulk of the container file system to the destination server (H2). The next step involves checkpointing, and dumping the process in memory state to a file, this is what allows the container to be restarted from the same checkpointed state. However for sanity of the container process, it is important to restart the container from the same file system state as it was when the checkpoint was taken. To ensure this, we take a second rsync of the ploop device of C1, and sync it with H2, after this the original container can be restarted. Next we copy the dump file from step 3, from H1 to H2, and resume a new container from that dump file.

The overhead of cloning depends on the I/O operations happening within the container between step 2 and step 4 (the first and the second rsync), as this will increase the number of dirty pages in the memory, which in turn will impact the amount of memory that needs to be copied during the suspend phase (as mostly the dirty bits at suspend time are those which were not committed to memory and hence need to be transferred). A few iteration of cloning a container back and forth between two OpenVZ instances (on KVM's within the same physical machine), resulted in an average suspend time of 1.8 seconds for the production container, and 3.3 seconds for launch of the test-container This is nearly the same as that of native live migration[**?**], and has lesser suspend time for the production container as we do not include the *"copy dump file"*, or the *"undump and resume phase"* for production containers. In section 7.1, we evaluate the performance of live cloning while doing increasing amount of random I/O write operations, as well as with doing page fetches from webserver running the production container.

2.3 Proxy Network Duplicator

As described earlier an important aspect of live cloning is that we have two replicas which share the same identity. While we do not have strong consistency requirements for our production and test-container, in order for sandbox-testing to work, both containers 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 blackbox and do not require much configuration. However, such port mirroring solution gives us minimal control on the

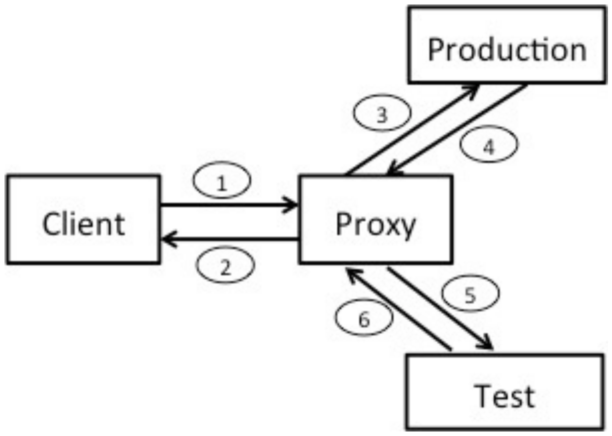

Figure 4: Description of the Network Duplicator. In *synchronized* mode: there are 2 threads for each connection, Thread 1 executes steps [1,3,5], and Thread 2 executes [2,4,6] sequentially, hence the speed of packets sent to the production and the test container are synchronized. In *asynchronous* mode, there are 4 threads for each connection, Thread 1 executes steps [1,3], Thread 2 executes [2,4], Thread 3 executes [5], and Thread 4 executes [6]. Hence communication to the test container and production container are asynchronous

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 maintaining 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

Network proxies can be created at different levels in the network stack, for our purposes we have created a TCP proxy which mirrors the incoming traffic. The TCP level duplicator is configured with the client facing ip address (hence it becomes 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.

In a naive scenario the connection could be simply forwarded to the test-container. However, it is likely that the rate of traffic consumption by the test-container is less than the production container. This means that the buffer size of the proxy for the test-container, the incoming client traffic and the amount of workload running on the test container will define the time window for which the test container will remain "in-sync" with the production container.

In this section we first discuss several strategies(we call modes) to design the duplication of traffic to the test container, and next we discuss briefly the "testing window" during which we believe that the test-container faithfully represents the execution of the production container.

2.3.1 Duplication Modes

1. **Synchronized Packet Forwarding Mode**

Algorithm 2 Network Duplication Algorithm

```
prod_port: The port at which the production server is running
test_port: The port at which the test container is running
listen_port: The port at which the proxy is listening
mode: 1 -> synchronous packet forwarding, 2 -> asynchronous packet forwarding, 3 -> asynchronous load balanced packet forwarding

function MAIN(prod_port, test_port, listen_port)
    spawn_worker_threads()
    bind(listen_port)
    listen(listen_port).on_event(mode,prod_port,test_port)

function ON_EVENT(mode, prod_port, test_port)
    if mode == 1 then
        accept()
        buffer=read_input()
        communicate_production (buffer, prod_port)
        communicate_test (buffer, test_port)
    if mode == 2 then
        accept()
        buffer=read_input()
        send_to_worker_thread( communicate_production (buffer, prod_port))
        send_to_worker_thread( communicate_test (buffer, test_port))
    if mode == 3 then
        accept()
        buffer=read_input()
        send_to_worker_thread( communicate_production (buffer, prod_port))
        send_to_worker_thread( communicate_test (buffer, test_port))

function COMMUNICATE_PRODUCTION(buffer, prod_port)
    connect(prod_port)
    sendall(buffer)
    sendToClient(recv())

function COMMUNICATE_TEST(buffer, test_port)
    connect(test_port)
    sendall(buffer)
    recv()
```

A naive strategy is to have a single worker thread to send and receive tcp stream to the production container as well as the test container. This is the simplest strategy and is quite robust as far as sending proxy data is concerned. To understand this better let us look at figure 4: Here each incoming connection would be handled by 2 parallel threads in the proxy T1, and T2. Where T1 sends data from the client to the proxy (communication link 1), then sends data from proxy to production (link 3), and finally from proxy to test container (link 5). Whereas, thread 2 sends replies from production to proxy(link 4), then receives replies from test to proxy (link 6), which are then dropped. Thread T2 then forwards Packets received on link 4 are forwarded on link 2 to the client.

By design TCP is a connection oriented protocol and is designed for stateful delivery and acknowledgement that each packet has been delivered. Packet sending and receiving are blocking operations by default, and hence if either the sender or the receiver is faster than the other the send/receive operations are automatically blocked or throttled.

In our case this can be viewed as follows: Let us assume that the client was sending packets at $XMbps$ (link 1), and the production container was receiving/processing packets at $YMbps$ (link 3), where $Y < X$. Then automatically, the speed of link 1 and link 2 would be throttled to $YMbps$ per second, i.e the packet sending at the client would be throttled to accommodate the production server. This behavior adheres to the default TCP protocol. However, in the synchronized mode we also send packets to the test-container (link 5) in the same sequential thread T1. Hence if the speed of link 5 is $Z\ Mbps$, where $Z < Y$, and $Z < X$, then the speed of link 1, and link 3 would also be throttled to $Z\ Mbps$.

Such a communication model effects the user-experienced delay for the targeted SOA application, and is against the guiding principal of `Parikshan` . To avoid the test-container to effect the communication between the client and the production server, we propose an asynchronous packet forwarding model discussed next.

2. **Asynchronous Packet Forwarding Model**

   As shown in the previous section, in the synchronized mode the test container, can effect the speed of the production container as well. The main reason for this is because of blocking sends being used to forward packets from the client to the test-container and production container by the same sequential process. In the asynchronous packet forwarding mode (see figure 4): we use 4 threads T1, T2, T3, T4 to manage each incoming connection to the proxy. Thread T1 forwards packets from client to proxy (link 1), and from proxy to production container (link 2). It then uses a non-blocking send to forward packets to an internal pipe buffer shared between thread T1, and thread T3. Thread T3, then reads from this piped buffer and sends traffic forward to the test-container. Similarly Thread T2, receives packets from production container, and forwards them to the client, while Thread T4, receives packets from the test-container and drops them.

   The advantage of this strategy is that any slowdown in the test-container will not effect the production container's communication as a non-blocking send is used to forward traffic to the production container. A side-effect of this strategy is that if the speed of the test-container is too slow compared to the production container, it may eventually lead to a buffer overflow. The time taken by a connection before which it overflows is called it's *testing-window*. We discuss the implications of the *testing window* in section 2.4.

3. **Asynchronous Load Balanced Packet Forwarding Model**

   It is still possible that there will be slowdown, and a short packet window because of the overhead of running test-cases in the test container. This would mean that the test container will have a short time-window to execute test-cases. To further increase this time window, we try and load balance testing across multiple test-containers, which can each get a duplicate copy of the incoming data. This would mean that there are multiple threads handling the incoming connection, one for the production container, and one for each of the test containers. We believe that such a strategy would significantly increase the testing window size especially if the testing load is heavy.

The algorithm for each of the packet forwarding modes has been described in Algorithm.2.

2.4 Testing Window

At the time of the live cloning, the testing container has an identical status and receives the same input as the production container. Hence, any test-cases run in the testing container, should faithfully represent the status of the production container. However, an obvious down-side to any debugging/testing is that it will add an overhead on the performance of the test-container as compared to the production environment. To avoid this slowdown impacting the actual production container, we discussed an asynchronous forwarding strategy in section **??**, where an unblocking send forwards traffic to a seperate thread which manages communication to the test-container. The thread has an internal buffer where all incoming requests are queued, the incoming request rate is dependent on how fast the production container manages the requests, where as the outgoing rate from the buffer is dependent on how fast the test-container processes the requests. Depending on the workload, and the overhead induced in the test-container, can eventually lead to a buffer overflow. The time period till buffer overflow happens is called the testing-window. For the duration of the testing-window, the test-container faithfully represents the production container as the testing-window of the test container. Once the buffer has overflown, the production container must be cloned again to ensure it has the same state.

In this section we try to model the testing window by using concepts well used in queuing theory(for the sake of brevity we will avoid going into too much detail, readers can find more about queueing theory models in [**?**]). The buffer overflow of our test-container can be modelled as a M/M/c/K queue (Kendall's notation[**?**]), for our simplest asynchronous model, and as a M/M/c/K queue in the asynchoronous loadbalanced model. An M/M/1/K queue, denotes a queue where requests arrive according to a poisson process with rate λ, that is the interarrival times are independent, exponentially distributed random variables with parameter λ . The service times are also assumed to be independent and exponentially distributed with parameter μ. Furthermore, all the involved random variables are supposed to be independent of each other. Further, the notation specifies that there are c queues/ or alternatively c servers managing the requests, and the queues is of a finite capacity, i.e. the queue can accommodate a maximum of K requests. In our case λ denotes the rate at which requests arrive to the buffer from the production container, and μ denotes the processing time overhead of each request in the test-container (to simplify the problem, we ignore the actual processing time of the test-container, as the incoming rate λ is already synchronized with the production container processing time, and it is only the overhead added by the test-container which matters). In our simple asynchronous forwarding strategy, we have $c = 1$ as we have only a single test-container, potentially as explained earlier, in a load-balanced asynchronous model this could be extended to c servers to increase the time window.

Now based on this notation the expected size of the time window is :

3. IMPLEMENTATION

`Parikshan` is built on top of a production user-space container virtualization software, OpenVZ [5], with Centos 6.5 with Linux Kernel 2.6.32. Each container layout is managed using PLOOP[6] devices to enable faster and easier cloning. The OpenVZ functionality, was extended to enable live-cloning as explained in section2. We modified and used the OpenVZ toolkit vzctl version 4.7 to create our cloning, creation and destroy scripts for the container. While the technology has also been tested on Debian systems, the evaluation in this paper has been done on RHEL(Centos) Systems. For our evaluation, we have not put in resource restriction on the containers (i.e. the containers have access to the same hardware resources as the host machine). Users using `Parikshan` may put resource restriction as required.

We have tested the system in 3 different configurations. In the first case we tested `Parikshan` 's internal mode configuration by installing `Parikshan` in a single host OS, with Intel Core 7 CPU, 8 Cores, 16GB RAM, and running Ubuntu 14.04. `Parikshan` was installed on multiple VM's running on the host OS using KVM based virtualization. Containers were cloned across these machines, with a seperate VM acting as the client. We used NAT, and ip namespaces for network access to the VM's.

In the second mode, we tested our system's external mode by installing `Parikshan` on the base kernel in identical host nodes. Each of these host nodes have an Intel Core 2 Duo Processor, 8GB of RAM, and ran Centos 6.5 with Linux Kernel 2.6.32.

In the third mode, we tested our system on Google's Cloud Infrastructure (Google Compute [?]). The production and the test container's were run on different virtual nodes, with 2 VCPU's and 4G RAM. The main advantage of using the Google Compute Engine was to run our cloning scripts on real data-centers, and also to scale out our evaluation.

The network proxy was implemented in C/C++. The forwarding in the proxy is done by forking off multiple processes each handling one send/or receive connection in a loop. Data from processes handling communication with the production container, is transferred to those handling communication with the test containers using pipes.

4. DISCUSSION AND CHALLENGES

In this section we describe some of the challenges, and limitations

4.1 Non-Determinism

Execution of the same input in different conditions can produce different outputs from an application, this behavior is referred to as non-deterministic execution. With the exception of dedicated controlled executions in embedded systems etc., most large scale applications and complex systems exhibit non-determinism, which is what makes debugging a challenge. Broadly non-determinism can be caused because of the following reasons: 1. Input non-determinism, 2. Configuration non-determinism, 3. Concurrency based non-determinism.

Input non-determinism is non-deterministic behavior caused because of different input received by the application (eg. requests to the server, user input etc.). Obviously, any output in the application and it's execution trace, will depend on the input it receives. In offline debugging it is often difficult to capture all possible inputs, and hence deal with input non-determinism. In order to avoid input non-determinism, `Parikshan` uses a network proxy to send all inputs received in the production container also to the test-container. Hence both of them behave in almost the same fashion.

Configuration non-determinism relates to the state of the container, the application configuration, system/kernel parameters, hardware resources etc. Once again, this can often effect the logic of the application. For eg. max no. of theads is a common configuration parameter in several multi-process server applications, and can effect the performance of the application in high workloads. `Parikshan` can deal with most configuration/state non-determinism as it captures the state of the production container at the time of cloning, and creates an exact replica. This replica should have an identical behavior to the original and should have similar memory consumption, threading behavior etc. However, some amount of inconsistency between the test-container and production container is possible.

Another kind of non-determinism is caused by concurrent applications or parallel applications which may behave differently in each execution. Capturing this kind of non-determinism is important to debug bugs such as race-conditions, locks etc. Unfortunately, the current system design of `Parikshan` cannot capture concurrency based non-determinism, we hope to revise and include this in future versions.

4.2 Slowdown

4.3 Consistency Requirements

5. TRIGGERING AND INSERTING ANALYSIS

The key idea behind sandbox testing is to debug problems in real-world scenarios Once we have forked off a clone, we are now ready to do some deeper analysis. Here we explain how debugging can potentially be done in real live systems to give the operator and developers a deeper view of the target application. `Parikshan` could potentially be used in several ways for application debugging and introspection. However, in the interest of brevity, we talk about only two categories. Firstly, inserting probes which can help in generating traces so that the user can understand the execution path of the application. Secondly, performance analysis, such as memory usage, number of context switches etc. this can give the user an idea of what could be the bottleneck for the target application.

5.1 Inserting Probes & TestCases

Execution tracing is a key debugging tool for locating any bugs. Essentially, an execution trace is a log of the functions executed, values of certain variables being tracked etc. within a time-frame. The path taken and the value of the variables are important as they explain the logic flow of the application for the *"configuration"* and *"context"* at the time of creating the clone. To generate this trace, we insert probes or watch points for variables in the application similar to existing development phase debugging tools such as GDB(gnu debugger) [?]. Inserting probes in the sandbox can be done using any existing dynamic instrumentation tools such as PIN [12], Valgrind [14], Dyninst [11]. For our purposes we used `iProbe`[5] [9] a dynamic binary instrumentation tool de-

[5]we have used iProbe only because of our familiarity

signed by the authors to insert probes in running applications. `iProbe`uses compile time flags to generate placeholders in the binary, and inserts instrumentation probes in the binary at run-time. Users can use it to insert print statements on function executions.

5.2 Performance Analysis

While execution traces provide a good indication for localizing bugs, often system level statistics, such as memory usage, cpu usage, context switches, number of threads can be a good indication towards bug localization. This is especially true if the bug is because of a configuration error or in the interaction of the application to the base kernel system.

Given that the production and the test-container have similar resources allocated to them, we believe that the system level statistics of the test-container will be a close representation of the original as it was a clone of the production container. We accept that it cannot represent the production container entirely, as there is bound to be some non-determinism in the containers. However, any input driven state changes are bound to be caught.

Previous approaches including some of the work done by the authors [17] [15] and others in academia [**?**] [**?**] have often used kernel level statistics or system call traces, in localizing bugs. The idea behind these approaches is generally to provide a black-box solution to applications. However, in practice they can only be used in lower workload's or can only provide vague clues as they need to restrict the amount of instrumentation to avoid a heavy overhead.

6. NETWORK ISSUES

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

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

7. EVALUATION

In this section we present the evaluation of `Parikshan` . The key questions facing us were:

- How does cloning the container effect the performance of the production container?
- How long of a testing-window do we have?
- How does running tests in the test-container effect the performance of the production container?

In order to answer these questions, we seperated our evaluation in looking at two different stages: cloning stage, time-window analysis.

7.1 Cloning: Micro-Benchmarks

The profile of the cloning operation can be divided in 4 stages: (1) Suspend & Dump: this is the time taken to suspend the container, (2) Pcopy after suspend: which does the rsync after the suspend of the file system of the container, (3) Copy Dump File: which copies the process state, and finally (4) Undump and Resume: which is the time taken to resume the containers. We first looked at the performance of the cloning operation to look at the time taken to do cloning while. In table 1, we show the suspend times in all 3 modes: internal, external, and google-compute, while comparing it with a production container, that is idle vs. a container which is running an apache hog benchmark [13] on it. The first column gives the average performance of the cloning operation without any hog operation running on it. An idle or a containter with minimal processing is cloned relatively fast 2.66 seconds on the idle container. We then tried to run an apache hog to make a baseline of apache's performance without cloning, and found that a simple page fetch gave us a throughput of 1691 req/s (internal mode), and then we tried to do cloning of the same container while running the hog. and found negligible change in the cloning performance. The key thing to note in these experiments for all 3 modes, was that we **did not have any connection failiure or connection refused, and only a slight decline in the throughput during the cloning operation**. Natrually, at the application layer, the tcp packet drops are hidden as packet resends from within tcp protocol hides the performance impact. To further investigate the tcp packet dropping, we ran an iperf[**?**](a tool to measure tcp benchmark) server while cloning the production container. We were indeed able to observe packet dropping for about 2 seconds in the iperf client, however, the important point to note is that there were no requests dropped for the application while doing cloning.

As explained earlier, the cloning process can be divided into two parts: an rsync operation which does an "pre-copy" of the VM, and a follow-up rsync operation while the target container is suspended, to make sure that both the production and test containers have the exact same state. The idea is to reduce the time taken to suspend the production container, so that it has minimal effect on the user. The main factor that effects this is the number of "dirty pages" in the suspend phase, which have not copied over in the pre-commit rsync operation. Natrually, the number of write operations in the container while cloning the container, will increase the number of dirty pages, and increase the time of the suspend operation. To understand the effect that i/o operations have on the cloning operation, we ran a controlled experiment gradually increasing the number of i/o writes. We use fio(flexible io tool for linux)[10], to increase the number of i/o operations while doing cloning. Specifically, we do random-write operations of a 500MB file with fixed i/o bandwidth rates. As shown in figure **??**, the cloning operation has minimal impact till a fairly decent i/o bandwidth for write operations(100Kbps). However, it increased exponentially beyond that, we attribute this to a caching behavior, but in general the increased cloning time confirmed our intuition that the cloning operation will be impacted when running an i/o intensive application. Current research in live migration has looked into further decreasing the sync time by doing active migration, and trigger page fault for the dirty bits which were not copied over. This is a similar to a copy-on-write method, that could possibly reduce our suspend time. However, it would impact, the overall performance of both systems as it would do the rsync operation for a much longer time-period.

7.2 Time-Window Size Evaluation

As explained in section 7.2 if the overhead of the test-container is too high, the buffer may overflow. This indirectly means that the test-container and the production-container are potentially out of sync. In our current de-

Modes	**Internal Mode**			**External Mode**			**Google Compute**		
	Cl	**Hog**	**Hog+Cl**	**Cl**	**Hog**	**Hog+Cl**	**Cl**	**Hog**	**Hog+Cl**
Throughput	–	1691.0 req/s	1509 req/s	–	712	625	–	510	450
Suspend + Dump	0.49	–	0.46	0.00	0.00	0.00	0.00	0.00	0.00
Pcopy after suspend	0.22	–	0.27	0.00	0.00	0.00	0.00	0.00	0.00
Copy Dump File	0.62	–	0.64	0.74	0.59	0.65	0.00	0.00	0.00
Undump and Resume	1.33	–	1.53	0.74	0.59	0.65	0.00	0.00	0.00
Total Suspend Time	2.66	–	2.91	0.74	0.59	0.65	0.00	0.00	0.00

Table 1: Performance of Live Cloning (external mode) with a random file dump process running in the container

sign we re-initiate the test-container by cloning again, and we call the time taken to reach a buffer overflow the ”time-window” for the test-container. As explained earlier, the size of this test-container, depends both on the overhead of the ”test”/”instrumentation”, as well as the incoming workload.

In this section we evaluate the testing window size using varying amounts of instrumentation, and the workload. For the purpose of this evaluation, we keep a fixed buffer size. First we use a controlled workload rate, and gradually increase the overhead, then we use another scenario, where we keep the try to keep the same overhead, and try to increase the workload. We also use real-world network packet capture data, to simulate a realistic workload and gradually increase the overhead there

7.3 Overhead while Running tests

One of the most important goals of using `Parikshan` is to allow debugging without having any overhead on the actual application. In this section we verify that this goal of `Parikshan` holds true i.e. debugging in the sandbox-container, does not effect the performance container.

8. CASESTUDIES

Parikshan enables the users to freely run any test-case in the test-container while not effecting the production container. At the same time the output of these tests should not effect the functionality or the performance of the production system. The main advantage of such a system can be seen in service oriented applications which are user facing and can hence ill-afford to be shutdown for inspecting bugs. As mentioned earlier, another major advantage is that we are able to capture live user-input.

8.1 CaseStudy 1: Debugging using Execution Tracing of MySQL bug 18511

Performance profiling such as function execution trace, execution time, resource usage etc., is often used to indicate and localize performance bugs in real world systems. While performance profiling is simple to implement, it obviously incurs an overhead and will effect user-experience of the target system. Effectively, this means that despite it’s advantages, the amount of profiling that can be done in a production system is extremely limited.

As our first case study we focused on profiling and capturing a performance bug in a session with several randomly created user transactions to MySQL. It was reported by users, that some of the user requests were running significantly slower than others. To test out `Parikshan` to catch this bug, we re-created a 2 tier client-server setup with the server(container) running a buggy mysql server application, and made a random workload with several repetitive instances of queries triggering the bug. We initiated debugging by creating a live clone of the container running the mysql server. Using a trial and error method we profiled high granularity functions in mysql and gradually looked at finer granularity modules to isolate the performance problem. This allowed us to successfully isolate the function with the performance problem.

We believe this case study shows a classical performance bug, where `Parikshan` can be used. Firstly, it’s a performance bug which is non-critical i.e. does not lead to crashes etc. `Parikshan` has been designed to look into real live bug diagnosis, rather than looking at bugs after a crash etc. [6] Secondly, we assume that the user-input (which caused the bug), occurs fairly frequently. In this case, a database query which looks into data containing chinese characters, or japanese characters could be a fairly common occurence. In our experience we found several such performance bugs which occur in only a small percentage of user transactions. These bugs are often diffcult to catch as they happen only in corner case inputs, and generally do not lead to application crashes. `Parikshan` would be useful to debug such performance errors, by giving deep insight in a live running system.

8.2 CaseStudy 2: Performance Profiling, using dyninst

CaseStudy 3: A/B Testing A/B Testing
CaseStudy 4: Fault Tolerance Testing Fault Injection to look at fault tolerance Security Honeypots?

9. 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**
- **A-B Testing**

[6]A lot of SOA applications are fault tolerant, and can continue even after the crash by relaunching processes etc. Potentially `Parikshan` can also be used in such a scenario to trace the crash itself.

- **Symbian Monkey**
- **DevOps**

10. CONCLUSION AND ACKNOWLEDGEMENTS

In this paper we described `Parikshan` a mechanism to do live-debugging of large scale systems. `Parikshan` is a novel technique to debug live systems allowing faster times to reach bug resolution. We evaluated `Parikshan` on real-world scenarios to show it's applicability and discussed several modes in which it can be used.

We would like to acknowledge ... for their insight and feedback in designing `Parikshan` , and in the evaluation of this technology. This authors are affiliated with NEC Labs, Princeton and PSL Lab at Columbia University.

11. REFERENCES

[1] Linux containers, userspace tools for the linux kernel containers. https://linuxcontainers.org.

[2] Nat: Network address translation. http://en.wikipedia.org/wiki/Network_address_translation.

[3] Netflix: On-demand internet streaming media. https://www.netflix.com.

[4] Openstack: Opensource platform for building private and public clouds. https://www.openstack.org.

[5] Openvz linux containers. http://www.openvz.org.

[6] Ploop: Containers in a file. http://openvz.org/Ploop.

[7] vzctl: Toolset to perform various operations in openvz. http://www.openvz.org/Man/vzctl.8.

[8] What is devops? http://www.radar.oreilly.com.

[9] N. Arora, H. Zhang, J. Rhee, K. Yoshihira, and G. Jiang. iprobe: A lightweight user-level dynamic instrumentation tool. In *Automated Software Engineering (ASE), 2013 IEEE/ACM 28th International Conference on*, pages 742–745. IEEE, 2013.

[10] J. Axboe. Fio-flexible io tester. *uRL: http://freecode.com/projects/fio*, 2008.

[11] B. Buck and J. K. Hollingsworth. An api for runtime code patching. *Int. J. High Perform. Comput. Appl.*

[12] C.-K. e. a. Luk. Pin: building customized program analysis tools with dynamic instrumentation. In *PLDI '05*, 2005.

[13] D. Mosberger and T. Jin. httperfâĂŤa tool for measuring web server performance. *ACM SIGMETRICS Performance Evaluation Review*, 26(3):31–37, 1998.

[14] N. Nethercote and J. Seward. Valgrind: a framework for heavyweight dynamic binary instrumentation. In *PLDI '07*, 2007.

[15] J. Rhee, H. Zhang, N. Arora, G. Jiang, and K. Yoshihira. Software system performance debugging with kernel events feature guidance. In *Network Operations and Management Symposium (NOMS), 2014 IEEE*, pages 1–5. IEEE, 2014.

[16] A. Tseitlin. The antifragile organization. *Communications of the ACM*, 56(8):40–44, 2013.

[17] H. Zhang, J. Rhee, N. Arora, S. Gamage, G. Jiang, K. Yoshihira, and D. Xu. Clue: System trace analytics for cloud service performance diagnosis. In *Network Operations and Management Symposium (NOMS), 2014 IEEE*, pages 1–9. IEEE, 2014.