Kensa: Sandboxed, Online Debugging of Production Bugs with No Overhead

Paper #43 - 14 pages total

Abstract

Short time-to-bug localization and resolution is extremely important for any 24x7 service-oriented application. In this work, we present a novel-mechanism which allows debugging of production systems on-the-fly. We leverage user-space virtualization technology (OpenVZ/LXC), to launch replicas from running instances of a production application, thereby having two containers: *production* (which provides the real output), and *debug* (for debugging). The *debug container* provides a sandbox environment for debugging without any perturbation to the production environment. Customized network-proxy agents asynchronously replicate and replay network inputs from clients to both the production and debug-container, as well as safely discard all network output from the debug-container. We evaluated this low-overhead record and replay technique on five real-world applications, finding that it was effective at reproducing real bugs. In comparison to existing monitoring solutions which can slow-down production applications, *Kensa* allows application monitoring at "zero-overhead".

1. Introduction

Rapid resolution of incident (error/alert) management [40] in online service-oriented systems [13, 14, 38, 47] is extremely important. The large scale of such systems means that any downtime has significant financial penalties for all parties involved. However, the complexities of virtualized environments coupled with large distributed systems have made bug localization extremely difficult. Debugging such production systems requires careful re-creation of a similar environment and workload, so that developers can reproduce and identify the cause of the problem.

Existing state-of-art techniques for monitoring production systems rely on execution trace information. These traces can be replayed in a developer's environment, allowing them to use dynamic instrumentation and debugging tools to understand the fault that occurred in production. On one extreme, these monitoring systems may capture only very minimal, high level information, for instance, collecting existing log information and building a model of the system and its irregularities from it [8, 20, 23, 33]. While these systems impose almost no overhead on the production system being debugged (since they simply collect log information already being collected, or have light-weight monitoring), they are limited in their fault finding and redproduction power, hence limited in their utility to developers. On the other extreme, some monitoring systems capture complete execution traces, allowing the entire application execution to be exactly reproduced in a debugging environment [6, 19, 29, 37]. Despite much work towards minimizing the amount of such trace data captured, overheads imposed by such tracing can still be unacceptable for production use: in most cases, the overhead of tracing is at least 10%, and it can balloon up to 2-10x overhead. [48, 53].

We seek to allow developers to diagnose and resolve crashing and non-crashing failures of production service-oriented systems *without suffering any performance overhead*. Our key insight is that for most service-oriented systems, a failure can be reproduced simply by replaying the network inputs passed to the application. For these failures, capturing very low-level sources of non-determinism (e.g. thread scheduling or general system calls, often with very high overhead) is unnecessary to successfully and automatically reproduce the buggy execution in a development environment. We evaluated this insight by studying 16 real-world bugs (see Section 4), which we were able to trigger by only duplicating and replaying network packets. Furthermore, we categorized 217 bugs from three real world applications, finding that most were similar in nature to the 16 that we reproduced, suggesting that our approach would be applicable to them as well (see Section 5.3).

Guided by this insight, we have created *Kensa*[1], which allows for real-time, online debugging of production services

[1] *Kensa* is the *japanese* word for testing

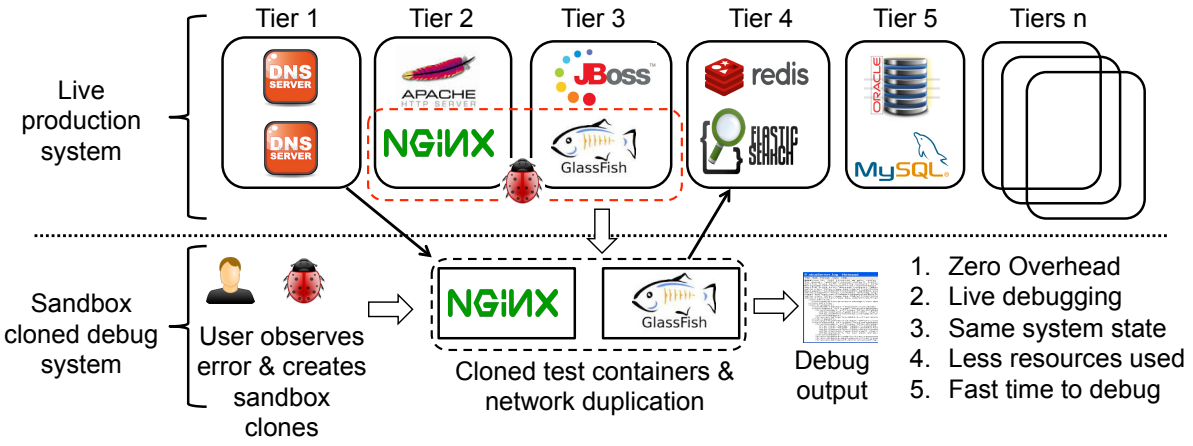

Figure 1. Workflow of *Kensa* in a live multi-tier production system with several interacting services. When the administrator of the system observes errors in two of it's tiers, he can create a sandboxed clone of these tiers and observe/debug them in a sandbox environment without impacting the production system.

without imposing any performance penalty. At a high level, *Kensa* leverages live cloning technology to create a sandboxed replica environment. This replica is kept isolated from the real world so that developers can modify the running system in the sandbox to support their debugging efforts without fear of impacting the production system. Once the replica is executing, *Kensa* replicates all network inputs flowing to the production system, buffering and feeding them (without blocking the production system) to the debug system. Within that debug system, developers are free to use heavy-weight instrumentation that would not be suitable in a production environment to diagnose the fault. Meanwhile, the production system can continue to service other requests. *Kensa* can be seen as very similar to tools such as Aftersight [15] that offload dynamic analysis tasks to replicas and VARAN [31] that support multi-version execution, but differs in that its high-level recording level (network inputs, rather than system calls) allows it to have significantly lower overhead.

Kensa focuses on helping developers debug faults *online* — as they occur in production systems. We expect *Kensa* to be used in cases of tricky bugs that are highly sensitive to their environment, such as semantic bugs, performance bugs, resource-leak errors, configuration bugs, and concurrency bugs. Although in principle, *Kensa* can be used to diagnose crashing bugs, we target primarily non-crashing bugs, where it is important for the production system to remain running even after a bug is triggered, for instance, to continue to process other requests. We present a more detailed explanation of these categories in Section 4.

We leverage container virtualization technology (e.g., Docker [42], OpenVZ [36]), which can be used to prepackage services so as to make deployment of complex multi-tier systems easier (i.e. DockerHub [12, 18] provides prepackaged containers for storage, webserver, database services etc.). Container based virtualization is now increasingly being used in practice [10]. In contrast to VM's containers run natively on the physical host (i.e. there is no hypervisor layer in between), this means that there is no additional overhead, and near-native performance for containers [24, 54]. While *Kensa* could also be deployed using VM's, container virtualization is much more light weight in terms of resource usage.

The key benefits of our system are:

- **Zero Overhead Monitoring:** While existing approaches have focused on minimizing the recording overhead. *Kensa* uses novel non-blocking network duplication to avoid any overhead at all in the production environment.
- **Sandbox debugging:** *Kensa* provides a cloned sandbox environment to debug the production application. This allows a safe mechanism to diagnose the error, without impacting the functionality of the application.
- **Capture large-scale context:** Allows capturing the context of large scale production systems, with long running applications. Under normal circumstances capturing such states is extremely difficult as they need a long running test input and large test-clusters.

The rest of the paper is organized as follows. In Section 2, we describe a motivating scenario. Section 3 and 3.5 describe the design and implementation of *Kensa* and each of it's internal components. We then present a case study of 16 real-world bugs successfully reproduced by *Kensa* in Section 4. This is followed by the evaluation in Section 5. In Section 6, we discuss potential applications of *Kensa*. Finally, we discuss some challenges in Section 7, give some related work in Section 8 and conclude.

2. Motivating Scenario

Consider the complex multi-tier service-oriented system shown in Figure 1 that contains several interacting services (web servers, application servers, search and index-

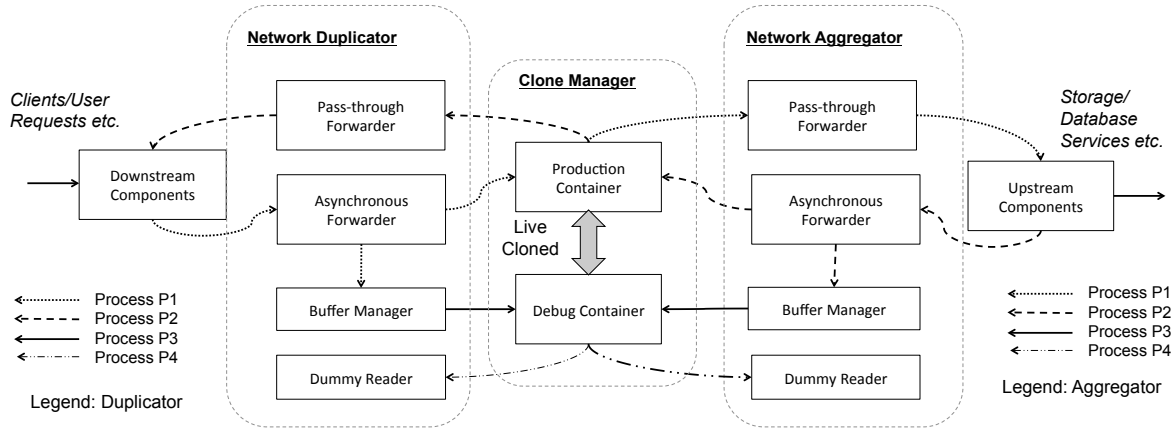

Figure 2. High level architecture of *Kensa*, showing the main components: Network Duplicator, Network Aggregator, and Cloning Manager. The replica (debug container) is kept in sync with the master (production container) through network-level record and replay. In our evaluation, we found that this light-weight procedure was sufficient to reproduce many real bugs.

ing, database, etc.). The system is maintained by operators who can observe the health of the system using lightweight monitoring that is attached to the deployed system. At some point, an unusual memory usage is observed in the glassfish application server, and some error logs are generated in the Nginx web server. Administrators can then surmise that there is a potential memory leak/allocation problem in the app-server or a problem in the web server. However, with a limited amount of monitoring information, they can only go so far.

Typically, trouble tickets are generated for such problems, and they are debugged offline. However using *Kensa*, administrators can generate replicas of the Nginx and Glassfish containers as ***Nginx-debug*** and ***glassfish-debug***. *Kensa*'s network duplication mechanism ensures that the debug replicas receive the same inputs as the production containers and that the production containers continue to provide service without interruption. This separation of the production and debug environment allows the operator to use dynamic instrumentation tools to perform deeper diagnosis without fear of additional disruptions due to debugging. Since the replica is cloned from the original potentially "buggy" *production container*, it will also exhibit the same memory leaks/or logical errors. Additionally, *Kensa* can focus on the "buggy" parts of the system, without needing to replicate the entire system in a test-cluster. This process will greatly reduce the time to bug resolution, and allow real-time bug diagnosis capability.

The replica can be created at any time: either from the start of execution, or at any point during execution that an operator deems necessary, allowing for post-facto analysis of the error, by observing execution traces of incoming requests (in the case of performance bugs and memory leaks, these will be persistent in the running system). Within the debug replica, the developer is free to employ any dynamic analysis tools to study the buggy execution, as long as the only side-effect those tools is on execution speed.

3. *Kensa*

In Figure 2, we show the architecture of *Kensa* when applied to a single mid-tier application server. *Kensa* consists of 3 modules: **Clone Manager**: manages "live cloning" between the production containers and the debug replicas, **Network Duplicator**: manages network traffic duplication from downstream servers to both the production and debug containers, and **Network Aggregator**: manages network communication from the production and debug containers to upstream servers. The network duplicator also performs the important task of ensuring that the production and debug container executions do not diverge. The duplicator and aggregator can be used to target multiple connected tiers of a system by duplicating traffic at the beginning and end of a workflow. Furthermore, the aggregator module is not required if the debug-container has no upstream services.

3.1 Clone Manager

Live migration [16, 28, 43] refers to the process of moving a running virtual machine or container from one server to another, without disconnecting any client or process running within the machine (this usually incurs a short or negligible suspend time). In contrast to live migration where the original container is destroyed, the "Live Cloning" process used in *Kensa* requires both containers to be actively running, and be still attached to the original network. The challenge here is to manage two containers with the same identities in the network and application domain. This is important, as the operating system and the application processes running in it may be configured with IP addresses, which cannot be changed on the fly. Hence, the same network identifier should map to two separate addresses, and enable communication with no problems or slowdowns.

We now describe two modes (see Figure 3) in which cloning has been applied, followed by the algorithm for live cloning:

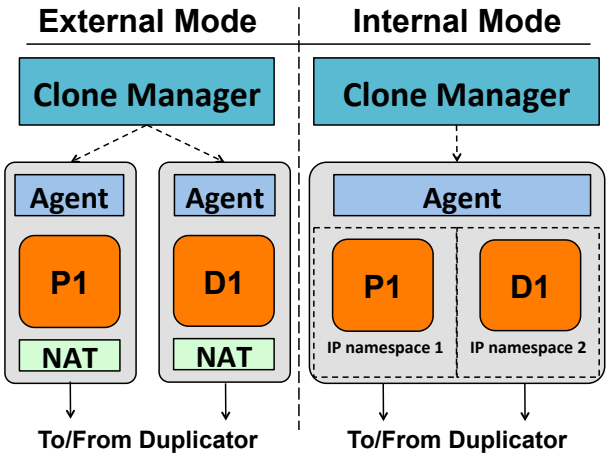

Figure 3. External and Internal Mode for live cloning: P1 is the production, and D1 is the debug container, the clone manager interacts with an agent which has drivers to implement live cloning.

Internal Mode: In this mode, we allocate the production and debug 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). Additionally, it is more cost-effective since the number of servers remain the same. On the other hand, co-hosting the debug and production containers could potentially have an adverse effect on the performance of the production container because of resource contention. Network identities in this mode are managed by encapsulating each container in separate network namespaces [3]. This allows both containers to have the same IP address with different interfaces. The duplicator is then able to communicate to both these containers with no networking conflict.

External Mode: In this mode we provision an extra server as the host of our debug-container (this server can host more than one debug-container). While this mechanism can have a higher overhead in terms of suspend time (dependent on workload) and requires provisioning an extra host-node, the advantage of this mechanism is that once cloned, the debug-container is totally separate and will not impact the performance of the production-container. We believe that external mode will be more practical in comparison to internal mode, as cloning is likely to be transient, and high network bandwidth between physical hosts can offset the slowdown in cloning performance. Network identities in external mode are managed using NAT [2] (network address translator) in both host machines. Hence both containers can have the same address without any conflict.[2]

[2] Another additional mode can be *Scaled Mode*: This can be viewed as a variant of the external mode, where we can execute debug analysis in parallel on more than one debug-containers each having its own cloned connection. This will distribute the instrumentation load and allow us to do more analysis concurrently, without overflowing the buffer. We aim to explore this in the future.

Algorithm 1 describes the specific process for cloning some production container P1 from Host H1 to replica D1 on Host H2.

Algorithm 1 Live cloning algorithm using OpenVZ

1. Safety checks and pre-processing (ssh-copy-id operation for password-less rsync, checking pre-existing container ID's, version number etc.)
2. Create and synchronize file system of P1 to D1
3. Set up port forwarding, duplicator, and aggregator
4. Suspend the production container P1
5. Checkpoint & dump the process state of P1
6. Since step 2 and 5 are non-atomic operations, some files may be outdated. A second sync is run when the container is suspended to ensure P1 and D1 have the same state
7. Resume both production and debug containers

The suspend time of cloning depends on the 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. This suspend time can be viewed as an amortized cost in lieu of instrumentation overhead. We evaluate the performance of live cloning in Section 5.1.

3.2 Network Duplicator and Aggregator

The network proxy duplicator and aggregator are composed of the following internal components:

- **Synchronous Passthrough**: The synchronous passthrough is a daemon that takes the input from a source port, and forwards it to a destination port. The passthrough is used for communication from the production container out to other components (which is not duplicated).
- **Asynchronous Forwarder**: The asynchronous forwarder is a daemon that takes the input from a source port, and forwards it to a destination port, and also to an internal buffer. The forwarding to the buffer is done in a non-blocking manner, so as to not block the network forwarding.
- **Buffer Manager**: Manages a FIFO queue for data kept internally in the proxy for the debug-container. It records the incoming data, and forwards it a destination port.
- **Dummy Reader**: This is a standalone daemon, which reads and drops packets from a source port

Proxy Network Duplicator: To successfully perform online debugging in the replica to work, both production and debug containers must receive the same input. A major challenge in this process is that the production and debug container may execute at different speeds (debug will be slower than

production): this will result in them being out of sync. Additionally, we need to accept responses from both servers and drop all the traffic coming from the debug-container, while still maintaining an active connection with the client. Hence simple port-mirroring and proxy mechanisms will not work for us.

TCP is a connection-oriented protocol and is designed for stateful delivery and acknowledgment that each packet has been delivered. Packet sending and receiving are blocking operations, and if either the sender or the receiver is faster than the other the send/receive operations are automatically blocked or throttled. 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 2), where $Y < X$. Then automatically, the speed of link 1 and link 2 will be throttled to $YMbps$ per second, i.e the packet sending at the client will be throttled to accommodate the production server. Network throttling is a default TCP behavior to keep the sender and receiver synchronized. However, if we also send packets to the debug-container sequentially in link 3 the performance of the production container will be dependent on the debug-container. If the speed of link 3 is $Z\ Mbps$, where $Z < Y$, and $Z < X$, then the speed of link 1, and link 2 will also be throttled to $Z\ Mbps$. The speed of the debug container is likely to be slower than production: this may impact the performance of the production container.

Our solution is a customized TCP level proxy. This proxy duplicates network traffic to the debug container while maintaining the TCP session and state with the production container. Since it works at the TCP/IP layer, the applications are completely oblivious to it. To understand this better let us look at Figure 2: Here each incoming connection is forwarded to both the production container and the debug container . This is a multi-process job involving 4 parallel processes (P1-P4): In P1, the asynchronous forwarder sends data from client to the production service, while simultaneously sending it to the buffer manager in a non-blocking send. This ensures that there is no delay in the flow to the production container because of slow-down in the debug-container. In P2, the pass-through forwarder reads data from the production and sends it to the client (downstream component). Process P3, then sends data from Buffer Manager to the debug container, and Process P4 uses a dummy reader, to read from the production container and drops all the packets

The above strategy allows for non-blocking packet forwarding and enables a key feature of *Kensa*, whereby it avoids slowdowns in the debug-container to impact the production container. We take the advantage of an in-memory buffer, which can hold requests for the debug-container, while the production container continues processing as normal. A side-effect of this strategy is that if the speed of the debug-container is too slow compared to the packet arrival rate in the buffer, it may eventually lead to an overflow. We call the time taken by a connection before which the buffer overflows its *debug-window*. We discuss the implications of the *debug window* in Section 3.3.

Proxy Network Aggregator: The proxy described in Section 3.2 is used to forward requests from downstream tiers to production and debug containers. While the network duplicator duplicates incoming requests, the network aggregator manages incoming "responses" for requests sent from the debug container. Imagine if you are trying to debug a mid-tier application container, the proxy network duplicator will replicate all incoming traffic from the client to both debug and the production container. Both the debug container and the production, will then try to communicate further to the backend containers. This means duplicate queries to backend servers (for instance, sending duplicate 'delete' messages to MySQL), thereby leading to an inconsistent state. Nevertheless, to have forward progress the debug-container must be able to communicate and get responses from upstream servers. The "proxy aggregator" module stubs the requests from a duplicate debug container by replaying the responses sent to the production container to the debug-container and dropping all packets sent from it to upstream servers.

As shown in Figure 2, when an incoming request comes to the aggregator, it first checks if the connection is from the production container or debug container. In process P1, the aggregator forwards the packets to the upstream component using the pass-through forwarder. In P2, the asynchronous forwarder sends the responses from the upstream component to the production container, and sends the response in a non-blocking manner to the internal queue in the buffer manager. Once again this ensures no slow-down in the responses sent to the production container. The buffer manager then forwards the responses to the debug container (Process P3). Finally, in process P4 a dummy reader reads all the responses from the debug container and discards them.

We assume that the production and the debug container are in the same state, and are sending the same requests. Hence, sending the corresponding responses from the FIFO queue instead of the backend ensures: (a) all communications to and from the debug container are isolated from the rest of the network, (b) the debug container gets a logical response for all it's outgoing requests, making forward progress possible, and (c). similar to the proxy duplicator, the communications from the proxy to internal buffer is non-blocking to ensure no overhead on the production-container.

3.3 Debug Window

Kensa's asynchronous forwarder uses an internal buffer to ensure that incoming requests proceed directly to the production container without any latency, regardless of the speed at which the debug replica processes requests. The incoming request rate to the buffer is dependent on the user, and is limited by how fast the production container manages the requests (i.e. the production container is the rate-limiter). The outgoing rate from the buffer is dependent on how fast the

debug-container processes the requests. Instrumentation overhead in the debug-container can potentially cause an increase in the transaction processing times in the debug-container. As the instrumentation overhead increases, the incoming rate of requests may eventually exceed the transaction processing rate in the debug container. If the debug container does not catch up, this in turn can lead to a buffer overflow. We call the time period until buffer overflow happens the *debug-window*. This depends on the size of the buffer, the incoming request rate, and the overhead induced in the debug-container. For the duration of the debugging-window, we assume that the debug-container faithfully represents the production container. Once the buffer has overflown, the debug-container may be out of sync with the production container. At this stage, the production container needs to be re-cloned, so that the replica is back in sync with the production and the buffer can be discarded. In case of frequent buffer-overflows, the buffer size needs to be increased or the instrumentation to be decreased in the replica, to allow for longer debug-windows.

The debug window size also depends on the application behavior, in particular how it launches TCP connections. *Kensa* generates a pipe buffer for each TCP connect call, and the number of pipes are limited to the maximum number of connections allowed in the application. Hence, buffer overflows happen only if the requests being sent in the same connection overflow the queue. For webservers, and application servers, the debugging window size is generally not a problem, as each request is a new "connection." This enables *Kensa* to tolerate significant instrumentation overhead without a buffer overflow. On the other hand, database and other session based services usually have small request sizes, but multiple requests can be sent in one session which is initiated by a user. In such cases, for a server receiving a heavy workload, the number of calls in a single session may eventually have a cumulative effect and cause overflows.

To further increase the *debug window*, we propose load balancing debugging instrumentation overhead across multiple debug-containers, which can each get a duplicate copy of the incoming data. For instance, debug-container 1 could have 50% of the instrumentation, and the rest on debug-container 2. We believe such a strategy would significantly reduce the chance of a buffer overflow in cases where heavy instrumentation is needed. Section 5.2 explains in detail the behavior of the debug window, and how it is impacted by instrumentation.

3.4 Divergence Checking

In *Kensa* it is possible that non-deterministic behavior (discussed in Section 7) in the two containers or user instrumentation, causes the production and debug container to diverge with time. To understand and capture this divergence, we compare the corresponding network output received in the proxy. This is an optional component, which gives us a black-box mechanism to check the fidelity of the replica based on its communication with external components. In our current prototype, we use a hash on each data packet, which is collected and stored in memory for the duration that each packet's connection is active. The degree of acceptable divergence is dependent on the application behavior, and the operator's wishes. For example, an application that includes timestamps in each of its messages (i.e. is expected to have some non-determinism) could perhaps be expected to have a much higher degree of acceptable divergence than an application that should normally be returning deterministic results.

3.5 Implementation

The clone-manager and the live cloning utility are built on top of the user-space container virtualization software OpenVZ [36]. *Kensa* extends *VZCTL* 4.8 [26] live migration facility [43], to provide support for online cloning. To make **live cloning** easier and faster, we used OpenVZ's *ploop* devices [5] as the container disk layout. The network isolation for the production container was done using Linux network namespaces [3] and NAT [2]. While *Kensa* is based on light-weight containers, we believe that *Kensa* can easily be applied to heavier-weight, traditional virtualization software where live migration has been further optimized [17, 50].

The network proxy duplicator and the network aggregator was implemented in C/C++. The forwarding in the proxy is done by forking off multiple processes each handling one send/or receive a connection in a loop from a source port to a destination port. Data from processes handling communication with the production container, is transferred to those handling communication with the debug containers using *Linux Pipes* [1]. Pipe buffer size is a configurable input based on user-specifications.

4. Case Studies

One of our core insights is that for most SOA systems, production bugs can hence be triggered by network replay alone. To validate this insight, we selected sixteen real-world bugs, applied *Kensa*, reproduced them in a production container, and observed whether they were also simultaneously reproduced in the replica. For each of the sixteen bugs that we triggered in the production environments, *Kensa* faithfully reproduced them in the replica.

We selected our bugs from those examined in previous studies [41, 55], focusing on bugs that involved performance, resource-leaks, semantics, concurrency, and configuration. We have further categorized these bugs whether they lead to a crash or not, and if they can be deterministically reproduced. Table 1 presents an overview of our study. We now discuss these bugs and our experience reproducing them with *Kensa* in greater detail:

Semantic Bugs: The majority of the bugs found in production SOA systems can be categorized as semantic bugs. These bugs often happen because an edge condition was not checked during the development stage or there was a logical error in the algorithm etc. Many such errors result in an unexpected

Bug Type	Bug ID	Application	Symptom/Cause	Determ-inistic	Crash	Trigger
Performance	MySQL #15811	mysql-5.0.15	Bug caused due to multiple calls in a loop	Yes	No	Repeated insert into table
	MySQL #26527	mysql-5.1.14	Load data is slow in a partitioned table	Yes	No	Create table with partition and load data
	MySQL #49491	mysql-5.1.38	calculation of hash values inefficient	Yes	No	MySql client select requests
Concurrency	Apache #25520	httpd-2.0.4	Per-child buffer management not thread safe	No	No	Continuous concurrent requests
	Apache #21287	httpd-2.0.48, php-4.4.1	Dangling pointer due to atomicity violation	No	Yes	Continuous concurrent request
	MySQL #644	mysql-4.1	data-race leading to crash	No	Yes	Concurrent select queries
	MySQL #169	mysql-3.23	Race condition leading to out-of-order logging	No	No	Delete and insert requests
	MySQL #791	mysql-4.0	Race - visible in logging	No	No	Concurrent flush log and insert requests
Semantic	Redis #487	redis-2.6.14	Keys* command duplicate or omits keys	Yes	No	Set keys to expire, execute specific reqs
	Cassandra #5225	cassandra-1.5.2	Missing columns from wide row	Yes	No	Fetch columns from cassandra
	Cassandra #1837	cassandra-0.7.0	Deleted columns become available after flush	Yes	No	Insert, delete, and flush columns
	Redis #761	redis-2.6.0	Crash with large integer input	Yes	Yes	Query for input of large integer
Resource Leak	Redis #614	redis-2.6.0	Master + slave, not replicated correctly	Yes	No	Setup replication, push and pop some elements
	Redis #417	redis-2.4.9	Memory leak in master	Yes	No	Concurrent key set requests
Configuration	Redis #957	redis-2.6.11	Slave cannot sync with master	Yes	No	Load a very large DB
	HDFS #1904	hdfs-0.23.0	Create a directory in wrong location	Yes	No	Create new directory

Table 1. List of real-world production bugs studied with *Kensa*

output or possibly can crash the system. We recreated 4 real-world production bugs from Redis [14] queuing system, and Cassandra [38] a NoSQL database.

For instance, one such bug Redis#761 is an integer overflow error. This error is triggered, when the client tries to insert and store a very large number. This leads to an unmanaged exception, which crashes the production system. Others such as Redis#487 resulted in expired keys still being retained in Redis, because of an unchecked edge condition. While this error does not lead to any exception or any error report in application logs, it gives the user a wrong output. In the case of such logical errors, the application keeps processing, but the internal state can stay incorrect. In our experiments, we were able to clone the input of the production in the debug containers and easily observe both these errors.

Performance Bugs: These bugs do not lead to crashes but cause significant impact to user satisfaction. A casestudy [32] showed that a large percentage of real-world performance bugs can be attributed to uncoordinated functions, executing functions that can be skipped, and inefficient synchronization among threads (for example locks held for too long etc.). Typically, such bugs can be caught by function level execution tracing and tracking the time taken in each execution function. Another key insight provided in [32] was that two-thirds of the bugs manifested themselves when special input conditions were met, or execution was done at scale. Hence, it is difficult to capture these bugs with traditional offline white-box testing mechanisms.

For one of the bugs in MySQL#15811, it was reported that some of the user requests which were dealing with complex scripts (Chinese, Japanese), were running significantly slower than others. To evaluate *Kensa*, we re-created a two-tier client-server setup with the server (container) running a buggy MySQL server and sent queries to the production container with complex scripts (Chinese). These queries were asynchronously replicated, in the debug container. To further investigate the bug-diagnosis process, we also turned on execution tracing in the debug container using SystemTap [21]. This gives us the added advantage, of being able to profile and identify the functions responsible for the slow-down, without the tracing having any impact on production.

Resource Leaks: Resource leaks can be either memory leak or un-necessary zombie processes. Memory leaks are common errors in service-oriented systems, especially in C/C++ based applications which allow low-level memory management by users. These leaks build up over time and can cause slowdowns because of resource shortage, or crash the system. Debugging leaks can be done either using systematic debugging tools like Valgrind, which use shadow memory to track all objects, or memory profiling tools like VisualVM, mTrace, or PIN, which track allocations, de-allocations, and heap size. Although Valgrind is more complete, it has a very high overhead and needs to capture the execution from the beginning to the end (i.e., needs application restart). On the other hand, profiling tools are much lighter and can be dynamically patched to a running process.

Let us take Redis#417 for instance, here we had a redis master and slave set up for both production and debug container. We then triggered the bug by running concurrent requests through the client which can trigger the memory leak. The memory leak was easily visible in the debug container by turning on debug tracing, which showed a growing memory usage.

Concurrency Bugs One of the most subtle bugs in production systems is caused due to concurrency errors. These bugs are hard to reproduce, as they are non-deterministic, and may

or may not happen in a given execution. Unfortunately, *Kensa* cannot guarantee that if a buggy execution is triggered in the production container, an identical execution will trigger the same error in the debug container. However, given that the debug container is a live-clone of the production container, and that it replicates the state of the production container entirely, we believe that the chances of the bug also being triggered in the debug container are quite high. Additionally, the debug container is a useful tracing utility to track thread lock and unlock sequences, to get an idea of the concurrency bug.

Configuration Bugs: Configuration errors are usually caused by wrongly configured parameters, i.e., they are not bugs in the application, but bugs in the input (configuration). These bugs usually get triggered at scale or for certain edge cases, making them extremely difficult to catch.

A simple example of such a bug is Redis#957, here the slave is unable to sync with the master. The connection with the slave times out and it's unable to sync because of the large data. While the bug is partially a semantic bug, as it could potentially have checks and balances in the code. The root cause itself is a lower output buffer limit. Once again, it can be easily observed in our debug-containers that the slave is not synced, and can be investigated further by the debugger.

5. Evaluation

To evaluate the performance of *Kensa*, we pose and answer the following research questions:

RQ1: How long does it take to create a live clone of a production container and what is it's impact on the performance of the production container?

RQ2: What is the size of the debugging window, and how does it depend on resource constraints?

RQ3: Can we generalize the results of our case study to see if *Kensa* can target even more real bugs?

We evaluated the **internal mode** on two identical VM's with an Intel i7 CPU, with 4 Cores, and 16GB RAM each in the same physical host (one each for production and debug containers). We evaluated the **external mode** on two identical host nodes with Intel Core 2 Duo Processor, 8GB of RAM. All evaluations were performed on CentOS 6.5.

5.1 Live Cloning Performance

As explained in Section 3, a short suspend time during live cloning is necessary to ensure that both containers are in the exact same system state. The suspend time during live cloning can be divided in 4 parts: (1) Suspend & Dump: time taken to pause and dump the container, (2) Pcopy after suspend: time required to complete rsync operation (3) Copy Dump File: time taken to copy an initial dump file. (4) Undump & Resume: time taken to resume the containers. To evaluate "live cloning", we ran a micro-benchmark of I/O operations, and evaluated live-cloning on some real-world applications running real-workloads.

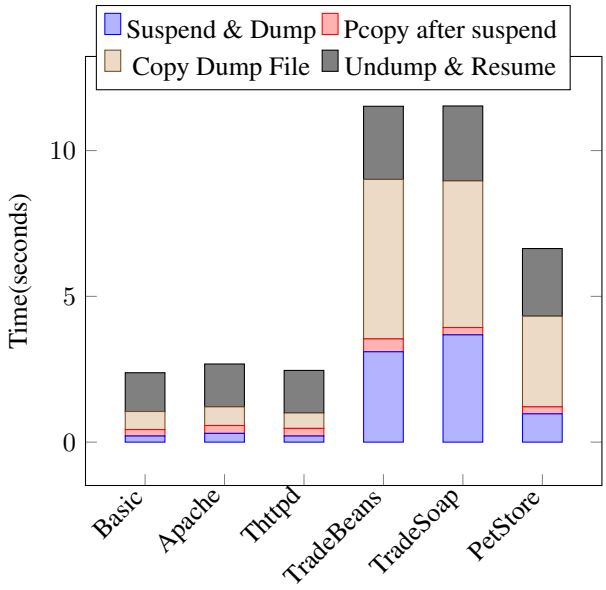

Figure 4. Suspend time for live cloning, when running a representative benchmark

Real world applications and workloads: To begin to study the overhead of live cloning, we performed an evaluation using five well-known applications. Figure 4 presents the suspended times for five well-known applications when cloning a replica with *Kensa*. We ran the httperf [44] benchmark on Apache and *thttpd* to compute max throughput of the web-servers, by sending a large number of concurrent requests. Tradebeans and Tradesoap are both part of the dacapo [11] benchmark "DayTrader" application. These are realistic workloads, which run on a multi-tier trading application provided by IBM. PetStore [4] is also a well known J2EE reference application. We deployed PetStore in a 3-tier system with JBoss, MySQL and Apache servers, and cloned the app-server. The input workload was a random set of transactions which were repeated for the duration of the cloning process.

As shown in Figure 4, for Apache and Thttpd the container suspend time ranged between 2-3 seconds. However, in more memory intensive application servers such as PetStore and DayTrader, the total suspend time was higher (6-12 seconds). Nevertheless, we did not experience any timeouts or errors for the requests in the workload[3]. However, this did slowdown requests in the workload. This shows that short suspend times are largely not visible or have minimal performance impact to the user, as they are within the time out range of most applications. Further, a clean network migration process ensures that connections are not dropped, and are executed successfully. We felt that these relatively fast temporary app suspensions were a reasonable price to pay to launch an otherwise overhead-free debug replica. To further

[3] In case of packet drops, requests are resent both at the TCP layer, and the application layer. This slows down the requests for the user, but does not drop them

characterize the suspend time imposed by the live cloning phase of *Kensa*, we created a synthetic micro-benchmark to push *Kensa* towards its limit.

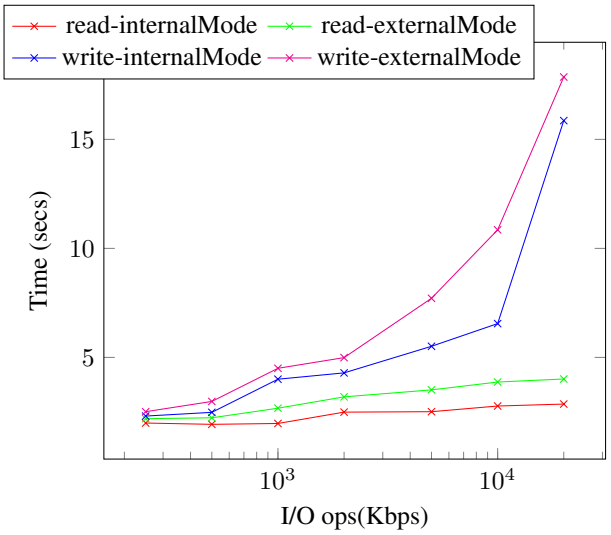

Figure 5. Live Cloning suspend time with increasing amounts of I/O operations

Micro Benchmark using I/O operations: The main factor that impacts suspend time is the number of "dirty pages" in the suspend phase, which have not been copied over in the pre-copy rsync operation (see section 3.1). To understand this better, we use *fio* (flexible I/O tool for Linux) [7], to gradually increase the number of I/O operations while doing live cloning. We run the *fio* tool to do read and writes of random values with a controlled I/O bandwidth. Additionally, we ensure that the I/O job being processed by *fio* is long enough to last through the cloning process.

As shown in figure 5, read operations have a much smaller impact on suspend time of live cloning compared to write operations. This can be attributed to the increase of "dirty pages" in write operations, whereas for read, the disk image remains largely the same. The internal mode is much faster than the external mode, as both the production and debug-container are hosted in the same physical device. We believe, that for higher I/O operations, with a large amount of "dirty-pages", network bandwidth becomes a bottleneck: leading to longer suspend times. Overall in our experiments, the internal mode is able to manage write operation up to 10 Mbps, with a total suspend-time of approx 5 seconds. Whereas, the external mode is only able to manage up to 5-6 Mbps, for a 5 sec suspend time.

To answer **RQ1**, live cloning introduces a short suspend time in the production container dependent on the workload. Write intensive workloads will lead to longer suspend times, while read intensive workloads will take much less. Suspend times in real workload on real-world systems vary from 2-3 seconds for webserver workloads to 10-11 seconds for application/database server workloads. Compared to external mode, internal mode had a shorter suspend time. A production-quality implementation could reduce suspend time further by rate-limiting incoming requests in the proxy, or using copy-on-write mechanisms and faster shared file system/storage devices already available in several existing live migration solutions.

5.2 Debug Window Size

To understand the size of the debug-window and it's dependence on resources, we did some experiments on real-world applications, by introducing a delay while duplicating the network input. This gave us some real-world idea of buffer overflow and it's relationship to the buffer size and input workload. Since it was difficult to observe systematic behavior in a live system to understand the decay rate of the debug-window, we also did some simulation experiments, to see how soon the buffer would overflow for different input criteria.

Input Rate	Debug Window	Pipe Size	Slowdown
530 bps, 27 rq/s	∞	4096	1.8x
530 bps, 27 rq/s	8 sec	4096	3x
530 bps, 27 rq/s	72 sec	16384	3x
Pois., λ = 17 rq/s	16 sec	4096	8x
Pois., λ = 17 rq/s	18 sec	4096	5x
Pois.,λ = 17 rq/s	∞	65536	3.2x
Pois.,λ = 17 rq/s	376 sec	16384	3.2x

Table 2. Approximate debug window sizes for a MySQL request workload

Experimental Results: We call the time taken to reach a buffer overflow the "debug-window". As explained earlier, the size of this debug-window depends on the overhead of the "instrumentation", the incoming workload distribution, and the size of the buffer. To evaluate the approximate size of the debug-window, we sent requests to both a production and debug MySQL container via our network duplicator. Each workload ran for about 7 minutes (10,000 "select * from table" queries), with varying request workloads. We also profiled the server, and found that is able to process a max of 27 req/s[4] in a single user connect session. For each of our experiments, we vary the buffer sizes to get an idea of debug-window. Additionally, we generated a slowdown by first modeling the time taken by MySQL to process requests (27 req/s or 17req/s), and then putting an approximate sleep in the request handler.

[4] Not the same as bandwidth, 27 req/s is the maximum rate of sequential requests MySQL server is able to handle for a user session

Initially, we created a connection and sent requests at the maximum request rate the server was able to handle (27 req/s). We found that for overheads up-to 1.8x (approx) we experienced no buffer overflows. For higher overheads the debug window rapidly decreased, primarily dependent on buffer-size, request size, and slowdown.

Next, we mimic user behavior, to generate a realistic workload. We send packets using a Poisson process with an average request rate of 17 requests per second to our proxy. This varies the inter-request arrival time, and let's the cloned debug-container catch up with the production container during idle time-periods in between request bursts. We observed, that compared to earlier experiments, there was more slack in the system. This meant that our system was able to tolerate a much higher overhead (3.2x) with no buffer overflows.

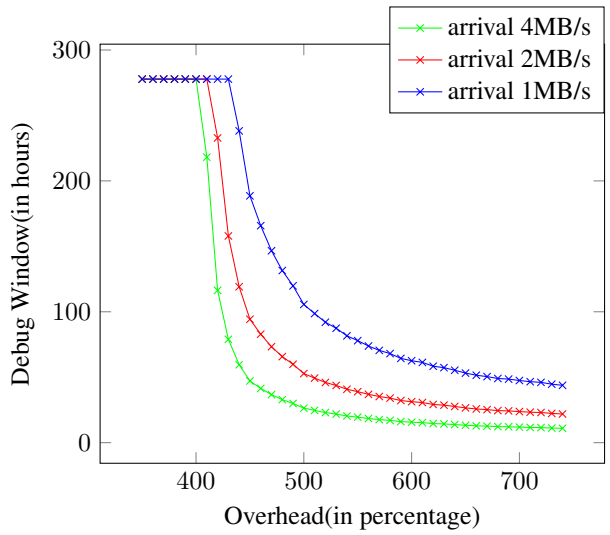

Figure 6. Simulation results for debug-window size. Each series has a constant arrival rate, and the buffer is kept at 64GB.

Simulation Results: In our next set of experiments, we simulate packet arrival and service processing for a buffered queue in SOA applications. We use a discrete event simulation based on an MM1 queue, which is a classic queuing model based on Kendall's notation [34], and is often used to model SOA applications with a single buffer based queue. Essentially, we are sending and processing requests based on a Poisson distribution with a finite buffer capacity. In our simulations (see Figure 6), we kept a constant buffer size of 64GB, and iteratively increased the overhead of instrumentation, thereby decreasing the service processing time. Each series (set of experiments), starts with an arrival rate approximately 5 times less than the service processing time. This means that at 400% overhead, the system would be running at full capacity (for stable systems SOA applications generally operate at much less than system capacity). Each simulation instance was run for 1000000 seconds or 277.7 hours. We gradually increased the instrumentation by 10% each time, and observed the *hitting-time* of the buffer (time it takes for the buffer to overflow for the first time). As shown there is no buffer overflow in any of the simulations until the overhead reaches around 420-470%, beyond this the debug-window decreases exponentially. Since beyond 400% overhead, the system is over-capacity, the queue will start filling up fairly quickly. This clarifies the behavior we observed in our experiments, where for lower overheads (1.8-3.2x) we did not observe any overflow, but beyond a certain point, we observed that the buffer would overflow fairly quickly. Also as shown in the system, since the buffer size is significantly larger than the packet arrival rate, it takes some time for the buffer to overflow (several hours). We believe that while most systems will run significantly under capacity, large buffer sizes can ensure that our debug-container may be able to handle short bursts in the workload. However, a system running continuously at capacity is unlikely to tolerate significant instrumentation overhead.

> To answer **RQ2**, we found that the debug-container can stay in a stable state without any buffer overflows as long as the instrumentation does not cause the service times to become less than the request arrival rate. Furthermore, a large buffer will allow handling of short bursts in the workload until the system returns back to a stable state. The debug-window can allow for a significant slowdown, which means that many existing dynamic analysis techniques [25, 46], as well as most fine-grained tracing [23, 33] can be applied on the debug-container without leading to an incorrect state.

5.3 A survey of real-world bugs

In Table 3, we present the results of a survey of bug reports of three production SOA applications. In order to understand how we did the survey, let us look at MySQL as an example. We first searched for bugs which were tagged as "fixed" by developers and dumped them. We then chose a random timeline (2013-2014) and filtered out all bugs which belonged to non-production components - like documentation, installation failure, compilation failure. We then manually went through each of the bug-reports, filtering out the ones which were mislabeled or were reported based on code-analysis, or did not have a triggering test report (essentially we focused only on bugs that happened during production scenarios). We then classified these bugs into the categories shown in Table 3 based on the bug-report description, and the patch fix, to-do action item for the bug.

One of the core-insights provided by this survey was that most bugs (93%) triggered in production systems are deterministic in nature (everything but concurrency bugs), among which the most common ones are semantic bugs (80%). This is understandable, as they usually happen because of unex-

Category	Apache	MySQL	HDFS
Performance	3	10	6
Semantic	36	73	63
Concurrency	1	7	6
Resource Leak	5	6	1
Total	45	96	76

Table 3. Survey and classification of bugs

pected scenarios or edge cases, that were not thought of during testing. Recreation of these bugs depend only on the state of the machine, the running environment (other components connected when this bug was triggered), and network input requests, which trigger the bug scenario. *Kensa* is a useful testing tool for testing these deterministic bugs in an exact clone of the production state, with replicated network input. The execution can then be traced at a much higher granularity than what would be allowed in production containers, to find the root cause of the bug.

On the other hand, concurrency errors, which are non-deterministic in nature make up for less than 7% of the production bugs. Owing to non-determinism, it is possible that the same execution is not triggered. However concurrent points can still be monitored and a post-facto search of different executions can be done to find the bug [25, 51] to capture these non-deterministic errors.

To answer **RQ3**, we found that almost 80% of bugs were semantic in nature, while less than 6% of the bugs are non-deterministic. About 13-14% of bugs are performance and resource-leak bugs, which are generally persistent in the system.

6. Applications of Live Debugging

Statistical Testing: One well-known technique for debugging production applications is statistical testing. This is achieved by having predicate profiles from both successful and failing runs of a program and applying statistical techniques to pinpoint the cause of the failure. The core advantage of statistical testing is that the sampling frequency of the instrumentation can be decreased to reduce the instrumentation overhead. However, the instrumentation frequency for such testing to be successful needs to be statistically significant. Unfortunately, overhead concerns in the production environment limit the frequency of instrumentation. In *Kensa*, the buffer utilization can be used to control the frequency of such statistical instrumentation in the debug-container. This would allow the user to utilize the slack available in the debug-container for instrumentation to it's maximum, without leading to an overflow. Thereby improving the efficiency of statistical testing.

Record and Replay: Record and Replay techniques have been proposed to replay production site bugs. However, they are not yet used in practice as they can impose unacceptable overheads in the service processing time. *Kensa* replicas can be used to do recording at a much finer granularity (higher overhead), allowing for easy and fast replays offline. Similar to existing mechanisms, the system can be replayed can then be used for offline debugging, without imposing any recording overhead to the production container.

Patch Testing: Bug fixes and patches to resolve errors, often need to undergo testing in the offline environment and are not guaranteed to perform correctly. Patches can be made to the replica instead. The fix can be traced and observed if it is correctly working, before moving it to the production container. This is similar in nature to AB-Tesing, which is applied to find if a new fix is useful or works [22]

7. Discussion and Limitations

Through our case studies and evaluation, we concluded that *Kensa* can faithfully reproduce many real bugs in complex applications with no running-overhead. However, there may be several threats to the validity of our experiments. For instance, in our case study, the bugs that we selected to study may not be truly representative of a broad range of different faults. Perhaps, *Kensa*'s low-overhead network record and replay approach is less suitable to some classes of bugs. To alleviate this concern, we selected bugs that represented a wide range of categories of bugs, and further, selected bugs that had already been studied in other literature, to alleviate a risk of selection bias. We further strengthened this studied with a follow-up categorization of 217 bugs in three real-world applications, finding that most of those bugs were semantic in nature, and very few were non-deterministic, and hence, having similar characteristics to those 16 that we reproduced.

There are also several underlying limitations and assumptions regarding *Kensa*'s applicability:

Non-determinism: Non-determinism can be attributed to three main sources (1) system configuration, (2) application input, and (3) ordering in concurrent threads. Live cloning of the application state ensures that both applications are in the same "system-state" and have the same configuration parameters for itself and all dependencies. *Kensa*'s network proxy ensures that all inputs received in the production container are also forwarded to the debug container. However, any non-determinism from other sources (e.g. thread interleaving, random numbers, reliance on timing) may limit *Kensa*'s ability to faithfully reproduce an execution. While our current prototype version does not handle these, we believe there are several existing techniques that can be applied to tackle this problem in the context of live debugging. However, as can be seen in our case-studies above, unless there is significant non-determinism, the bugs will still be triggered in the replica, and can hence be debugged. Approaches like statis-

tical debugging [39], can be applied to localize bug. *Kensa* allows debugger to do significant tracing of synchronization points, which is often required as an input for constraint solvers [25, 27], which can go through all synchronization orderings to find concurrency errors. We have also tried to alleviate this problem using our divergence checker (Section 3.4)

Distributed Services: Large-scale distributed systems are often comprised of several interacting services such as storage, NTP, backup services, controllers and resource managers. *Kensa* can be used on one or more containers and can be used to clone more than one communicating . Based on the nature of the service, it may be (a). Cloned, (b). Turned off or (c). Allowed without any modification. For example, storage services supporting a replica need to be cloned or turned off (depending on debugging environment) as they would propagate changes from the debug container to the production containers. Similarly, services such as NTP service can be allowed to continue without any cloning as they are publishsubscribe broadcast based systems and the debug container cannot impact it in anyway. Furthermore, instrumentation inserted in the replica, will not necessarily slowdown all services. For instance, instrumentation in a MySQL query handler will not slowdown file-sharing or NTP services running in the same container.

8. Related Work

Record and Replay Systems: Record and Replay [6, 19, 29, 30, 52] has been an active area of research in the academic community for several years. These systems offer highly faithful re-execution in lieu of performance overhead. For instance, ODR [6] reports 1.6x, and Aftersight [15] reports 5% overhead, although with much higher worst-case overheads. *Kensa* avoids run-time overhead, but its cloning suspend time may be viewed as an amortized cost in comparison to the overhead in record-replay systems. Among record and replay systems, the work most closely related to ours is Aftersight [15]. Aftersight records a production system and replays it concurrently in a parallel VM. While both Aftersight and *Kensa* allow debuggers an almost real-time diagnosis facility, Aftersight suffers from recording overhead in the production VM. Additionally, it needs the diagnosis VM to either catch up with the production VM, which further slows down the application, or to allow it to proceed with divergence. The average slow-down in Aftersight is 5% and can balloon upto 31% to 2.6x for worst-case scenario. VARAN [31] is an N-version execution monitor that maintains replicas of an existing app, while checking for divergence. *Kensa*'s debug containers are effectively replicas: however, while VARAN replicates applications at the system call level, *Kensa*'s lower overhead mechanism does not impact the performance of the master (production) app. Unlike lower-level replay based systems, *Kensa* tolerates a greater amount of divergence from the original application: i.e., the replica may continue to run even if the analysis slightly modifies it.

Real-Time techniques: This is a category of approaches which attempt to do real-time diagnosis. Chaos Monkey [9] uses fault injection in real production systems to do fault tolerance testing. It randomly injects time-outs, resource hogs etc. in production systems. This allows Netflix to test the robustness of their system at scale, and avoid large-scale system crashes. Another approach called AB Testing [22] probabilistically tests updates or beta releases on some percentage of users, while letting the majority of the application users work on the original system. AB Testing allows the developer to understand user-response to any new additions to the software, while most users get the same software. Unlike *Kensa*, these approaches are restricted to software testing and directly impact the user.

Live Migration & Cloning Live migration of virtual machines facilitates fault management, load balancing, and low-level system maintenance for the administrator. Most existing approaches use a *pre-copy* approach that copies the memory state over several iterations, and then copies the process state. This includes hypervisors such as VMWare [45], Xen [16], and KVM [35]. VM Cloning, on the other hand, is usually done offline by taking a snapshot of a suspended/ shutdown VM and restarting it on another machine. Cloning is helpful for scaling out applications, which use multiple instances of the same server. There has also been limited work towards live cloning. For example Sun et al. [49] use copy-on-write mechanisms, to create a duplicate of the target VM without shutting it down. Similarly, another approach [28] uses live-cloning to do cluster-expansion of systems. However, unlike *Kensa*, both these approaches starts a VM with a new network identity and may require re-configuration of the duplicate node.

9. Conclusion & Future Work

Kensa is a novel framework that uses redundant cloud resources to debug production SOA applications in real-time. It can be combined with several existing bug diagnosis technique to localize errors. Compared to existing monitoring solutions, which have focused on reducing instrumentation overhead, our tool is able to avoid any performance slowdown at all, at the same time potentially allow significant monitoring for the debugger.

In the future, we will explore: (1) Applications: we aim to apply our system to real-time intrusion detection and statistical debugging. (2) Analysis: we wish to define "real-time" data analysis techniques for traces and instrumentation done in *Kensa*. (3) We plan to reduce the suspend time of live cloning, by utilizing several recent works in live migration. We will make *Kensa* available on GitHub for use by other researchers and practitioners. For each of the 16 faults studied in our case study, we will also release a docker container (with README) that can be launched to trigger the bug.

References

[1] Linux ipc pipes. http://man7.org/linux/man-pages/man7/pipe.7.html.

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

[3] Network namespaces. https://lwn.net/Articles/580893/.

[4] Petstore a sample java platform, enterprise edition reference application. http://www.oracle.com/technetwork/java/petstore1-1-2-136742.html.

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

[6] G. Altekar and I. Stoica. ODR: output-deterministic replay for multicore debugging. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, pages 193–206. ACM, 2009.

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

[8] P. Barham, A. Donnelly, R. Isaacs, and R. Mortier. Using Magpie for request extraction and workload modelling. In *OSDI*, volume 4, pages 18–18, 2004.

[9] C. Bennett and A. Tseitlin. Netflix: Chaos Monkey released into the wild. netflix tech blog, 2012.

[10] D. Bernstein. Containers and cloud: From lxc to docker to kubernetes. *IEEE Cloud Computing*, 1(3):81–84, 2014.

[11] S. M. Blackburn, R. Garner, C. Hoffmann, A. M. Khang, K. S. McKinley, R. Bentzur, A. Diwan, D. Feinberg, D. Frampton, S. Z. Guyer, et al. The dacapo benchmarks: Java benchmarking development and analysis. In *ACM Sigplan Notices*, volume 41, pages 169–190. ACM, 2006.

[12] C. Boettiger. An introduction to docker for reproducible research. *ACM SIGOPS Operating Systems Review*, 49(1): 71–79, 2015.

[13] D. Borthakur. Hdfs architecture guide. *HADOOP APACHE PROJECT http://hadoop. apache. org/common/docs/current/hdfs design. pdf*, page 39, 2008.

[14] J. L. Carlson. *Redis in Action*. Manning Publications Co., 2013.

[15] J. Chow, T. Garfinkel, and P. M. Chen. Decoupling dynamic program analysis from execution in virtual environments. In *USENIX 2008 Annual Technical Conference on Annual Technical Conference*, pages 1–14, 2008.

[16] C. Clark, K. Fraser, S. Hand, J. G. Hansen, E. Jul, C. Limpach, I. Pratt, and A. Warfield. Live migration of virtual machines. In *Proceedings of the 2nd conference on Symposium on Networked Systems Design & Implementation-Volume 2*, pages 273–286. USENIX Association, 2005.

[17] U. Deshpande and K. Keahey. Traffic-sensitive live migration of virtual machines. *Future Generation Computer Systems*, 2016.

[18] DockerHub. Build ship and run anywhere. https://www.hub.docker.com/.

[19] G. W. Dunlap, S. T. King, S. Cinar, M. A. Basrai, and P. M. Chen. Revirt: Enabling intrusion analysis through virtual-machine logging and replay. *ACM SIGOPS Operating Systems Review*, 36(SI):211–224, 2002.

[20] F. C. Eigler and R. Hat. Problem solving with systemtap. In *Proc. of the Ottawa Linux Symposium*, pages 261–268. Citeseer, 2006.

[21] F. C. Eigler, V. Prasad, W. Cohen, H. Nguyen, M. Hunt, J. Keniston, and B. Chen. Architecture of systemtap: a linux trace/probe tool. 2005.

[22] B. Eisenberg and J. Quarto-vonTivadar. *Always be testing: The complete guide to Google website optimizer*. John Wiley & Sons, 2009.

[23] U. Erlingsson, M. Peinado, S. Peter, M. Budiu, and G. Mainar-Ruiz. Fay: Extensible distributed tracing from kernels to clusters. *ACM Trans. Comput. Syst.*, 30(4):13:1–13:35, Nov. 2012. ISSN 0734-2071. . URL `http://doi.acm.org/10.1145/2382553.2382555`.

[24] W. Felter, A. Ferreira, R. Rajamony, and J. Rubio. An updated performance comparison of virtual machines and linux containers. In *Performance Analysis of Systems and Software (ISPASS), 2015 IEEE International Symposium On*, pages 171–172. IEEE, 2015.

[25] C. Flanagan and P. Godefroid. Dynamic partial-order reduction for model checking software. In *Proceedings of the 32Nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '05, pages 110–121, New York, NY, USA, 2005. ACM. ISBN 1-58113-830-X. . URL `http://doi.acm.org/10.1145/1040305.1040315`.

[26] M. Furman. *OpenVZ Essentials*. Packt Publishing Ltd, 2014.

[27] M. K. Ganai, N. Arora, C. Wang, A. Gupta, and G. Balakrishnan. Best: A symbolic testing tool for predicting multi-threaded program failures. In *Proceedings of the 2011 26th IEEE/ACM International Conference on Automated Software Engineering*, ASE '11, pages 596–599, Washington, DC, USA, 2011. IEEE Computer Society. ISBN 978-1-4577-1638-6. . URL `http://dx.doi.org/10.1109/ASE.2011.6100134`.

[28] A. Gebhart and E. Bozak. Dynamic cluster expansion through virtualization-based live cloning, Sept. 10 2009. URL `https://www.google.com/patents/US20090228883`. US Patent App. 12/044,888.

[29] D. Geels, G. Altekar, P. Maniatis, T. Roscoe, and I. Stoica. Friday: Global comprehension for distributed replay. In *NSDI*, volume 7, pages 285–298, 2007.

[30] Z. Guo, X. Wang, J. Tang, X. Liu, Z. Xu, M. Wu, M. F. Kaashoek, and Z. Zhang. R2: An application-level kernel for record and replay. In *Proceedings of the 8th USENIX conference on Operating systems design and implementation*, pages 193–208. USENIX Association, 2008.

[31] P. Hosek and C. Cadar. Varan the unbelievable: An efficient n-version execution framework. In *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '15, pages 339–353, New York, NY, USA, 2015. ACM. ISBN 978-1-4503-2835-7. . URL `http://doi.acm.org/10.1145/2694344.2694390`.

[32] G. Jin, L. Song, X. Shi, J. Scherpelz, and S. Lu. Understanding and detecting real-world performance bugs. In *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Imple-*

mentation, PLDI '12, pages 77–88, New York, NY, USA, 2012. ACM. ISBN 978-1-4503-1205-9. . URL http://doi.acm.org/10.1145/2254064.2254075.

[33] B. Kasikci, B. Schubert, C. Pereira, G. Pokam, and G. Candea. Failure sketching: A technique for automated root cause diagnosis of in-production failures. In *Proceedings of the 25th Symposium on Operating Systems Principles*, SOSP '15, pages 344–360, New York, NY, USA, 2015. ACM. ISBN 978-1-4503-3834-9. . URL http://doi.acm.org/10.1145/2815400.2815412.

[34] D. G. Kendall. Stochastic processes occurring in the theory of queues and their analysis by the method of the imbedded markov chain. *Ann. Math. Statist.*, 24(3):338–354, 09 1953. . URL http://dx.doi.org/10.1214/aoms/1177728975.

[35] A. Kivity, Y. Kamay, D. Laor, U. Lublin, and A. Liguori. kvm: the linux virtual machine monitor. In *Proceedings of the Linux Symposium*, volume 1, pages 225–230, 2007.

[36] K. Kolyshkin. Virtualization in linux. *White paper, OpenVZ*, 3: 39, 2006.

[37] O. Laadan, N. Viennot, and J. Nieh. Transparent, lightweight application execution replay on commodity multiprocessor operating systems. In *ACM SIGMETRICS Performance Evaluation Review*, volume 38, pages 155–166. ACM, 2010.

[38] A. Lakshman and P. Malik. Cassandra: a decentralized structured storage system. *ACM SIGOPS Operating Systems Review*, 44(2):35–40, 2010.

[39] B. R. Liblit. *Cooperative Bug Isolation*. PhD thesis, University of California, Berkeley, Dec. 2004.

[40] J.-G. Lou, Q. Lin, R. Ding, Q. Fu, D. Zhang, and T. Xie. Software analytics for incident management of online services: An experience report. In *Automated Software Engineering (ASE), 2013 IEEE/ACM 28th International Conference on*, pages 475–485. IEEE, 2013.

[41] S. Lu, Z. Li, F. Qin, L. Tan, P. Zhou, and Y. Zhou. Bugbench: Benchmarks for evaluating bug detection tools. In *Workshop on the evaluation of software defect detection tools*, volume 5, 2005.

[42] D. Merkel. Docker: lightweight linux containers for consistent development and deployment. *Linux Journal*, 2014(239):2, 2014.

[43] A. Mirkin, A. Kuznetsov, and K. Kolyshkin. Containers checkpointing and live migration. In *Proceedings of the Linux Symposium*, pages 85–92, 2008.

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

[45] M. Nelson, B.-H. Lim, G. Hutchins, et al. Fast transparent migration for virtual machines. In *USENIX Annual Technical Conference, General Track*, pages 391–394, 2005.

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

[47] S. Newman. *Building Microservices*. " O'Reilly Media, Inc.", 2015.

[48] H. Patil, C. Pereira, M. Stallcup, G. Lueck, and J. Cownie. Pinplay: a framework for deterministic replay and reproducible analysis of parallel programs. In *Proceedings of the 8th annual IEEE/ACM international symposium on Code generation and optimization*, pages 2–11. ACM, 2010.

[49] Y. Sun, Y. Luo, X. Wang, Z. Wang, B. Zhang, H. Chen, and X. Li. Fast live cloning of virtual machine based on xen. In *Proceedings of the 2009 11th IEEE International Conference on High Performance Computing and Communications*, HPCC '09, pages 392–399, Washington, DC, USA, 2009. IEEE Computer Society. ISBN 978-0-7695-3738-2. . URL http://dx.doi.org/10.1109/HPCC.2009.97.

[50] P. Svärd, B. Hudzia, S. Walsh, J. Tordsson, and E. Elmroth. Principles and performance characteristics of algorithms for live vm migration. *ACM SIGOPS Operating Systems Review*, 49(1):142–155, 2015.

[51] P. Thomson and A. F. Donaldson. The lazy happens-before relation: Better partial-order reduction for systematic concurrency testing. In *Proceedings of the 20th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP 2015, pages 259–260, New York, NY, USA, 2015. ACM. ISBN 978-1-4503-3205-7. . URL http://doi.acm.org/10.1145/2688500.2688533.

[52] K. Veeraraghavan, D. Lee, B. Wester, J. Ouyang, P. M. Chen, J. Flinn, and S. Narayanasamy. Doubleplay: Parallelizing sequential logging and replay. *ACM Trans. Comput. Syst.*, 30(1):3:1–3:24, Feb. 2012. ISSN 0734-2071. . URL http://doi.acm.org/10.1145/2110356.2110359.

[53] Y. Wang, H. Patil, C. Pereira, G. Lueck, R. Gupta, and I. Neamtiu. Drdebug: Deterministic replay based cyclic debugging with dynamic slicing. In *Proceedings of annual IEEE/ACM international symposium on code generation and optimization*, page 98. ACM, 2014.

[54] M. G. Xavier, M. V. Neves, F. D. Rossi, T. C. Ferreto, T. Lange, and C. A. De Rose. Performance evaluation of container-based virtualization for high performance computing environments. In *2013 21st Euromicro International Conference on Parallel, Distributed, and Network-Based Processing*, pages 233–240. IEEE, 2013.

[55] D. Yuan, Y. Luo, X. Zhuang, G. R. Rodrigues, X. Zhao, Y. Zhang, P. U. Jain, and M. Stumm. Simple testing can prevent most critical failures: An analysis of production failures in distributed data-intensive systems. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*, pages 249–265, 2014.