Imperial College London
Department of Computing

Highly Reliable Upgrading of Software Containers

by
Samira Rabbanian (sr1213)

Submitted in partial fulfilment of the requirements for the MSc Degree in Computing Science of Imperial College London

September 2014

Abstract

This is where the abstract will be

Contents

List of Figures

Chapter 1

Introduction

1.1 Motivation

Software development is a complex process where errors can easily cause instability, unreliability and incorrect behavior. The most significant problems are; the risks inherent with software upgrade, and the cost of software development and support.

Reliable Software Upgrade - in safety critical industries such as air traffic control or healthcare reliable software is essential for ensuring that lives are not put at risk due to software errors or crashes. In E-commerce reliable software is critical for maximizing the return on investment of software development and deployment. Unreliable software has many damaging impacts such as loss of profit due to unavailability of revenue generating services, damaging reputation of the service provider often with long-term impact, risk of incorrect transactions for example in financial sectors, risk of security vulnerability and costs to fix and monitor errors.

All companies and organizations use software applications to run their business, deliver services or manufacture products. Therefore, software upgrade plays a key role to maintain reliable applications that are free from errors and security vulnerabilities. Internet facing applications are continuously exposed to an ever increasing set of security threats as listed by the Open Web Application Security Project foundation (OWASP)[1] . These threats include injection attacks, broken authentication and session management, cross-site scripting or cross-site request forgery. To avoid such attacks software must be appropriately patched against vulnerabilities. Once a software security patch has been released, the vulnerability is in the public domain and can therefore be used maliciously by attackers. It is therefore important that software is updated to remove these vulnerabilities. Furthermore, software is updated to enable new services or functionality to be provided to users. Such updates are often important for companies to ensure they are providing the most reliable, competitive and profitable services to their customers. However, software upgrade itself may cause unreliability by introducing new errors, and security vulnerabilities, as stated by Fred Brooks in The Mythical Man-Month.

"The fundamental problem with program maintenance is that fixing a defect has a substantial (20-50%) chance of introducing another. So the whole process is two steps forward and one step back"

Fred Brooks, Turing Award Winner (1999) - The Mythical Man-Month

Recent studies have demonstrated that 14.8% to 25% of software upgrades are incorrect and have made impacts on end-users [46]. Many different systems and techniques have been

[1] https://www.owasp.org/index.php/Top_10_2013-Top_10

proposed to support reliable software upgrade. Currently, these systems are either specific for particular programming languages or are complex, heavyweight and expensive.

Reducing Cost of Software Development and Support - studies have shown that fixing software errors and bugs after delivery is often 100 times more expensive than finding and fixing them during the development process. Moreover, fast feedback is critical in reducing the cost of bugs in all phases ([47], [48]). Figure 1.1 shows that as the feature development lifecycle progresses the cost of bugs significantly increases.

Thus, there has been a continuous introduction of new techniques to reduce the feedback cycle. Initially code review, pair programming and continuous integration were adopted. More recently these approaches have been further developed into new techniques such as continuous delivery and immutable infrastructures. However, these more recent techniques are challenging to adopt. In a typical software development team testing is performed both by developers and software testers on a testing environment that mirrors production. For continuous delivery and immutable infrastructures to be adopted this testing environment should be updated with the latest software changes on a regular cycle, however, each update takes a long time and prevents any testing on the testing environment. The frequency of update is therefor in conflict with the availability of the testing environment. Also the extent to which the infrastructure is immutable is also in conflict as more immutable an infrastructures is the longer the deployment typically takes as more activities are performed during the deployment.

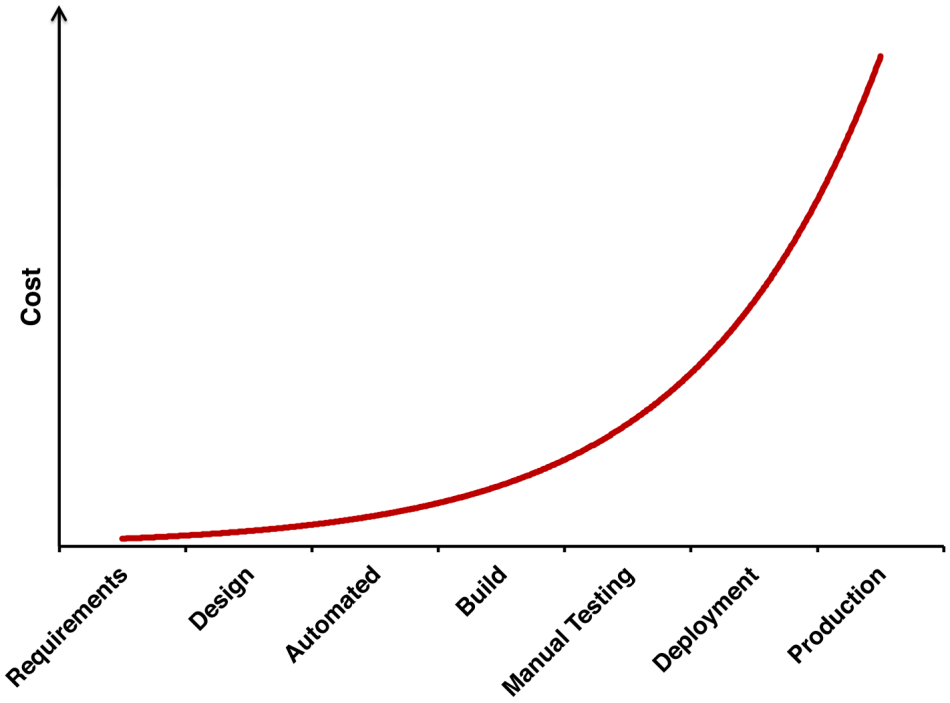

Figure 1.1: Cost of bugs during feature development life Cycle

To date, there is no existing system considered robust enough to be used widely for fully automated continuous delivery and reliable software upgrades from development into production.

1.2 Objectives

The main objectives of this project were to implement a proxy that adds no significant overhead while:

- supporting reliable dynamic software update by seamlessly routing requests to the latest

application version

- supporting seamless rollback to the previous version of the application when the upgraded version behaves incorrectly
- managing application clusters using container virtualization
- being reliable, robust and stable under heavy load

1.3 Contributions

This report presents a novel, lightweight, robust and reliable proxy that supports dynamic software upgrade with no downtime and automatic rollback in response to application failure. The main features provided are:

- Seamless dynamic software upgrade using four different upgrade mechanisms:
  - Instant upgrade - immediate upgrade to the latest version of an application
  - Session upgrade - upgrading only new sessions to the latest version of an application
  - Gradual upgrade - gradual upgrade of clients to the latest version of an application
  - Concurrent upgrade - routing requests to multiple versions of an application simultaneously and returning the response from the latest version of the application that is behaving correctly
- Automatic rollback when an upgraded version does not behave correctly.
- Two mechanisms to manage software versions:
  - Distinct TCP sockets for each version of an application. The proxy is configured to communicate with each application version on a separate IP and port combination.
  - Distinct Docker images for each version of an application. The proxy manages a Docker container based on each application version using a different image or image tag. Each application version is therefore isolated in a separate user space sand-boxed process providing resource isolation while sharing a single operating system kernel.
- Two mechanisms to configure the proxy:
  - File base API in JSON format which is parsed when the proxy loads. All features of the TCP based application version configuration and Docker image based configuration are supported.
  - A REST API in JSON / HTTP format which can be used to query, add or remove application versions. All features of the TCP based application version configuration and Docker image based configuration are supported.
- Extensive support for starting, stopping and updating Docker containers and images, including support for all Docker container and image features.

1.4 Outline of the Report

Related Work Chapter - Presents different approaches and technologies used in the past decades to achieve and maintain highly available software and reliable dynamic software updates.

Technical Background Chapter -Explains and justifies the technologies used to implement the dynamic software update proxy.

Design Chapter - DESIGN

Implementation Chapter - IMPLEMENTATION

Evaluation Chapter - Presents the evaluations of the dynamic software update proxy to test its performance, functional correctness and robustness while proxying requests to real applications.

Conclusion & Future Chapter - Presents the overall discussion of the results and the achievements of the project. The limitations of the system are also presented. Finally, possible future work for extensions of the dynamic software upgrade proxy.

Appendices - Appendices

Chapter 2

Related Work

Software reliability is compromised of the following main activities:

- Error prevention
- Fault detection and removal
- Fault resilience

The Institute of Electrical and Electronics Engineers (IEEE) defines reliability as "the ability of a system or component to perform its required functions under stated conditions for a specified period of time". This section of the report describes different approaches and technologies used to achieve and maintain software reliability, specifically to support the activities mentioned above.

2.1 Software High Availability

System availability is the percentage of time the system is available to users. A highly available application is fault tolerant and reliable. A reliable system ensures that failure of a single component in the system does not cause failure of the entire application. In a reliable system data updates are not lost and the most recent data is available within acceptable tolerances.

Software Update

Software update typically requires restarting of the system that is to be updated. This can happen in two ways either the system is stopped, the update is applied and the system is restarted. Alternatively the update is applied while the system is still running and then the application is restarted. In the second scenario the system's binary code on persistent storage (*e.g.* disk) is updated while the system is running in memory before it is restarted and the new binary code is loaded into memory [1].

In both cases performing the code update requires the system to be restarted resulting in a period of unavailability. Many applications are performing critical life saving functions and cannot be interrupted, for example, nuclear control systems, air-traffic controllers or hospital life-support software. In addition, other applications have an extremely high downtime cost such as E-commerce, telecommunications and banking systems [2]. It is therefore extremely important to support seamless upgrade where the system stays fully available during the upgrade.

Dynamic software update also known as live software update is the process of updating parts of a program without having to interrupt its execution. Dynamic system updates can be performed at either a hardware level or a software level. Hardware based dynamic updating are based on hardware redundancy. Systems such as Tandem Nonstop [3] used in the healthcare and banking industries

support dynamic hardware updating and resilience to hardware failure. Detailed techniques and approaches that support dynamic software update are described in section 2.4.

2.2 Reliable Software Architectures

There are multiple architectures that increase high availability and support dynamic software update. These include application clustering, active-passive architectures, data replication and clustered databases.

2.2.1 Application Clustering

A common practice to increase availability and fault tolerance is using a cluster of multiple application (or component) instances [4]. Most commonly a load balancer is used to distribute the load between nodes in the cluster by pushing requests to each node [10]. For example, a load balancer can receive requests for a web application and distribute the requests between different web services so that the load is distributed across all web services. Alternatively, the nodes within the cluster can pull requests as each node becomes available. For instance, a request queue on a Message Oriented Middleware (MOM) or a message broker such as ActiveMQ[1] or RabbitMQ[2] can be used to allow nodes to pull requests (Section 3.2).

When application requests are balancing across multiple nodes, if a single application (or component) instance fails, other nodes in the cluster will still be available to process requests. This clustering approach therefore improves overall application availability. In addition, a clustering approach provides an increased ability to handle spikes in load. Spikes in load are spread across multiple nodes reducing their impacts on any given server. Moreover, an application that is deployed as a cluster is typically easy to scale horizontally by adding additional nodes because it has been designed to run as a cluster [4].

One common approach widely used in industry to support dynamic software update is a content switching load balancer (*e.g.* F5[3], Citrix NetScaler[4]) in front of an application cluster [10]. To perform such an upgrade the cluster is typically split into two halves called A and B. First all load is drained from A by preventing any new requests (*e.g.* business transactions) from being processed. Once A has completed processing all existing requests, it is upgraded and restarted. Now A can start to accept new requests and B is drained and subsequently upgraded. However, this approach is slow and not free from risk as it requires manual interaction with the load balancer to split the cluster and re-route traffic between each step.

2.2.2 Active-Passive Architecture

In addition to clustering another common approach to increase software reliability is to have an active-passive architecture where there are two application instances (or two application clusters). One of the application instances (or application clusters) is actively receiving requests while the other is passive and it is not processing any load [5]. This arrangement improves reliability because if the active instance starts to behave incorrectly or crashes, all requests can be immediately routed to the passive instance, resulting in the passive instance becoming active.

A content switching router can be used to automate the routing of requests to the passive instance as soon as it detects the active instance is no longer correctly processing requests. This architecture can also be used to support reliable dynamic software updates. The passive instance

[1] http://activemq.apache.org/
[2] http://www.rabbitmq.com/
[3] https://f5.com/
[4] http://www.citrix.com/products/netscaler-application-delivery-controller/overview.html

can be updated first and requests can slowly start to be routed to the passive instance. During this period if the software update applied to the passive instance fails or causes it to crash, all requests can be immediately routed back to the active instance. However, this approach is wasteful since a complete backup application instance or backup application cluster is required. In addition, this approach is slow and it is not free from risk as it requires manual interaction with the load balancer to re-route traffic.

2.2.3 Data Replication

Data replication can also be used to increase software reliability. Conventional SQL Relation Database Management Systems (RDBMS) typically do not operate in a cluster [6]. This is because managing ACID transactions and locking across multiple nodes in a cluster requires commit and rollback to be coordinated across multiple nodes. In addition, consistencies, such as foreign key relationships, must be managed across multiple nodes where the parent and child of the foreign key relationship may be on separate nodes. Although clustered RDBMS do exist, it is expensive, complex and can be error prone. Alternatively, to improve software reliability data can be replicated to a backup database. This works in a similar way to the active-passive architecture where one database is used to process active requests and a second database is only receiving replicated changes and not directly responding to requests. If the active database fails or crashes the passive database can immediately start processing requests and becomes the new active database [6].

To support dynamic updating of databases, data replication can be used where a backup database receives a replicated copy of the changes made to the primary database. This supports dynamic update in a similar way to the active-passive architecture where software updates can be initially applied to the replicated database; once this is completed, the passive database can replace the active database (and the replication direction is reversed). If the software update fails or causes the database to crash, the non-updated database is still available and can immediately start processing requests again.

2.2.4 Clustered Databases

As the requirements for high availability and high scalability have increased, there has been a shift from using conventional SQL RDBMS to clustered NoSQL databases. In addition, shared database integration is no longer a common technique as it is now widely recognized to break encapsulation and introduce high-coupling. Hence, it is no longer as important for databases to provide strong data integrity. NoSQL databases are designed with clustering and high availability as a priority over strong data integrity and locking [7]. Therefore, NoSQL databases run in large clusters and replicate data between multiple nodes making them very resilient to failure of one or more nodes. In addition, they are also able to handle rapid increases in load by spreading the load across multiple nodes in the cluster.

Since NoSQL databases do not provide strong data integrity, it is typically much simpler to dynamically apply updates. For example a document database may store documents in the form of JavaScript Object Notation (JSON)[5]. When the data model is updated by adding or removing fields, no changes to the database are required. Instead applications reading or writing documents are expected to handle different document formats in a flexible way. If such a change was required in a RDBMS, the table structure would need to be modified. The modification is a highly risky activity in a live database and would typically require application downtime and a full database backup.

[5] `http://www.json.org/`

2.3 Reliable Software Techniques

In addition to architectures that support software reliability, there are also techniques that are commonly used to increase reliability of applying software updates. These include automated deployment, configuration as code and continuous delivery.

2.3.1 Automated Deployment

Automated deployment is used to reduce the cost and the risk associated to installing or updating software. Such an approach is particularly important for cluster or active-passive architectures where the same application must be installed or updated repeatedly. Automated deployment involves using fully scripting application installation or update processes including any operating system (OS) installation, package installation, application installation and configuration [8]. There are several tools that are commonly used for automated deployment such as Puppet[6], Chef[7], Ansible[8] and Salt[9].

2.3.2 Configuration As Code

Configuration as code is a technique used to ensure all environment configuration and settings for an application installation or update are written as code often in the form of an automated deployment script. Such an approach ensures that applications are deployed with the correct environment configuration and that the development team and infrastructure team can easily agree the required configuration [9]. This approach reduces the risk of deployment by ensuring configuration settings can be tested and reapplied identically every time. In addition, such an approach allows configuration changes to be versioned and changed in-line with the application code or any software updates guaranteeing that the correct configuration for a given update is applied.

2.3.3 Continuous Delivery

Continuous delivery is a software development approach that ensures application updates can be deployed to production at any point throughout the development life cycle. To support continuous delivery a pipeline is typically used to promote application updates automatically through several stages from initial development through to production [9]. Continuous delivery supports reliable software update by ensuring that each update has been applied to multiple stages prior to reaching production.

Furthermore, continuous delivery promotes small incremental updates that are deployed on a regular bases reducing the risk from any given update. Continuous delivery also increases software reliability by reducing the cost of applying updates. If an update is extremely easy to deploy, then any defect that has been deployed into production can be more easily fixed by a subsequent deployment. However, continuous delivery from development all the way into production is rare with most companies only managing to promote application updates as far as pre-production. This is because performing automated deployment via continuous delivery into production is considered too risky. This fact reduces many of the benefit that continuous delivery provide.

[6] https://puppetlabs.com/
[7] http://www.getchef.com/chef/
[8] http://www.ansible.com/home
[9] http://www.saltstack.com/enterprise/

2.3.4 Immutable Infrastructure

A common problem in infrastructures is that manual changes introduce state into servers and network components. This state is unrecorded, unknown and often unreproducible causing issues when resolving problems or attempting to rebuild the environment, for example, after a critical failure. An immutable infrastructure is an infrastructure that is made of components that are replaced (or completely rebuilt) on every deployment. The advantage of using the immutable infrastructure technique is that state is siloed and can not leak through out the entire infrastructure. The boundaries between layers storing state and the layers that are ephemeral, which may not be running the next minute, are clearly drawn and no leakage can possibly happen between those layers. State is stored in the database or in deployment scripts (also called infrastructure-as-code) and it is not scattered through out the architecture.

2.4 Dynamic Software Update Systems

Software updates are an important part of maintaining a long-lived system with new software enhancements, fixes and modifications being released on a continuous basis. In the past decades several systems have been developed to support highly reliable dynamic software upgrades. Each of these systems uses different approaches to change the code and the data of a program from an old version to a new version.

2.4.1 Code Update

Systems such as Ksplice [11], OPUS [12] , DynaMOS [13], and POLUS [14] replace the old code with a small piece of code, called trampoline. Trampoline executes the replaced instruction and then will jump to the function's new version. One of the main weaknesses of using this mechanism is that the trampoline requires a writable code segment, which makes the application vulnerable to code injection attack [11].

Ginseng [15] and K42 [16] systems use indirection instead of trampolines. Ginseng uses binary rewriting to direct function calls into calls via function pointers, while K42's OS uses indirection through an object translation table. In the aforementioned systems updates occur by redirecting indirection targets to the new version. However, using this technique can add overhead to normal execution process.

Dynamic software update systems mentioned above affect code updates at the granularity of individual functions or objects. However, those systems are not capable of updating functions that contain event-handling loops or functions like *main* that rarely end. Hence, systems such as Kitsune [19], UpStare [17], Ekiden [18], which focus on updating the whole program rather than individual functions have been developed. UpStare uses a stack reconstruction update mechanism. In this mechanism the running application automatically unrolls the call stack when an update occurs, while saving all stack frames. It then modifies the call back by replacing old functions with their new version, and at the same time mapping data structures in the old frames to their new versions. In contrast, Kitsune and Ekiden both use a manual approach by relying on the programmer to migrate control to the correct equivalent point in the new version of the program.

2.4.2 Data Update

Most dynamic software update systems handle the data update by using object replacement. The system or the developer allocates replacement objects and initialize them using data from the old version. Ginseng uses type-wrapping approach. In this approach the programs are compiled so that *struct*s have an added version field and extra "slop" space to allow for future extensions. Transformation of old objects will be initiated by inserting calls to mediator functions which access

updated objects. Ksplice and DynaMOS do not change the old objects but allocate shadow data structures containing only the new fields. Shadow data structures have the advantage of changing fewer functions by an update. When a new field is added to a *struct*, only the code using that field is affected but not all the code that uses the *struct*.

2.5 Dynamic Software Update Safety

Choosing when to safely apply updates has been one of the main concerns of the prior work on dynamic software updating. Some solutions rely on no updates to active code, *i.e.* no thread is running that code, and no thread's stack refers to it, ([20], [11], [16], [21]). This restriction reduces post-update errors but it does not eliminate them, and additionally imposes strong restrictions on the form of an update and how quickly it can be applied. Moreover, some researchers suggest that no updated code should access data generated prior to the update being applied ([23], [22], [24]). This technique ensures that updates with type difference do not pose a threat to type safety. The final approach suggested by researchers is using transactions in a distributed or local context to enforce stronger timing constrains ([28], [27], [29]). However, it is currently been shown that update timing may not be a main concern and a few programmer-designed update points are typically sufficient to determine safe and timely update states ([23], [19], [25], [26]).

2.6 Software Update Using Multi-Version Framework

The idea of N-version programming, also known as multi-version programming, was originally introduced in 1970s. This method is defined as the independent generation of more than two functionally equivalent programs. Separate developers develop each version of the program all using the same initial specification. These versions will be run concurrently in the application environment. Each version will handle identical inputs and the output of all the versions will be collected and the voting scheme are used to decide which version(s) of the program behave correctly [35]. N-version programming technique was originally proposed as a method of providing reliable and fault tolerance software. This methodology has inspired many researchers to propose new techniques for development of reliable software applications

In 1999 Cook *et al.* introduced a multi-version framework called Hercules for the highly reliable upgrading of software components [30]. In this framework instead of removing the old version of a component, multiple versions of the same component are kept running in parallel and the behavior of each version is utilized. This approach allows the system integrity to be maintained in the presence of bugs introduced due to the new version of the component. Therefore, Hercules ensures reliability by keeping existing versions of the component running and the old version is only fully removed when the new version has satisfied all its rules. Additionally, Berger and Zorn proposed a replica framework each with a different randomized layout of objects within the heap to provide probabilistic memory safety [31]. Furthermore Veeraraghavan et al. suggested running multiple replicas with complementary thread schedules to avoid errors in multi-threaded programs [32].

More recently, Hosek *et al.* proposed a novel framework called Mx, which takes advantage of the idle resources made available by multi-core platforms, and allows applications to survive crash errors introduced by incorrect software updates ([33], [34]). Similar to Hercules, Mx achieves reliability by running the old and new version of an application concurrently. The fundamental difference between the two frameworks is that Hercules requires the programmer to define the functionality of each component version, whilst Mx targets crash bugs and is fully automated. Additionally, in the latter system all versions are live at all times and when Mx detects that one of the versions is not behaving correctly or has crashed, the correctly behaving version is used to handle all software requests. This allows appropriate actions to be taken at a convenient moment; at this point, the incorrectly behaving version can be fixed or restarted.

Chapter 3

Technical Background

This chapter explains and justifies the technologies used to implement the dynamic software update proxy described in Chapter 4.

3.1 Hypertext Transfer Protocol

The Hypertext Transfer Protocol (HTTP) is a stateless application-level protocol for distributed hypertext and hypermedia information systems supporting the interlinking of text and multimedia. Hypertext, a term coined by Ted Nelson [45], is text which is not constrained to be linear and contains links to other texts. Hypermedia is an extension to hypertext adding support for multimedia, such as images, video and sound.

HTTP is the foundation protocol of the World Wide Web and it is widely used between different systems that need to exchange text and multimedia. The most common HTTP relationships are client - server and server - server. The most common client - server relationship is when a web browser (or user agent) browses a web site on a web server. A server - server relationship is when a web service requires information from another web service, a common example is OAuth[1] where a web service can act on behalf of a user by interacting with an OAuth web service provider. An important feature of HTTP is the typing and negotiation of data representation, allowing systems to be built independently of the data being transferred. This is often referred to REpresentational State Transfer (REST).

3.1.1 HTTP Requests

An HTTP request has the following format:

```
<Request> ::=   <Request-Line>
                <general-headers>
                |<request-headers>
                | <entity-headers> CRLF
                CRLF
                <opt-message-body>
```

<Request-Line> consists of a method, request URI and the protocol version. Method defines the HTTP functions such as GET, POST, PUT, DELETE that is performed on the resources identified by request URI .

[1] http://oauth.net/

<*general-headers*> provide general control information and they can be included in both requests and responses. *Transfer-Encoding* is an example of general headers which defines what (if any) type of transformation has been applied to the message body for its safe transfer between the sender and the recipient. For example if *Transfer-Encoding* is defined as "chunked", the data will be sent to the recipient in a series of "chunks".

<*request-headers*> allow the client to pass additional information about the request, and about the client itself, to the server.

<*entity-headers*> define optional and required meta-information about the request body and if the request has no body, provide information about the resource identified by the request. For example, *Content-Length* header indicates the size of the request body.

CRLF CRLF indicating a carriage return ('\r') followed by a line feed ('\n') is used to separate the header section from the body.

<*opt-message-body*> contains the data transmitted in the body of the request.

For example the following shows a simple HTTP request:

```
GET / HTTP/1.1
User-Agent: curl/7.30.0
Host: 127.0.0.1:1235
Accept: */*
```

3.1.2 HTTP Responses

An HTTP response has the following format:

```
<Response> ::= <Status-Line>
               <general-headers>
               |<response-headers>
               |<entity-headers> CRLF
               CRLF
               <opt-message-body>
```

<*Status-Line*> consists of the HTTP version followed by a numeric status code and its associated textual phrase. The status code indicates the action taken on the corresponding request. The status code has 5 groupings as follows:

- **1xx Informational** - request received continue processing *e.g.* 101 Switching Protocols
- **2xx Success** - request was successfully received, understood, and accepted *e.g.* 202 Accepted
- **3xx Redirection** - client actions required to complete the request *e.g.* 301 Moved Permanently
- **4xx Client Error** - request contains a syntax error or cannot be fulfilled *e.g.* 400 Bad Request
- **5xx Server Error** - server failed to fulfill the request *e.g.* 500 Internal Server Error

<*general-headers*> provide general control information and they can be included in both requests and responses. *Transfer-Encoding* is an example of general headers which defines what (if any) type of transformation has been applied to the message body for its safe transfer between the sender and the recipient. For example if *Transfer-Encoding* is defined as "chunked", the data will be sent to the recipient in a series of "chunks".

<*response-headers*> allow the server to pass additional information about the response and about the server itself to the client.

<*entity-headers*> define optional and required meta-information about the response body and if the response has no body, provide information about the resource returned in the response. For example, *Content-Length* header indicates the size of the response body.

CRLF CRLF indicating a carriage return ('\r') followed by a line feed ('\n') is used to separate the header section from the body.

<*opt-message-body*> contains the data transmitted in the body of the response.

For example the following shows a simple HTTP response:

```
HTTP/1.1 200 OK
Set-Cookie: dynsoftup=be69fecd-2c53-11e4-9f0f-28cfe9158b63;
Date: Mon, 25 Aug 2014 12:37:35 GMT
Server: Apache/2.2.22 (Debian)
X-Powered-By: PHP/5.4.4-14+deb7u12
X-Pingback: http://192.168.50.5:8081/xmlrpc.php
Vary: Accept-Encoding
Content-Length: 7467
Content-Type: text/html; charset=UTF-8

<!DOCTYPE html> ... </html>
```

3.2 Messaging Systems

Messaging systems allow two or more applications to exchange information in the form of messaging. Advanced Message Queuing Protocol (AMQP) [42], Java Message Service (JMS) [43] and Zero Message Queuing (ZeroMQ)[2] are the most common messaging standards.

3.2.1 Broker-Based Messaging Systems

AMPQ and JMS are popular examples of broker-based messing systems. A message broker (also called Message Oriented Middleware) is a physical component that handles the communications between different applications. Hence, in a broker-based messaging system instead of applications directly communicating with each other, they communicate with the message broker [44]. The advantage of using this architecture is that the applications do not need to know the location of other applications. They only need to be aware of the network address of the broker. The broker then routes the messages to the correct applications based on the business requirements using the message properties, queue name or routing key [42].

[2] http://zeromq.org/

In addition, a broker-based messaging system is more resistant to the application failure. This is because if an application fails, messages that are already in the broker will be retained. However, broker-based messaging systems require excessive amount of network communication. Moreover, since all the messages have to be passed through the broker, the broker can turn out to be a bottleneck in the system. Therefore, the broker can be utilized to 100% while other components of the system are under-utilized or even idle. Finally, the broker has to be managed and maintained separately to the applications sending and receiving messages. This breaks encapsulation and separation of concerns because the broker contains application specific configuration and logic. Therefore, if the application requirements change, both the broker and the application must be updated in a coordinated way. In addition, one broker often contains logic and queues for several different applications.

3.2.2 Zero Broker Messaging Systems

In a broker-less messaging system each application directly talks to other applications without any middleware, hence, there are no bottlenecks associated with these systems. The application can manage and maintain its own messaging infrastructure and so encapsulation and separation of concerns are increased.

ZeroMQ is a broker-less, language agnostic, lightweight asynchronous messaging library. Asynchronous I/O model of ZeroMQ asynchronous message-processing required for scalable multi-core applications[3].

ZeroMQ provide sockets that carry atomic messages across various transports such as Transmission Control Protocol (TCP) or User Datagram Protocol (UDP) and communication styles such as point-to-point or multicast. Unlike conventional sockets that only allow strict one-to-one, many-to-one, or in some cases one-to-many relationships, ZeroMQ sockets can be connected to multiple endpoints while simultaneously accepting incoming connections from multiple endpoints (many-to-many connection). ZeroMQ sockets support connection patterns such as request-reply (sending requests from a client to a web service or cluster of web services and receiving reply from each request sent), publish-subscribe (one-to-many distribution of data from a single publisher to multiple subscribers in a fan out manner), pipeline (distributing data to nodes arranged in a pipeline) and exclusive pair (connect one peer to exactly one other peer for inter-thread communications) patterns that is summarized in `http://api.zeromq.org/2-1:zmq-socket`.

3.3 Efficient Cluster Management

As described in Section 2.2 clustering supports dynamic software update and software high availability; however, clustering increases the cost of managing and maintaining the application due to the multiple application instances within the cluster. Virtualization is a technique widely used to reduce the cost of hardware, management and risk associated to clustering.

3.3.1 Virtualization

Virtualization is the separation of a resource or service from the underlying physical delivery of that resource or service. Virtualization can occur on multiple different infrastructure layers such as network, storage, server hardware, operating systems or applications. Virtual memory, for example, simulates additional memory above the memory that is physically available by using a swap file on hard disk [36]. Filesystems are also virtualized; for example, a Logical Volume Manager (LVM) maps multiple physical disks to logical pools of storage (volume groups). A filesystem can then be created on top of the logical volume within a logical pool (volume group). The filesystem can

[3] `http://zeromq.org/`

therefore be spread across multiple physical disks, be re-sized and or moved from one physical disk to another while I/O is happening to the file system[4].

The main advantage of virtualization is separation between the virtualized infrastructure and the physical infrastructure. This means that applications can continue to execute with no downtime even when physical hardware is replaced, fails or any other hardware maintenance is performed. In addition, physical resources can be pooled and combined then redistributed as required.

Hypervisor Virtualization

Operating system virtualization that is called hypervisor virtualization allows multiple guest operating systems to run on a single host system at the same time [37]. This type of virtualization can be either native based (type 1) or hosted based (type 2) [38]. Hosted based hypervisor virtualization uses an application that is installed on an OS such as VMware[5] or VirtualBox[6]. Native hypervisor based virtualization in contrast avoids the overhead of the host OS by running the virtualization layer directly on the host machine (bare-metal). The guest OS shares the hardware of the host computer such that each OS appears to have its own processor, memory and other hardware resources. Since hypervisor has direct access to the hardware resources, it is efficient and enables greater scalability, robustness and performance. In addition, a hypervisor can run the virtualization layer across multiple physical machines [38]. This allows new physical machines to be added or maintenance to be performed against existing physical machines transparently without affecting the host operating systems. (Figure 3.1).

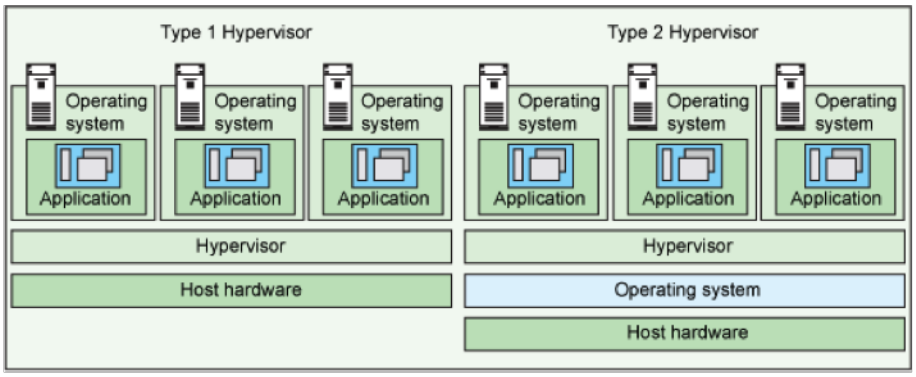

Figure 3.1: Hypervisor type 1 vs. type 2 [38]

Container Virtualization

Container virtualization is a lightweight operating system virtualization technique that instead of trying to run an entire guest OS, it isolates the guests, but does not virtualize the hardware [39] (Figure 3.2). Container virtualization is considerably more lightweight than hypervisor virtualization. It has been found to be as much as 40% less overhead using Docker based container virtualization compared to running full virtual machines on Amazon Elastic Compute Cloud (EC2)[7]. Each container can be treated like a regular operating system; it can be shut down, booted or rebooted. Resources such as disk space, CPU and memory associated to each container when created can be dynamically increased or decreased while the container is running and applications and users see each container as a separate host. Container virtualization allows installation of several different

[4] http://www.markus-gattol.name/ws/lvm.html
[5] http://www.vmware.com/
[6] https://www.virtualbox.org/
[7] https://www.appeagle.com/ecommerce-news/ecommerce-is-on-the-rise-in-2013/

operating systems on top of a single kernel. Although all the operating systems use the same kernel, they have their own filesystem, processes, memory and devices [39].

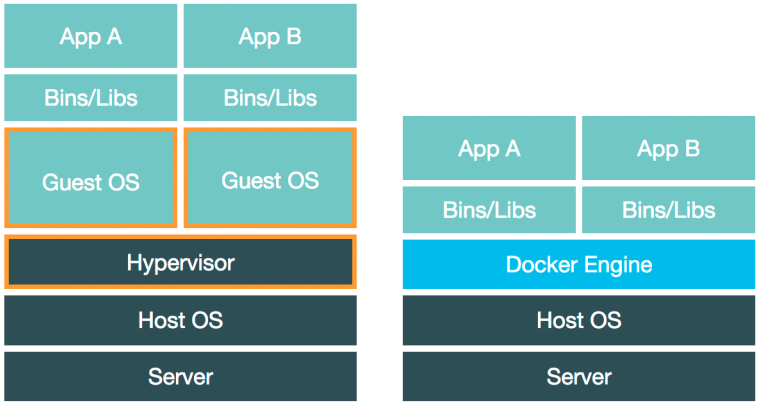

Figure 3.2: Containers vs. traditional virtual machines[7]

Linux Containers

Container virtualization has been developed independently for different operating systems such as Linux OpenVZ[8], Solaris Containers[9], FreeBSD Jails[10]. A popular example is Linux Containers (LXC)[11] that allows a complete copy of the Linux OS to run in a container without the overhead of running a type-2 hypervisor. LXC uses kernel namespaces, AppArmor[12], SELinux[13], chroots, and Control groups to provide container virtualization.

Kernel namespaces is used for virtualization of:

- Process identifiers
- Network interface controllers, firewall rules and routing tables
- Hostname
- Filesystem layouts
- Interprocess communication

Control groups is used to provide:

- Resource limiting to control use of resources such as memory
- Prioritization to control the share of the CPU being used
- Accounting to measure how much resources are being used

Chroots, also know as "chroot jails", are used to change the apparent root directory in the filesystem ensuring applications cannot view files and folders outside of the chroot.

AppArmor and SELinux are used to improve security and ensure that applications cannot break out of the LXC [40].

[7] https://www.docker.io/the_whole_story/
[8] http://openvz.org/Main_Page
[9] http://www.oracle.com/technetwork/server-storage/solaris/containers-169727.html
[10] http://www.freebsd.org/cgi/man.cgi?jail
[11] https://linuxcontainers.org/
[12] https://wiki.ubuntu.com/AppArmor
[13] http://selinuxproject.org/page/Main_Page

LXC provides user environments whose resources can be tightly controlled, without the need to virtualize the hardware resources. It also allows running many copies of application configurations on the same system[14]. This has proven to be a significantly useful feature of these containers for seamless software upgrade (Section 2.6). Furthermore, since the LXC is sharing the kernel with the host system, its processes and filesystem are completely visible from the host; however, this means that the user is limited to the modules and drivers that the container has loaded.

Docker

Docker is an open source application (or framework) that extend and simplifies LXC to provide Linux Containers. Docker allows easy creation of lightweight, portable, self-sufficient containers. Docker extends LXC by providing many features that make it is easier to develop, deploy, automate and share containers[15]. Docker simplifies containerization supporting techniques such as Continuous Delivery by allowing a Docker container built and tested on a developer's laptop to be run anywhere. Docker containers can run on bare metal servers, virtual machines, OpenStack[16] clusters or on a service provider's infrastructure such as Digital Ocean[17].

Advanced Multi-Layered Unification Filesystem (AuFS) is used in Docker as their filesystem. AuFS is a layered filesystem that can transparently overlay one or more existing filesystems (Figure 3.3). A Docker AuFS consists of multiple read-only layers with a single read-write layer at the top merged together to form a single filesystem representation. When a file is modified in the container, the read-only version of the file is copied into the read-write layer using a process called copy-on-write [41]. The copy-on-write approach means that the read-write layer only contains the files that have been modified by the container. Docker supports behaviors similar to git[18] where the read-write filesystem layer can be committed and turned into a new permanent read-only layer called an image. A new container can then be created based on this image or committed filesystem layer. A container created from the image will have a union filesystem that unifies a new copy-on-write read-write filesystem layer with the images' read-only filesystem layer and the dependent image filesystem layers beneath it. A Docker image is therefore simply a diff of changes from the previous base layers, effectively keeping the size of image files to minimum. This also means that image creators have a complete audit trail of changes from one version of a container to another.

In addition, Docker provides a scripting language for creating containers and images based on other images. The script, called a Dockerfile, defines the differences between the new image and the previous base image. Dockerfiles allow the execution of shell commands, the configuration of processes to run when the container is started and control over the public interface of the container including exposed ports and directories.

Docker also provides a registry of containers called the Docker Index. It allows containers to be publicly shared. The index contains images created by committing a filesystem layer and images created by the Docker Index build system using a Dockerfile [41].

With Docker, a new application on a host only needs its binaries or libraries but not a new guest OS. In addition, the same application binaries can be shared between multiple running copies of the application using a shared Docker image. If modifications are made between different versions of the application only the differences need to be maintained separately[19].

[14] https://linuxcontainers.org/
[15] https://www.docker.io/the_whole_story/
[16] https://www.openstack.org/
[17] https://www.digitalocean.com/
[18] http://git-scm.com/
[18] https://docs.docker.com/terms/layer/
[19] https://www.docker.io/the_whole_story/

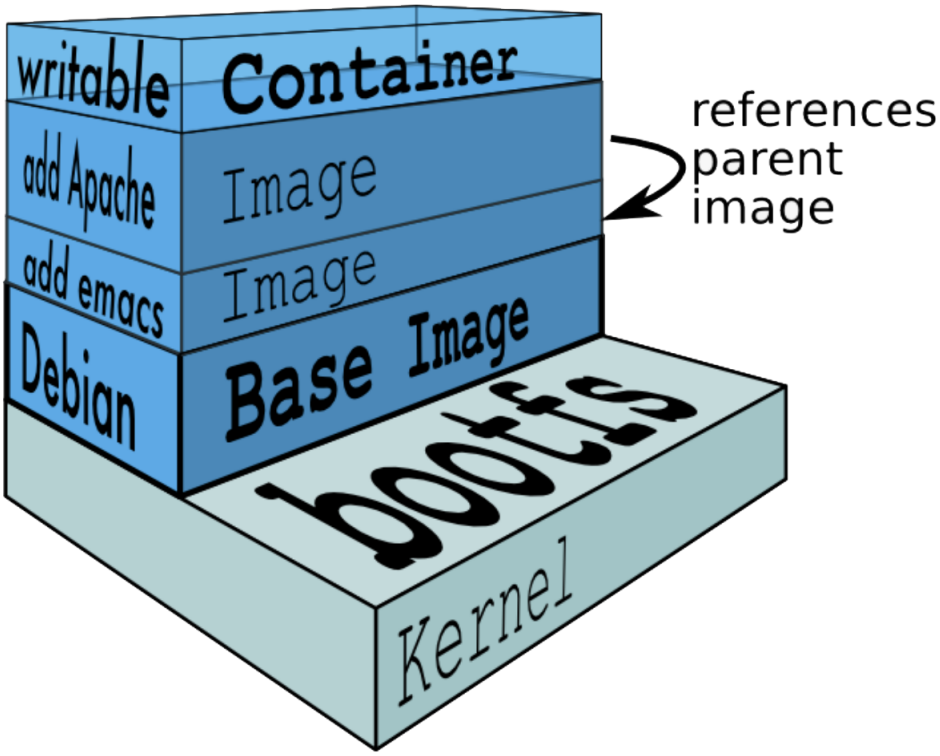

Figure 3.3: Docker filesystem[18]

Chapter 4

Design

4.1 Requirements

The main objectives of this project were to implement a proxy that adds no significant overhead while:

- supporting reliable dynamic software update by seamlessly routing requests to the latest application version
- supporting seamless rollback to the previous version of the application when the upgraded version behaves incorrectly
- managing application clusters using container virtualization
- being reliable, robust and stable under heavy load

Typically different application update situations require different update approaches:

- Serious bugs such as a critical security flaw requires the application to be updated immediately to reduce the error or security exposure as quickly as possible. All users should therefore immediately be updated to the most recent version.
- A new feature may change the behaviour of the system. Users who are currently interacting with the system should typically not experience the new feature until their current interaction session has completed. For example a user in the process of purchasing products on an e-commerce site would typically not want the sites behaviour to change during a shopping experience. In this case the user should not experience the new application version until their next interaction with the site.
- To reduce the risk associated with deploying a new feature application updates are often rolled out gradually. A gradual update mechanism slowly migrates users from the existing application version to the updated application version in a controlled manner.
- In the case where an application upgrade does not significantly change the user facing feature set or the shape of the API the old and the new versions can be run in parallel. All requests are sent to both versions of the application as long as the new application version responds correctly then this response will be returned to the user. If the new application version fails to respond, crashes or returns a server error then the response from the previous version can be returned to the user.

4.2 Dynamic Software Update Proxy

The following section describes the high level design an component architecture used to meet the requirements.

4.2.1 Technology Stack

The following technology stack has been chosen for the dynamic software update system:

- Go[1] programming language for building the proxy. Go is chosen because of its simplicity, efficiency, scalability, highly concurrency and garbage collection. Other fast programming languages such as C++ and Java were considered. However, C++ was not chosen since it does not have garbage collection and requires manual memory management. Java was not chosen since it is more heavyweight in comparison to Go, does not give the programmer control over the memory management and is therefore not likely to be as efficient as Go.
- Docker as a container virtualization mechanism that runs containers for both the target application that is to be updated and the content switching load balancer.
- ZeroMQ as message frameworks that support highly efficient load balancing and routing of messages without the need for any middleware.
- HTTP as a protocol for application requests and responses. HTTP is the most widely adopted application level protocol for distributed network systems In addition the use of headers and cookies in HTTP allow sessions and clients to be easily identify. HTTP headers also allows the efficient comparison between large responses without the need to buffer the entire response body.

4.2.2 Upgrade Modes

To support the multiple update approaches listed in the requirements (Section 3.1) four mechanisms are supported.

Instant Upgrade

This mechanism is useful when an update must be applied urgently, such as a critical security patch required to stop an in-progress security breach. In this mechanism the updated version of software immediately processes all requests. The non-updated version remains running and will be immediately available to process requests if the updated system does not behave correctly. Figure 4.1 shows a cookie, indicating the application version, is always added to a response. When an instant upgrade mode is used the proxy routes all requests to the new version and updates their cookie. When a failure occurs the proxy routes the request to the previous version and removes the failed version from the list of available versions.

Session Upgrade

This upgrade mode is useful when a new feature is release which affects the behaviour of the system and therefore only new sessions are switched to the updated version. Figure 4.2 shows that when a request is received the cookie, indicating which version the client was previously using, is examined to determine which version the request should be routed to. The cookie in the response is updated to add a timeout. A request with no cookie is only received from clients who have not made a request within the timeout period, therefore, a new session is identified by a request with no cookie.

Gradual Upgrade

This mechanism switches new sessions gradually to the updated version over multiple days or weeks and is useful to reduce the risk associated with a new application version by gradually increasing

[1] `http://golang.org/`

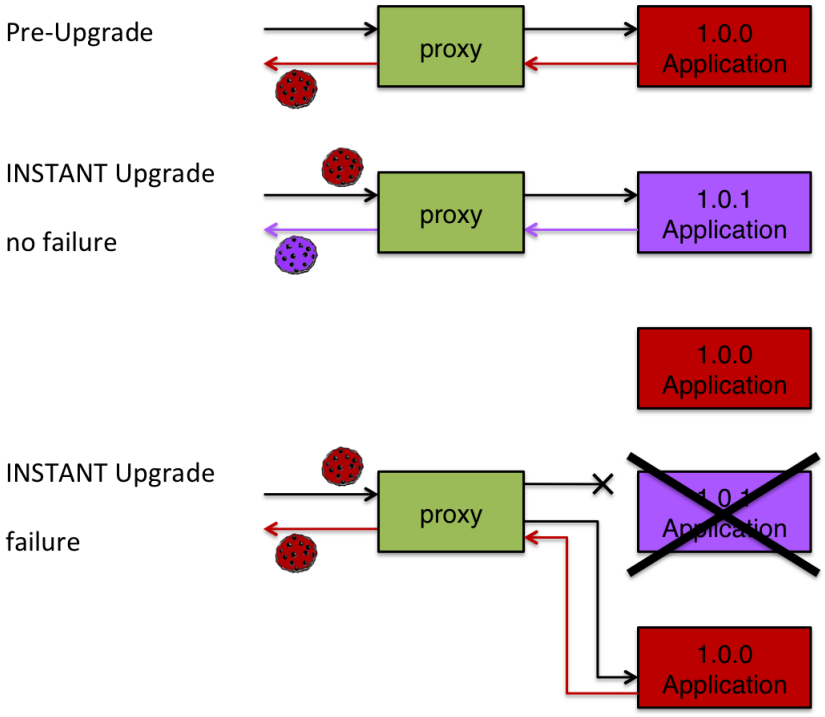

Figure 4.1: Instant upgrade and failure rollback

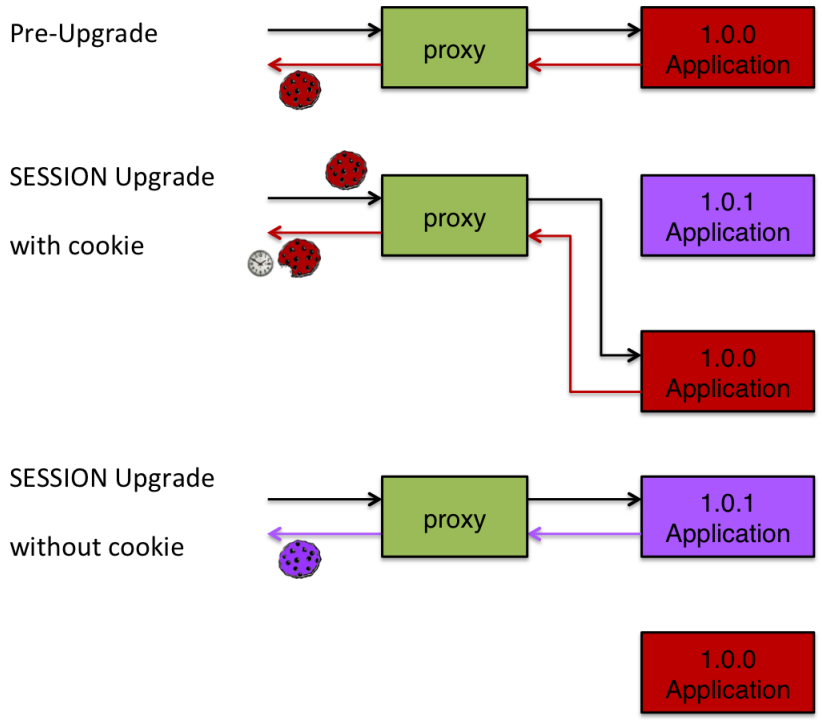

Figure 4.2: Session upgrade

the number clients that are exposed to the new version. Figure 4.2 shows how this approach uses a second cookie in addition to the cookie used in the session and instant upgrade modes. The second cookie has no expiry and is never updated once it is assigned to a client. This second cookie contains a unique randomly assigned id, that is given to a client, between 0 and 100. Each request will be routed to the version depending on which bucket it falls into. The Figure 4.2 shows in the first upgrade scenario the second cookie gives the request a number of 45. The bucket for application

1.0.1 is from 0 to 10 therefore the the request does not match the bucket as 45 falls outside the range 0 to 10. In the second upgrade scenario the request number of 45 falls within the bucket, for application 1.0.1, because the range has been increased and is now 0 to 50. This bucket range is gradually increased for each successfully processed request, at a configurable rate, so that the number of clients that use the new application version is gradually increased.

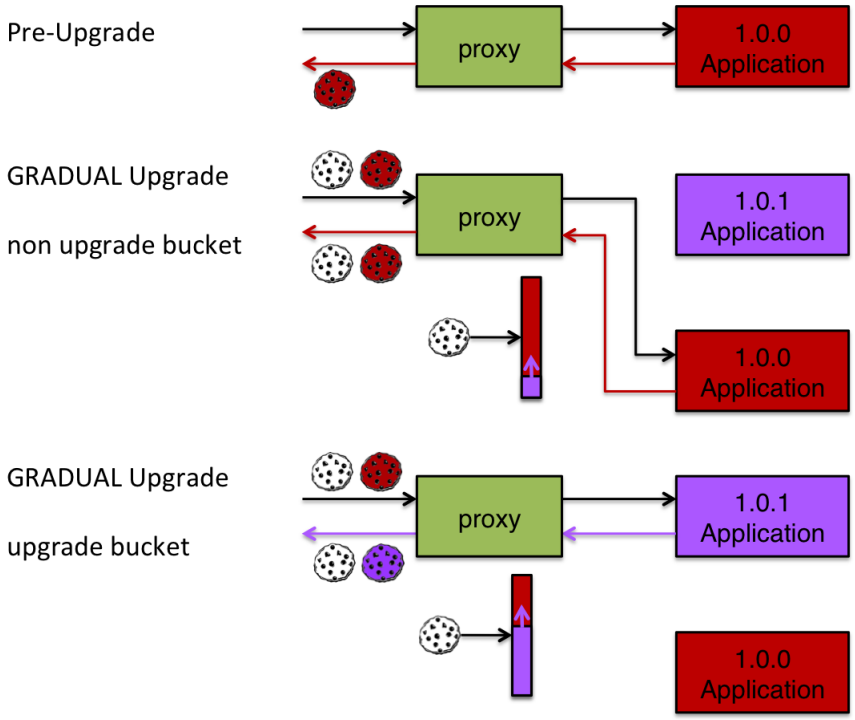

Figure 4.3: Gradual upgrade

Concurrent Upgrade

In this mechanism the updated and non-updated versions run concurrently and process identical requests. As shown in Figure 4.4 in this strategy, both the old and the new versions handle all requests. If the latest version of the application behaves correctly then the client receives the response from the latest version. The latest version of the application is considered to behave correctly if it accepts a TCP connection, returns a response, and the response has an HTTP status code that is not in the 5xx range, used to indicate server errors.

In all four mechanisms the behaviour of the updated software is automatically monitored to confirm the application is behaving correctly. The ability to receive incoming requests and a response with a non 5xx range status code is used to verify that the updated application is behaving correctly.

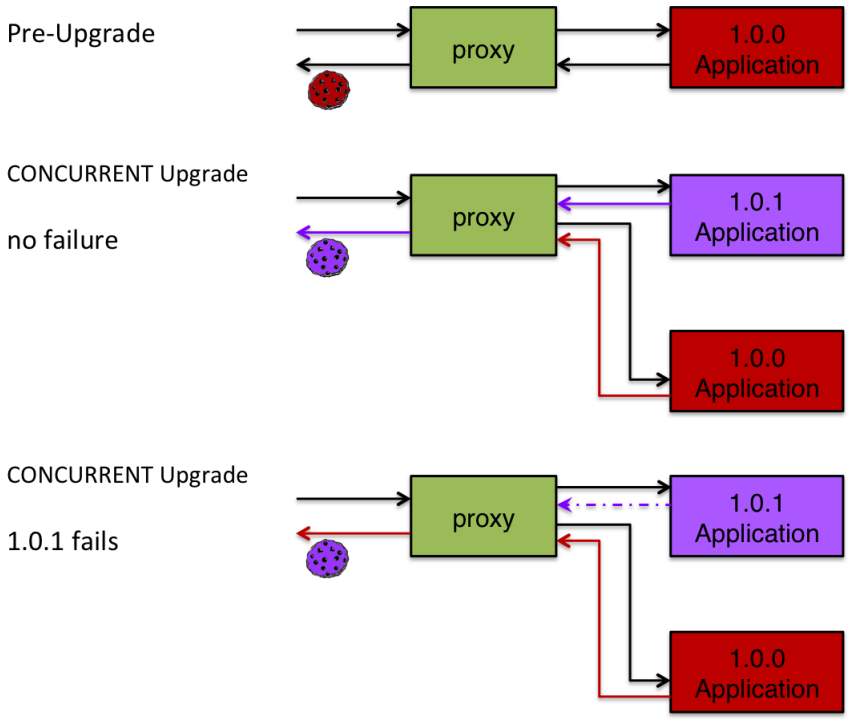

Figure 4.4: Concurrent upgrade

4.2.3 Component Architecture

Figure 4.5 demonstrates the main components and there interactions for the proposed dynamic software update system, which are described below.

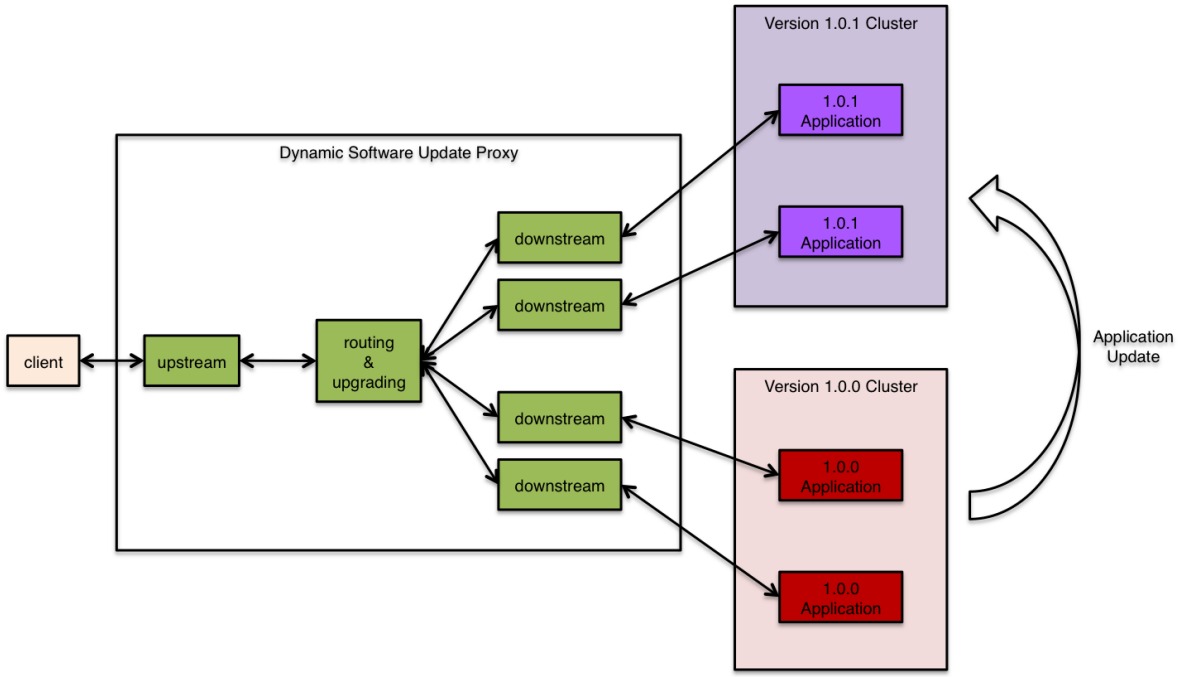

Figure 4.5: Component interactions within the proposed dynamic software update system

client

The client indicate the user of the application that is sending requests for the application to process. It is assumed that the client will correctly interact using the HTTP protocol including correctly handling cookies.

upstream

The upstream component encapsulates all the interactions with the client. This includes accepting incoming TCP connections, reading the request from the socket and writing the response to the socket.

routing & upgrading

This component is responsible to handling all the upgrade modes and determining which server to route the request is. This component parses the headers from both the request and response to manage the cookies required for the upgrade modes. This component is also used to detect when a server is behaving incorrectly by examining the server response. Load balancing is also performed by this component as it routes requests to the servers within a cluster in a round-robin fashion.

downstream

This component is responsible to making the TCP connection to the server. Once the connection is established this component will write the request to the socket and will read the response of the

socket.

Version 1.0.0 Cluster

Shows two application nodes, both using version 1.0.0 of the application, running in a single cluster.

Version 1.0.1 Cluster

Shows an updated version of the application as two application nodes, using version 1.0.1, of the application running in a single cluster.

Chapter 5

Implementation

During the implementation of the proxy three separate approaches were used. The first approach as described in Section 5.1 was to use ZeroMQ due to its support for low latency messaging and sophisticated support for multiple routing strategies. However this approach was not successful due to ZeroMQ's handling of multiple HTTP chunks and the overhead added. The second approach describe in Section 5.2.1 was to used raw sockets. To detect when the sockets should be closed the request and response body content was counted as each chunk was transmitted; once all the content was received the socket was closed. However, due to the complexity, inconsistency and inaccuracy in the way different servers implemented the HTTP protocol this approach was not pursued. The final approach described in Section 5.2.2 was to use raw sockets in a staged based proxy. This version improved encapsulation and separation-of-concerns through the use of multiple stages. In addition this approach used low level socket errors instead of content counting to determine when to close sockets. This final approach met all the initial requirements and hence it was developed further to provide the full set of features.

5.1 Message Based Proxy Using ZeroMQ

The first proxy developed to support dynamic software update was an HTTP reverse proxy based on ZeroMQ messaging system. ZeroMQ was chosen due to its support for low latency messaging and sophisticated support for multiple routing strategies. However this approach was not successful due to ZeroMQ's handling of multiple HTTP chunks and the overhead added as described in this section. A custom Dockerfile was written that creates a Docker container running an example web service using Netty[1]. The proxy uses different types of ZeroMQ sockets such as STREAM, REQ, REP and it is arranged as shown in Figure 5.1. Each box is a separate thread, a separate process or a Docker container and performs the following responsibilities:

- **upstream** uses STREAM socket for receiving HTTP requests and replying with the HTTP response. Upstream also consists of REQ socket that forwards the requests to the router.
- **router** is responsible for distributing requests across four downstream threads using ROUTER and DEALER sockets. ROUTER socket adds the identity of the sender and receiver to responses and requests respectively. DEALER socket distributes the requests across downstream threads in a round-robin order.
- **downstream** is responsible for receiving the requests from the router using REP sockets and sending HTTP requests to the two Docker containers containing the Netty web services and receiving the responses using STREAM socket.

[1] `http://netty.io/`

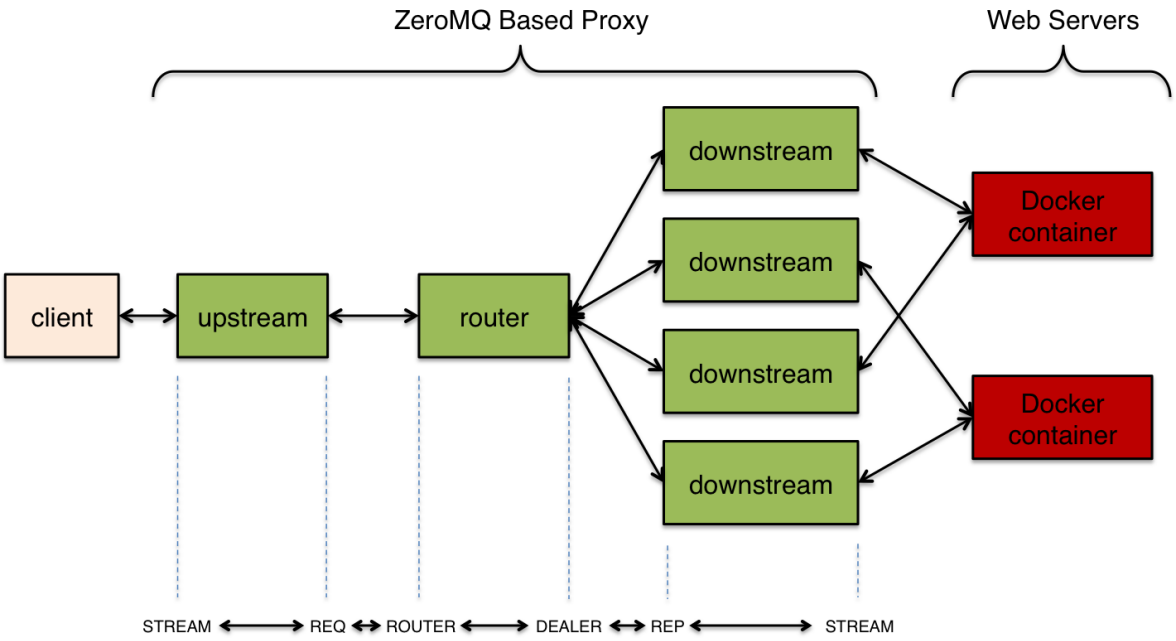

Figure 5.1: Message based proxy using ZeroMQ

5.1.1 Performance Evaluation

Detailed analysis of the reverse HTTP proxy showed that when the HTTP requests are chunked (Section 3.1.1), ZeroMQ STREAM socket treats each chunk as a separate message and the DEALER socket distributes each chunk of the same request to different recipients. Therefore, it was concluded that ZeroMQ messaging library is not suitable and reliable for load balancing chunked HTTP requests and a simpler approach would be to use raw Go sockets directly. Furthermore, HTTP provides adequate messaging support such as content encoding and caching and wrapping HTTP messages in another message layer adds unnecessary complexity and overhead. Therefore, other messaging systems were not explored for developing a content switching load balancer.

In conclusion, the ZeroMQ HTTP reverse proxy was not advanced further to support dynamic software update instead an alternative approach using raw sockets was developed.

5.2 Socket Based Proxies

As it was explained in the previous section using a messaging system was not the correct choice to develop a dynamic software update proxy. This section describes two different approaches using raw Go sockets.

5.2.1 Socket Based Proxy With Content Counting

The first version of the socket based proxy was designed as demonstrated in Figure 5.2. To detect when the sockets should be closed the request and response body content was counted as each chunk was transmitted; once all the content was received the socket was closed. However, due to the complexity, inconsistency and inaccuracy in the way different servers implemented the HTTP protocol this approach was not pursued, as described in this section.

To accept new connections this proxy had a listener running in an *Accept* loop. When a new connection was received a TCP connection between the client and the proxy was created over which the HTTP request read from the socket in *Read* loop. The routing component created another

connection between the proxy and the server. The HTTP request was then written to the server inside a *Write* loop.

An identical but opposite set of *Read* and *Write* loops are created to return the response from the server back to the client.

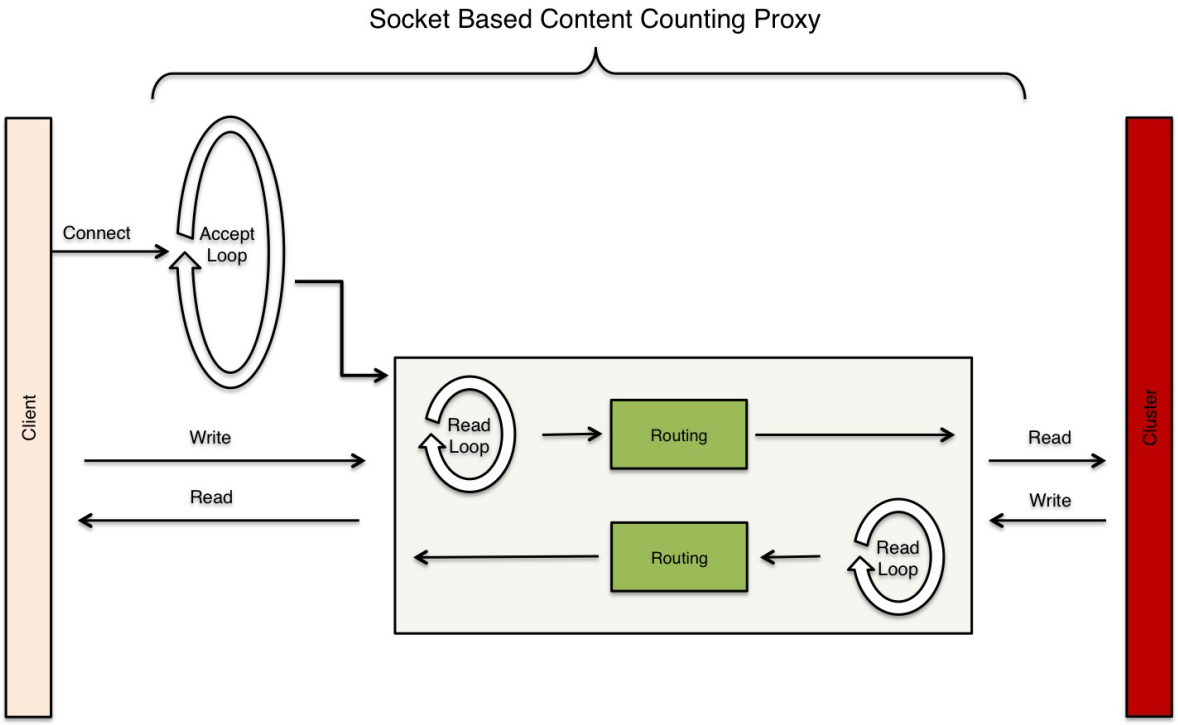

Figure 5.2: Socket based proxy with content counting

Content Counting

The content counting proxy uses the HTTP format to detect the end of each request and response (Section 3.1). Detecting the end of requests/responses is used to close connection sockets when the complete message has been transmitted. For HTTP requests, the method token in the Request-Line is used to detect what type of HTTP function is expected (Section 3.1.1). If the method is defined as GET, no message body in the request is expected and the *Read* loop ends after the first read containing all the headers. However, for other HTTP functions such as POST or PUT and for HTTP responses, the Transfer-Encoding and Content-Length headers were used (Section 3.1).

Content-Length

When Content-Length header is provided, the size of message body is known upfront. The proxy then counts the number of bytes in each chunk. The first chunk can contain both headers and part of the the message body so the CRLF CRLF is used to detect where the message body starts (Section 3.1). For each subsequent chunk the content length is counted. When the length of read data is the same as the size defined in Content-Length header, the *Read* loop terminates.

Transfer-Encoding: "chunked"

However, when no Content-Length header is provided, the size of the message body is not known upfront. Instead the Transfer-Encoding header with a value of "chunked" indicates the message body is chunked and a zero chunk will mark the end of the message body. At the start of each new chunk, the chunk size is defined as a hexadecimal number followed by '\r\n', therefore, the format

for the last chunk is always defined as '0 \r\n \r\n'. To detect the end of HTTP messages with chunked Transfer-Encoding header, the content counting proxy screens each chunk and when the last chunk, *i.e.* '0 \r\n \r\n', is detected, the *Read* loop ends.

Performance Evaluation

To evaluate the performance of the aforementioned proxy an HTTP benchmarking tool called *wrk*[2] was used. *wrk* is capable of generating significant load when it runs on a single multi-core CPU by combining a multi-threaded design and scalable event notification systems such as epoll[3] and kqueue[4].

HTTP Requests were fired to an example Go web service either directly or via the implemented content counting proxy. The processed time for each request was calculated using *wrk*. The following terminal outputs are an example of results obtained by *wrk* when the benchmark was run for 2 minutes, using 400 threads and keeping 400 HTTP connections open.

Direct HTTP Requests To Go Server

```
$ ./wrk -t400 -c400 -d120 --latency http://127.0.0.1:1024
Running 2m test @ http://127.0.0.1:1024
  400 threads and 400 connections
  Thread Stats   Avg      Stdev     Max   +/- Stdev
    Latency    56.33ms    5.34ms 113.13ms   97.48%
    Req/Sec    17.26      2.09    28.00     94.61%
  Latency Distribution
     50%   55.85ms
     75%   56.36ms
     90%   56.89ms
     99%   97.85ms
  854993 requests in 2.00m, 126.38MB read
Requests/sec:   7123.93
Transfer/sec:      1.05MB
```

HTTP Requests To Go Server Via Content Counting Proxy

```
$ ./wrk -t400 -c400 -d120 --latency http://127.0.0.1:1234
Running 2m test @ http://127.0.0.1:1234
  400 threads and 400 connections
  Thread Stats   Avg      Stdev     Max   +/- Stdev
    Latency   381.51ms  131.93ms 482.79ms   83.01%
    Req/Sec     0.48      2.34    26.00     95.10%
  Latency Distribution
     50%  472.32ms
     75%  480.15ms
     90%  481.42ms
     99%  482.69ms
  25835 requests in 2.00m, 3.82MB read
  Socket errors: connect 0, read 27144, write 681, timeout 21088
Requests/sec:    215.24
Transfer/sec:     32.58KB
```

The above results show that significant socket errors such as read and write errors occur when the requests are sent to the Go server via proxy. The socket errors consequently reduce the number of

[2] https://github.com/wg/wrk

[3] http://man7.org/linux/man-pages/man7/epoll.7.html

[4] http://www.freebsd.org/cgi/man.cgi?query=kqueue&sektion=2

requests processed by the server (*e.g.* from 7123.93/second to 215.24/second), the amount of data transferred (*e.g.* from 1.05MB/second to 32.58KB/second) and finally increases the time taken for each request to be processed (average time increased from 56.33ms to 381.51ms). Detailed analysis of the proxy showed that the content counting proxy is not consistently detecting the end of messages, therefore, the read and write sockets do not close appropriately which results in socket errors. This was caused by the complexity in counting all the bytes in each chunk and detecting the terminating chunk given that not all servers used a consistent approach. When the proxy was tested against public servers on the internet several servers sent incorrectly formatted terminating chunks with one or more /r or /n characters missing, other servers had a large delay before responding with the terminating chunk and some servers sent multiple terminating chunks.

In conclusion, the significant overhead and socket errors associated with the implemented proxy lead to the conclusion that a proxy with content counting is not suitable for supporting reliable and efficient dynamic software updates. Therefore, another version of the socket based proxy was developed as described in the next section.

5.2.2 Socket Based Staged Proxy With End Of File Signal

The final version of the proxy was implemented using raw sockets in a stage based design as demonstrated in Figure 5.3. This final approach met all the initial requirements and hence it was developed further to provide the full set of features. This approach was used to improve the code design, simplicity and to support high test coverage by promoting encapsulation, separation-of-concerns and allowing inversion-of-control and dependency-injection. This resulted in the following benefits:

- **Encapsulation And Separation Of Concerns** - resulted in separating logic for each functional area into a single component. This creates a simple and isolated code that focuses on one topic, making it clear and easier to understand. Additionally, encapsulation and separation of concerns simplified testing by allowing each test to be focused on one specific area. For example, multiple tests could be writing focusing on reading from a socket and dealing with different errors and situations that can occur. This approach increase overall reliability in comparison to the 5.2.1 proxy.
- **Inversion-Of-Control And Dependency Injection** - allowed incremental development of the proxy where new components were plugged-in and configured one by one while existing components all worked together. For the HTTP dynamic update proxy shown in Figure 5.3 the *Read* and *Write* stages were first developed. That was followed by development of the *Complete* and *Route* stages respectively and then the rest of proxy was implemented. Furthermore, This approach also simplified testing of the proxy by allowing the code-under-test to be isolated by mocking all dependencies and permitting the mocking of different error situations that would be impossible to test without mocking dependencies

To prevent the socket errors that occurred in the content counting proxy 5.2.1, the staged HTTP dynamic update proxy uses the End-Of-File (EOF) signal to detect the end of messages instead of content counting. Different component and stages of the proxy is shown in Figure 5.3 and their functionality is explained below.

Accept Loop

Accept loop accepts incoming HTTP connections on a listener and creates a *Forward pipe* for each request.

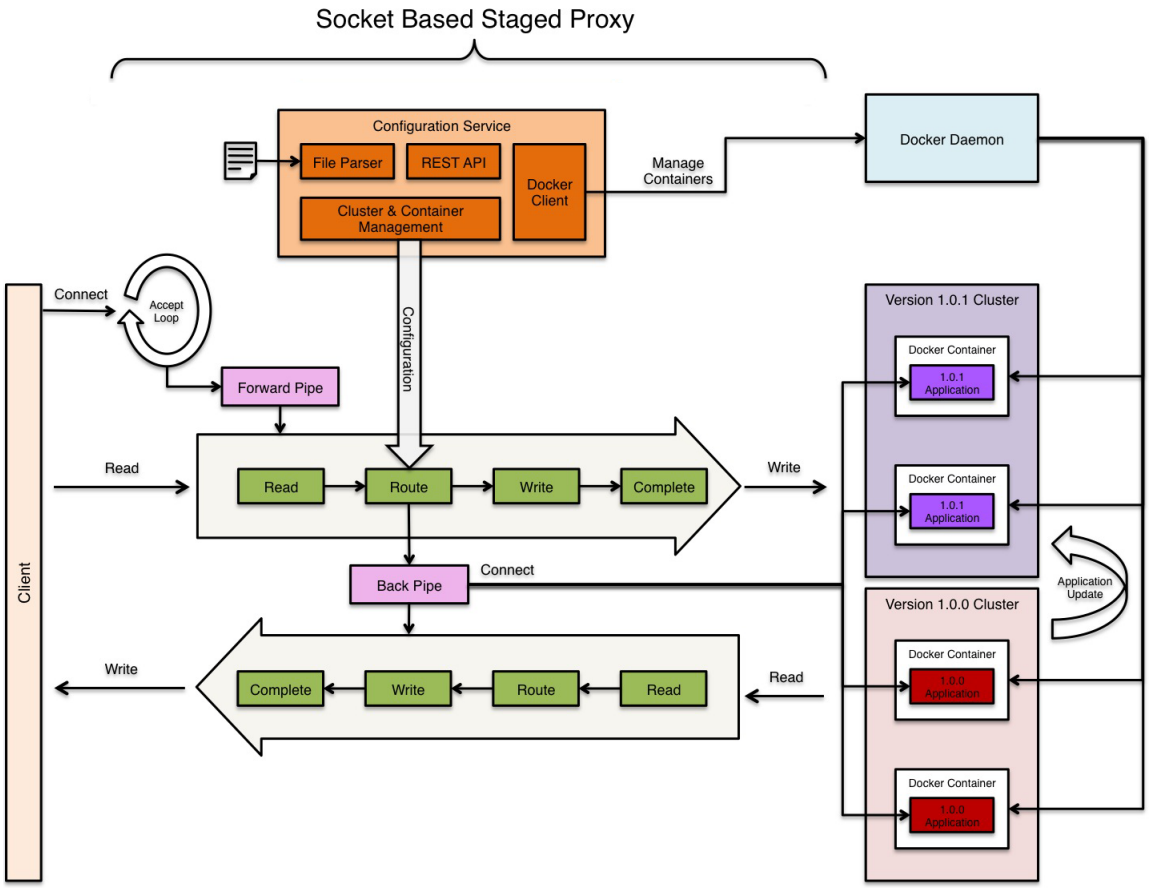

Figure 5.3: Socket based staged proxy design supporting dynamic software update

Forward Pipe

Forward pipe constructs the *ChunkContext Struct* for the received request and creates *Read*, *Route*, *Write* and *Complete* stages of the proxy for forwarding the request to the appropriate server in the cluster. The *ChunkContext Struct* is used to encapsulate the information for managing the transferring chunks of data between the client and servers of the proxy. The *ChunkContext* contains information such as network addresses of the client and the server and the data transfer direction, as follows:

```
type ChunkContext struct {
        Data                    []byte
        To                      tcp.TCPConnection
        From                    tcp.TCPConnection
        Err                     error
        TotalReadSize           int64
        TotalWriteSize          int64
        PipeComplete            chan int64
        FirstChunk              bool
        RoutingContext          *RoutingContext
        Direction               Direction
}
```

```
type RoutingContext struct {
        Headers []string
}
```

Read Stage

Read stage is responsible for reading input in a loop from the HTTP connection. The *Read* stage in *Forward pipe* reads requests from the client, while the one in *Back pipe* reads responses from the server. The *Read* loop terminates when an error is encountered while reading from a socket. In the staged proxy (Figure 5.3) when an EOF condition (*i.e.* when no more input is available) occurs, the *Read* loop treats this condition as a low level socket signal and it stops reading from the connection. This approach allows a graceful and efficient way of detecting the end of input. It also prevents the need for parsing requests / responses and the content counting process mentioned in section 5.2.1, which is very inefficient and unreliable. In addition, the socket based proxy handles socket interaction efficiently by only closing sockets when the other end of the connection closes its socket. This avoids overhead associated with creating new TCP connections such as the TCP 3-way handshake and TCP slow start.

Route Stage

Route stage has different functionality in the *Forward Pipe*, for processing requests and *Back pipe*, for processing responses.

Routing in *Forward Pipe* - When a new request arrives in the *Route* stage, the upgrade transition mode of the cluster with the highest version determines how the request is forwarded. Currently the proxy supports four different upgrade transition modes; INSTANT, SESSION, GRADUAL and CONCURRENT (Section 4.2.2). The *Route* stage distributes the requests using the modes according to the rules summarized below:

- **INSTANT** - for INSTANT mode requests are always forwarded to the cluster with the highest version. INSTANT is the default mode so if no upgrade transition is defined for a cluster then it is given INSTANT mode (Section 5.5.1). In INSTANT mode if the latest cluster fails, it will be removed from the set of available clusters resulting in client requests being routed to the previous version.

- **SESSION** - in SESSION mode the proxy generates a session cookie with a configured expiry time for each request forwarded to that cluster. The session cookie is added as a Set-Cookie header to all HTTP responses. An HTTP client will return the cookie as long as it has not expired. This ensures that a client will remain on the same cluster version until it has not interacted with the proxy for >= the cookie expiry period. If a request has a cookie (*i.e.* it has not timed out) the *Route* stage forwards the request to the cluster associated with the uuid in the cookie. If no cookie is found, requests will be forwarded according to the upgrade transition mode of the cluster with highest version number. In SESSION mode if the latest cluster fails, it will be removed from the set of available clusters resulting in client requests being routed to the previous version.

- **GRADUAL** - an additional transition cookie is used to control GRADUAL mode routing. The transition cookies is added to each response but as it does not have an expiry time, each client should keep the same cookie. The transition cookie is used to randomly allocate each client a number between 0 and 100. This number is then used to determine which clients should be routed to the latest cluster version. If the randomly allocated value is <= to a defined threshold then it will be routed to the latest cluster version. As more requests are correctly processed by the latest cluster version, the threshold continually increases. This results in a gradual increase in the percentage of clients being forwarded to the latest cluster version. As an example, if the threshold is 15 then 15% of random clients will use the latest cluster version. Once a client is assigned a random value, the client will keep that value. Therefore when clients start using the latest cluster, they will remain on that cluster. However, if the

latest cluster fails, it will be removed from the set of available clusters resulting in requests being routed to the previous version.

- **CONCURRENT** - for CONCURRENT mode, the *Route* stage forwards requests concurrently to a configurable number of the most updated cluster versions. If the latest cluster version correctly accepts a new TCP connection and responds with an HTTP status code that is not in the 5xx range, for server errors, then the client will receive a response from the latest cluster. If the latest cluster does not accept a new TCP connection or responds with an HTTP status code in the 5xx range then the client will receive a response from the previous cluster version.

If the latest cluster version is deleted using the REST API or an error occurs while routing the requests to the latest cluster version, the request will be automatically routed to the previous cluster version. If routing to all versions fails, the proxy returns an an empty response and outputs an appropriate error to the terminal indicating that no clusters are available. If the request is successfully routed, a *Back pipe* will be created by *Route* stage for processing responses.

Routing in *Back pipe* - The *Route* stage processes responses by adding the appropriate cookie headers. In INSTANT mode a cluster uuid cookie is added with no expiry time. In SESSION mode a cluster uuid cookie is added with the configured expiry time. In GRADUAL mode two cookies with no expiry time are added for the cluster uuid and the randomly assigned transition value. Finally, if the mode is defined as CONCURRENT, the response status code is checked. If the response from the latest version is in the 5xx range, that response will be dropped and the response from the previous version will be returned. Also in CONCURRENT mode a cluster uuid cookie is added with no expiry time.

Write Stage

Write stage writes requests / responses to output stream for the appropriate socket. The writing process stops if any errors such as *ErrShortWrite* are encountered while writing to the socket. The *Write* stage in *Forward pipe* writes data to the server, while the *Write* stage in *Back pipe* writes data to the client.

Complete Stage

Complete stage is responsible for shutting down the writing and reading sides of the TCP connection when the *Read* and *Write* stages terminate. This is done to insure that a TCP connection is reliably closed when the client/server communication is ended.

Back Pipe

Back pipe is responsible for receiving requests from the server and forwarding it to the client. Similar to *Forward pipe*, *Back pipe* constructs the *ChunkContext Struct* for responses and creates *Read*, *Route*, *Write* and *Complete* stages for forwarding the response to the client.

Configuration Service

Configuration Service is responsible for managing the configuration for how the proxy routes to different servers or containers. In addition the configuration service configures the ports exposed by the proxy for both the configuration service its self and for incoming HTTP requests that are

being proxied. The configuration service is made up of multiple separate components, including File Parser, REST API, Cluster & Container Management and the Docker Client.

File Parser

File Parser is responsible for parsing loading the configuration file provided to the proxy when it initially starts. The file parser reads the configuration in JSON format and validates it strickly ensuring to output any necessary errors as required.

REST API

REST API exposes a REST service API that can be used to query and update the configuration once the proxy has started. The REST API can be used to add new server or container clusters. When a new container cluster is added the configuration service uses the Docker Client to ensure that the necessary containers are created by the Docker Daemon. The REST API also supports deleting server or container configuration, if the configuration is deleted then the proxy will not longer send requests to the deleted servers or containers. In addition if the container configuration is deleted then the proxy will stop and remove then relevant containers by communicating with the Docker Daemon.

Docker Client

Docker Client manages all the interactions with the Docker Daemon including pulling images, creating containers, starting containers, querying container status, stopping containers, and removing containers. The Docker Client is also responsible for stream the response from the Docker Daemon to console output when the proxy starts or to the response of a REST API request. This ensures that any failure while performing any Docker action is clearly indicated to the user. To ensure that it is quick and simple for the user to resolve any issues with docker containers the Docker Client also attaches to each new container for the first 3 second of its initial start up and returns the log (standard out and standard error) to the user. This is particularly helpful when triaging why a container has not started.

The Docker Client enables support for the following Docker configurations:

- name of the image to create each container from
- the tag for the image, defaulting to ‘latest‘
- whether to automatically check for new image versions
- the name of the container, defaulting to an auto-generated unique name
- the current working directory inside the container’s root file system
- the container’s entrypoint
- the environment variables that are set in the running container
- the command to be executed when running the container
- the container’s hostname
- mounting volumes from either the host or from another docker container
- mounting all the volumes defined for one or more other docker containers, including control over whether the volumes are mounted in read-write or read-only mode
- binding ports from within the container to one or more host port and IP combinations
- linking the container to another Docker container
- setting the user user id and group id of the executing process running inside the container

- restricting the container's memory (in bytes)
- restricting the container's cpu share using relative weight compared to other containers
- adding custom LXC options for the container
- giving extended privileges to the container

Cluster & Container Management

Cluster & Container Management is the bridge between the File Parser, REST API, Docker Client and the routing stage of the proxy. This component manages the list of all server and container configurations and which configuration are grouped into a cluster. All persistent data is stored in this component to support routing decisions, such as, for example information to support the gradual upgrade mode, as follows:

```
type Clusters  struct {
        ContextsByVersion    *list.List
        ContextsByID         map[string]*Cluster
        DockerHostEndpoint   string
}
```

```
type Cluster struct {
        BackendAddresses                []*BackendAddress
        DockerConfigurations            []*docker_client.DockerConfig
        RequestCounter                  int64
        TransitionCounter               float64
        PercentageTransitionPerRequest  float64
        Uuid                            uuid.UUID
        SessionTimeout                  int64
        Mode                            TransitionMode
        Version                         string
}
```

```
type DockerConfig struct {
        Image           string `json:"image,omitempty"`
        Tag             string `json:"tag,omitempty"`
        AlwaysPull      bool `json:"alwaysPull,omitempty"`
        PortToProxy     int64 `json:"portToProxy,omitempty"`
        Name            string `json:"name,omitempty"`
        WorkingDir      string `json:"workingDir,omitempty"`
        Entrypoint      []string `json:"entrypoint,omitempty"`
        Environment     []string `json:"environment,omitempty"`
        Cmd             []string `json:"cmd,omitempty"`
        Hostname        string `json:"hostname,omitempty"`
        Volumes         []string `json:"volumes,omitempty"`
        VolumesFrom     []string `json:"volumesFrom,omitempty"`
        PortBindings    map[Port][]PortBinding `json:"portBindings,omitempty"`
        Links           []string `json:"links,omitempty"`
        User            string `json:"user,omitempty"`
        Memory          int64 `json:"memory,omitempty"`
        CpuShares       int64 `json:"cpuShares,omitempty"`
        LxcConf         []KeyValuePair `json:"lxcConf,omitempty"`
        Privileged      bool `json:"privileged,omitempty"`
}
```

5.3 Proxy Installation

The proxy can be installed in two simple steps as follows:

- git clone `https://github.com/samirabloom/dynamic-software-update`
- make

The above steps install the proxy to the PATH by adding it to the */usr/local/bin* directory. However, this will only work if the PATH environment variable has */usr/local/bin* in its list of directories.

5.4 Command Line Interface

The proxy can be run from the command line with the following options:

```
Usage of proxy:
  -configFile="./config.json":  Set the location of the configuration file
  that should contain configuration to start the proxy,
  for example:
              {
                  "proxy": {
                      "port": 1235
                  },
                  "configService": {
                      "port": 9090
                  },
                  "dockerHost": {
                      "ip": "127.0.0.1",
                      "port": 2375
                  },
                  "cluster": {
                      "containers":[
                          {
                              "image": "mysql",
                              "tag": "latest",
                              "name": "some-mysql",
                              "environment": [
                                  "MYSQL_ROOT_PASSWORD=mysecretpassword"
                              ],
                              "volumes": [
                                  "/var/lib/mysql:/var/lib/mysql"
                              ]
                          },
                          {
                              "image": "wordpress",
                              "tag": "3.9.1",
                              "portToProxy": 8080,
                              "name": "some-wordpress",
                              "links": [
                                  "some-mysql:mysql"
                              ],
                              "portBindings": {
                                  "80/tcp": [
                                      {
                                          "HostIp": "0.0.0.0",
                                          "HostPort": "8080"
                                      }
                                  ]
```

```
                            }
                        }
                    ],
                    "version": "3.9.1"
                }
            }

  -logLevel="WARN":  Set the log level as "CRITICAL", "ERROR", "WARNING",
  "NOTICE", "INFO" or "DEBUG"

  -h: Displays this message
```

For example the following command will run the proxy in the "INFO" log level and uses a file called *config/config_script.json* for its configuration.

```
proxy -logLevel=INFO -configFile= "config/config_script.json"
```

5.5 REST API

The proxy provides a simple REST API (Figure 5.3) to support dynamically updating the cluster configuration as follows:

- **PUT /configuration/cluster** - adds a new cluster configuration
- **GET /configuration/cluster/clusterId** - gets a single cluster configuration
- **GET /configuration/cluster** - gets a list of all cluster configurations
- **DELETE /configuration/cluster/clusterId** - deletes a single cluster configuration

The REST API supports the following HTTP response codes:

- 202 Accepted - a new cluster entity is successfully added or deleted
- 200 OK - cluster(s) entity is successfully returned
- 404 Not Found - cluster id is invalid
- 400 Bad Request - request syntax is invalid

5.5.1 PUT /configuration/cluster

The PUT request to */configuration/cluster* is used to add a new cluster to the proxy. There are two supported request body formats depending on whether the cluster is using docker containers or pre-existing servers (i.e. IP and port combinations), as follows:

Docker Container Configuration

```
{
    "cluster": {
        "containers": [
            {
                "image": "",
                "tag": "",
                "alwaysPull": false,
```

```
                "portToProxy": 0,
                "name": "",
                "workingDir": "",
                "entrypoint": [
                    ""
                ],
                "environment": [
                    ""
                ],
                "cmd": [
                    ""
                ],
                "hostname": "",
                "volumes": [
                    ""
                ],
                "volumesFrom": [
                    ""
                ],
                "portBindings": {
                    "": [
                        {
                            "hostIp": "",
                            "hostPort": ""
                        }
                    ]
                },
                "links": [
                    ""
                ],
                "user": "",
                "memory": 0,
                "cpuShares": 0,
                "lxcConf": [
                    {
                        "key": "",
                        "value": ""
                    }
                ],
                "privileged": false,
            }
        ],
        "version": ""
    }
}
```

Existing Server Configuration

```
{
    "cluster": {
        "servers": [
            {
                "hostname": "",
                "port": 0
            }
        ],
        "version": "",
        "upgradeTransition": {
            "mode": ""
```

```
            "sessionTimeout": 0
            "percentageTransitionPerRequest": 0
            "percentageTransitionPerRequest": 0
        }
    }
}
```

The JSON fields are used for the following reasons:

- **cluster.containers** - specifies the list of docker containers in the cluster.
- **cluster.containers[i].image** - name of the image to create each container from.
- **cluster.containers[i].tag** - the tag for the image, defaulting to 'latest'.
- **cluster.containers[i].alwaysPull** - whether to automatically check for new image versions.
- **cluster.containers[i].portToProxy** - the port the proxy routes HTTP request to
- **cluster.containers[i].name** - the name of the container, defaulting to an auto-generated unique name
- **cluster.containers[i].workingDir** - the current working directory inside the container's root file system
- **cluster.containers[i].entrypoint** - the container's entrypoint
- **cluster.containers[i].environment** - the environment variables that are set in the running container
- **cluster.containers[i].cmd** - the command to be executed when running the container
- **cluster.containers[i].hostname** - the container's hostname
- **cluster.containers[i].volumes** - mount volumes from either the host or from another docker container
- **cluster.containers[i].volumesFrom** - mount all the volumes defined for one or more other docker containers, including control over whether the volumes are mounted in read-write or read-only mode
- **cluster.containers[i].portBindings** - bind ports from within the container to one or more host port and IP combination, using format <port>/tcp or <port>/udp as the key
- **cluster.containers[i].portBindings[i].hostIp** - the host ip address to bind the port to
- **cluster.containers[i].portBindings[i].hostPort** - the host port to bind the port to
- **cluster.containers[i].links** - link the container to another Docker container
- **cluster.containers[i].user** - set the user user id and group id of the executing process running inside the container
- **cluster.containers[i].memory** - restrict the container's memory (in bytes)
- **cluster.containers[i].cpuShares** - restrict the container's cpu share using relative weight compared to other containers
- **cluster.containers[i].lxcConf** - add custom LXC options for the container
- **cluster.containers[i].lxcConf.key** - key (or name) for the custom LXC options for the container
- **cluster.containers[i].lxcConf.value** - value for the custom LXC options for the container
- **cluster.containers[i].privileged** - give extended privileges to the container
- **cluster.servers** - specifies the list of servers in the cluster.
- **cluster.servers[i].ip** - specifies the IP address or hostname of a server in the cluster.

- **cluster.servers[i].port** - specifies the port of a server in the cluster.
- **cluster.version** - specifies the cluster version.
- **cluster.upgradeTransition** - allows the configuration of the upgrade transition. If no *upgradeTransition* is specified, the upgrade transition mode defaults to INSTANT.
- **cluster.upgradeTransition.mode** - specifies the upgrade transition mode and support the following values: INSTANT, SESSION, GRADUAL and CONCURRENT.
- **cluster.upgradeTransition.sessionTimeout** - specifies the timeout period assigned to the SESSION transition mode.
- **cluster.upgradeTransition.percentageTransitionPerRequest** - specifies the transition percentage associated with each request in the GRADUAL transition mode.

When the user send a PUT request and successfully adds a new cluster, the proxy responds by returning a cluster id representing the new cluster entity that has been added. Appendix 8 demonstrates detailed examples of the PUT requests and responses supported by proxy and the type of values that can be used in each field of the message body.

5.5.2 GET - /configuration/cluster/{clusterId}

GET requests to */configuration/cluster/{clusterId}* gets a single cluster configuration. If no cluster id is specified and the GET request is sent to */configuration/cluster/*, a list of all the cluster configurations added to the proxy will be returned. The format of the response body for getting a cluster configuration with a specific cluster entity is shown below and detailed examples of GET requests and responses are demonstrated in Appendix 8.

```
{
    "cluster": {
        "servers": [
            {
                "hostname": "",
                "port": 0
            }
        ],
        "upgradeTransition": {
                "mode": ""
                "sessionTimeout": 0
                "percentageTransitionPerRequest": 0
        },
        "uuid": "",
        "version": ""
    }
}
```

5.5.3 DELETE - /configuration/cluster/{clusterId}

DELETE requests to */configuration/cluster/{clusterId}* deletes a single cluster configuration. No request body and response body is defined for the DELETE request. Appendix 8 shows an example of a DELETE request and the response received from the proxy.

5.6 Testing

As explained by the previous sections, the proxy consists of many components. Therefore, several testing strategies were used to suit the needs of each component (Figure xxxxxxx).

XXXXXXXX ADD TEST LEVELS DIAGRAM XXXXXXXXXXXXXXXXXXX

5.6.1 Unit Testing

These tests focus on low level checking of the functionality of each proxy component or function. Inversion-of-control and dependency-injection (Section 5.2.2) allowed dependencies to be injected into each component or function and therefore supported mocking of all dependencies. The unit tests cover both positive and negative cases resulting in overall of xxxxxxx line coverage and xxxxxxx branch coverage.

XXXXXXXX ADD COVERAGE REPORT XXXXXXXXXXXXXXXXXXX

5.6.2 Integration Testing

These tests focus on the integration between different components of the REST API and the proxy. To ensure each component of the REST API integrates correctly, tests were designed to send simultaneous PUT, GET and DELETE requests to the REST server and the responses were asserted. In addition, integration of different stages of the proxy was tested using different scenarios such as testing the behavior of when the proxy is configured for SESSION or CONCURRENT upgrade modes or when a cluster configuration is deleted from the proxy.

5.6.3 System Testing

The aim of these tests is to check that all components of the dynamic software update system including proxy and the REST API work together. System tests were written to cover complete user scenarios. The tests were executed by starting the proxy with an initial cluster configuration. The cluster configuration was then extended using the REST API. HTTP requests were then sent to the proxy to confirm that the proxy routed the requests to the correct clusters. Other scenarios also included deleting cluster configuration using the REST API and confirming the routing continued to behave as expected.

Chapter 6

Evaluation

This Chapter demonstrates the quantitative and qualitative analysis performed on the dynamic software update proxy presented in Chapter 5. The quantitative analysis focused on the load and saturation performance of the proxy; the speed that proxy applies an upgrade and rolls back to an older version when an upgraded version does not behave correctly. The qualitative analysis focused on the functional correctness and robustness of the proxy.

Real applications including Couchbase[1], WordPress[2] and Lighttpd[3] were used for the evaluations. Couchbase is a widely used high performing clustered NoSQL database and it is designed to handle large numbers of simultaneous requests. WordPress is a popular and mature content management system for writing blogs and websites based on PHP[4] and MySQL[5]. WordPress was used by more than 23% of the top 10 million websites as of September 2014[6]. Therefore, Couchbase and WordPress were ideal applications for evaluating the proxy. Lighttpd is a popular web server used by several high-traffic websites such as Wikipedia and YouTube. Even though Lighttpd is a populate web server, its bug tracking database[7] shows that it is not crash bug free. hence, in addition to Couchbase and WordPress, Lighttpd was used to test that the dynamic software update proxy is able to handle updates that cause incorrect behavior or crashes.

6.1 Quantitative Analysis

This section summarizes the load and saturation tests performed on the dynamic software update proxy. The performance of the proxy was also measured under multiple upgrades and different error conditions to evaluate how rapidly the proxy responds to an upgraded application version behaving incorrectly. The aim of these tests were to evaluate the performance and stability of the proxy.

6.1.1 Load Performance and Saturation Tests

The objectives of these evaluations were to test performance of the proxy for forwarding requests to different applications. Both saturation and load tests were performed on Coachbase and WordPress applications running within Docker containers inside a virtual machine by the proxy. The following proxy configurations were used to run WordPress and Coachbase Docker containers.

Proxy Configuration Running WordPress Docker Container

[1] http://www.couchbase.com/
[2] https://wordpress.com/
[3] http://www.lighttpd.net/
[4] http://php.net/
[5] http://www.mysql.com/
[6] http://w3techs.com/technologies/overview/content_management/all/
[7] http://redmine.lighttpd.net/issues/

```
{
    "proxy": {
        "port": 1235
    },
    "configService": {
        "port": 9090
    },
    "dockerHost": {
        "ip": "192.168.50.5",
        "port": 2375
    },
    "cluster": {
        "containers": [
            {
                "image": "mysql",
                "tag": "latest",
                "name": "some-mysql",
                "environment": [
                    "MYSQL_ROOT_PASSWORD=mysecretpassword"
                ],
                "volumes": [
                    "/var/lib/mysql:/var/lib/mysql"
                ]
            },
            {
                "image": "wordpress",
                "tag": "3.9.1",
                "portToProxy": 8080,
                "name": "some-wordpress",
                "links": [
                    "some-mysql:mysql"
                ],
                "portBindings": {
                    "80/tcp": [
                        {
                            "HostIp": "0.0.0.0",
                            "HostPort": "8080"
                        }
                    ]
                }
            }
        ],
        "version": "3.9.1"
    }
}
```

Proxy Configuration Running Couchbase Docker Container

```
{
    "proxy": {
        "port": 1235
    },
    "configService": {
        "port": 9090
    },
    "dockerHost": {
        "ip": "192.168.50.5",
        "port": 2375
    },
```

```
    "cluster": {
        "containers": [
            {
                "image": "couchbase",
                "tag": "2.5.1",
                "name": "couch_one",
                "portToProxy": 8091,
                "environment": [
                    "CLUSTER_INIT_USER=Administrator",
                    "CLUSTER_INIT_PASSWORD=password",
                    "SAMPLE_BUCKETS=\"beer-sample\""
                ],
                "portBindings": {
                    "8091/tcp": [
                        {
                            "HostIp": "0.0.0.0",
                            "HostPort": "8091"
                        }
                    ]
                }
            }
        ],
        "version": "2.5.1"
    }
}
```

Saturation tests focused on stability of the proxy by sending HTTP requests ($>$ 150 req/sec for WordPress and $>$ 400 req/sec for Couchbase) to the proxy for 10 hours. The performance tests focused on measuring the overhead added by the proxy to process HTTP requests and responses for 10 seconds, 10 minutes and one hour.

For both saturation and load tests, requests were sent to WordPress and Couchbase directly and via the proxy. Figure 6.1 and Figure 6.3 show the latency in milliseconds added when sending requests to the applications via the proxy in comparison to sending requests directly. Figure 6.2 and Figure 6.4 show the latency added when sending requests via the proxy as a percentage of the latency when sending requests directly to the applications. All performance figures show the latency using a percentile distribution; the percentage indicates the proportion of the requests that have the recorded latency or lower. For example, a latency of 40 milliseconds at the 99$^{\text{th}}$ percentile indicates that 99% of the requests have a 40 milliseconds latency or less.

The evaluation results from WordPress indicated that in the majority of the time processing requests via proxy was significantly more efficient (Figure 6.1 and Figure 6.1). It is not fully clear why processing requests via the proxy is faster than using WordPress directly. These findings require further investigation, however, a hypothesis is that the proxy manages connections more efficiently than WordPress using Apache HTTP Server. This is because the proxy is written in the Go programming language and uses goroutines[8] instead of threads to handle each requests. The proxy uses two goroutines per request and response; one goroutine is used for the request and one goroutine is used for the response. However when serving a PHP page, Apache HTTP Server uses a separate thread to handle each request [9]. Goroutines are executed using a thread pool and therefore are not one to one with a thread. This means that the proxy will generally not create a new thread for each request instead an existing thread will be used from the thread pool. As a result the proxy does not close connections as aggressively as Apache HTTP Server. In addition, Apache HTTP Server uses a separate thread per request and therefore cannot maintain as many open connections. The proxy therefor can send more requests from the same client down the same connection, as long

[8] http://golangtutorials.blogspot.co.uk/2011/06/goroutines.html
[9] http://httpd.apache.org/docs/current/mod/mpm_common.html

as the client does not close the connection. This is particularly true for HTTP/1.1 which supports connection re-uses when the "Connection" header has a value of "keep-alive".

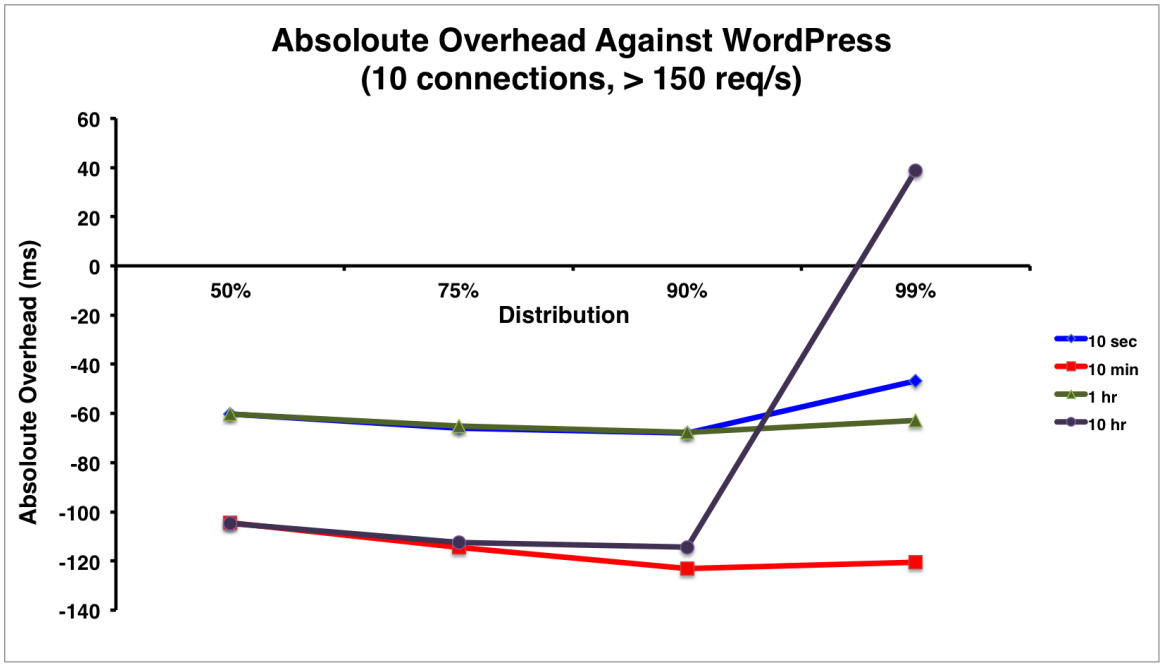

Figure 6.1: Absolute overhead added by the proxy against WordPress

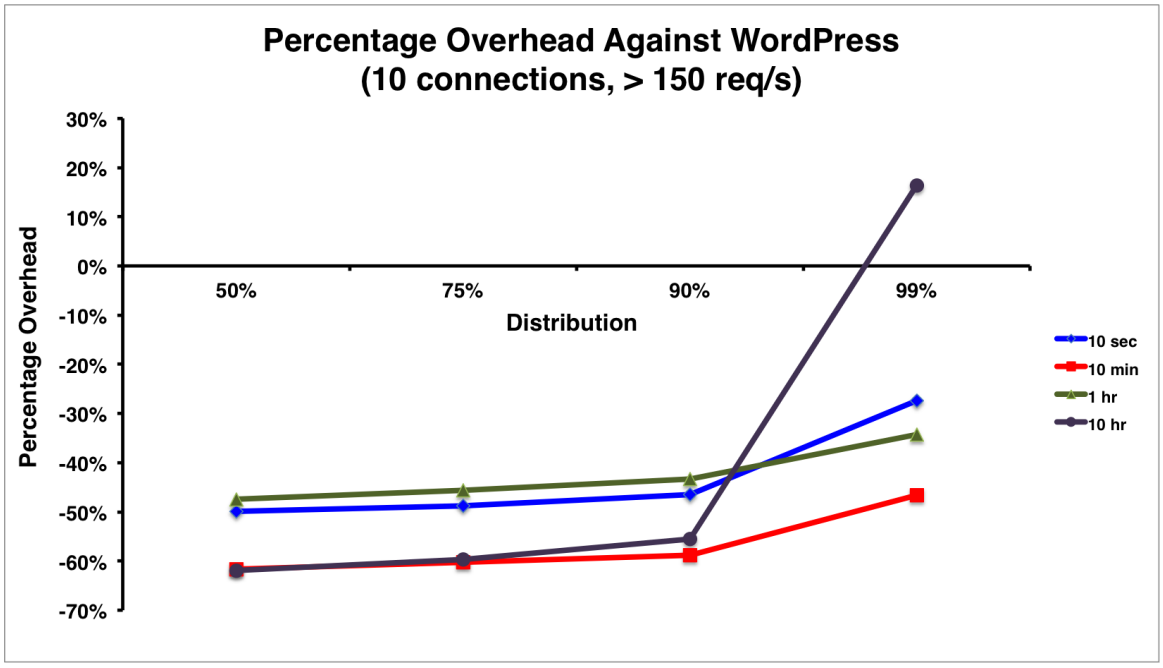

Figure 6.2: Percentage overhead added by the proxy against WordPress

Moreover, figure 6.1 and Figure 6.2 shows that the proxy has a greater standard deviation for response times, because as the percentile increases up to 99%, the overhead added by the proxy becomes greater. Go garbage collector and other sporadic activities happening at a low level which may increase the latency inside the Go application could be an explanation for these findings.

Similar to WordPress, Couchbase results also showed that processing requests via the proxy was mainly more efficient than processing them directly by Couchbase, or if any overhead was added by the proxy it was insignificant.

Couchbase performance and saturation tests showed that the proxy added a small over head

(<6%) when it was run for shorter periods of time. However, the proxy significantly increased the performance of Couchbase when it was run for medium and long periods of time (Figure 6.3 and Figure 6.4). Although further analysis is required to justify these findings, one hypothesis could be that Couchbase is more efficient at managing requests for a short period because it is keeping connections open. However after a period of time (>10 seconds), it runs out of resources and has to start closing connections more aggressively, therefore, its efficiency at handling connections reduces.

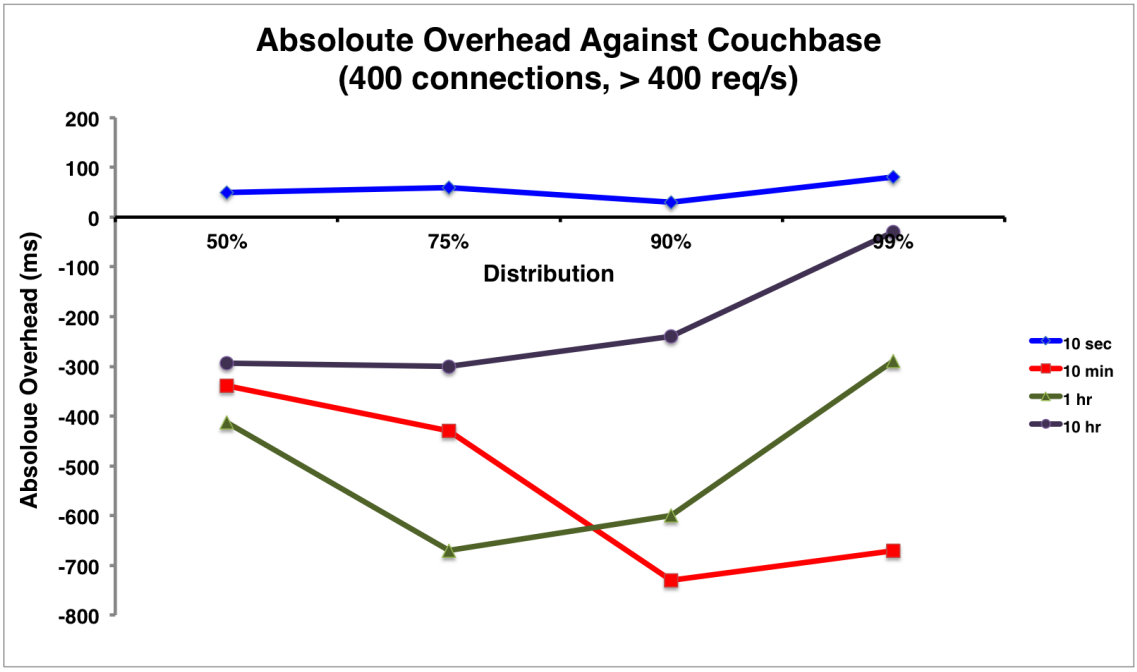

Figure 6.3: Absolute overhead added by the proxy against Couchbase

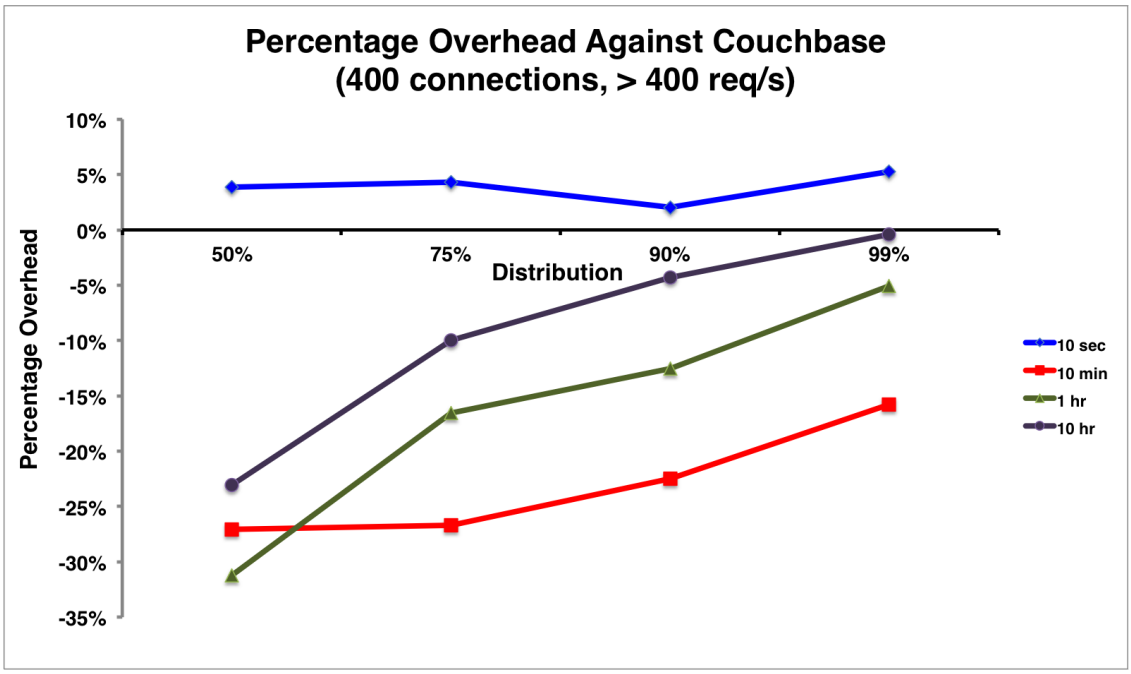

Figure 6.4: Percentage overhead added by the proxy against Couchbase

6.1.2 Latency in Response to Updated Version Failure

The aim of this evaluation was to measure how rapidly the proxy applies an upgrade and rolls back to an older version of the application when the new version does not behave correctly.

Table 6.1: Proxy Latency for dynamic upgrade and error recovery

Upgrade Scenarios	**Latency (ms)**	**+\- SD (ms)**
INSATNT Upgrade	44.00	1.36
INSTANT Upgrade Failure	45.00	0.75
CONCURRENT Upgrade	44.00	0.50
CONCURRENT Upgrade Failure	44.00	0.49

INSTANT and CONCURRENT upgrades (Section 4.2.2) were applied on version 3.9.1 and 3.9.2 of the WordPress application. Each version of the WorlPress application was run within a Docker container inside a virtual machine by the proxy as demonstrated in Section 6.1.1.

The proxy was initially configured to forward responses to the WordPress application version 3.9.1. The proxy was then configured to perform an upgrade to WordPress version 3.9.2 using either INSTANT or CONCURRENT mode. The latency for processing HTTP requests by WordPress via the proxy was recorded. The latency was also recorded for the scenario when the upgraded version failed (Table 6.1). The failure was simulated by stopping the upgraded version after the proxy had been configured to perform the upgrade. This test therefore measured the latency for the proxy to roll back a failed upgrade using INSTANT or CONCURRENT mode. SESSION and GRADUAL modes were not tested because they behave identically to INSTANT mode when a failure in an upgraded version occurs.

As shown in Table 6.1, there is no additional latency introduced by the proxy when rolling back in response to a failure in an upgraded application version.

6.2 Qualitative Analysis

Qualitative analysis were performed on the dynamic software update proxy (Chapter 5) using real applications. These evaluations insured that different features of the proxy such as load balancing and dynamic software upgrade functionality behave correctly and the proxy can survive crash bugs in real examples. The following qualitative analysis were performed:

- Load Balancing Analysis - To ensure that the proxy acts as a reverse proxy and distributes incoming traffics across a number of servers within a cluster.
- Real Software Upgrades - To prove the ability of the proxy to handle different upgrade strategies (*i.e.* INSTANT, SESSION, GRADUAL and CONCURRENT) described in Section 4.2.2 in real applications.
- Handling Incorrect Behavior - To ensure that the proxy is able to handle updates that cause incorrect behavior and allows systems to survive bugs using real examples.

6.2.1 Load Balancing Analysis

The dynamic software update proxy is designed to behave as a load balancer and distribute HTTP requests across multiple servers. Therefore, to test the load balancing feature of the dynamic software update proxy, HTTP requests were sent to a Couchbase cluster via the proxy with each node running in a separate Docker container. As shown in (Figure 6.5) the cluster contained three nodes running Couchbase server 2.5.1. Couchbase servers were chosen for the load balancing evaluation since they form a cluster with shared data. Therefore each server returns an identical response to a given request.

Each Couchbase node was run within a Docker container inside a virtual machine as shown below. Internally within the virtual machine each Docker container for each Couchbase node used

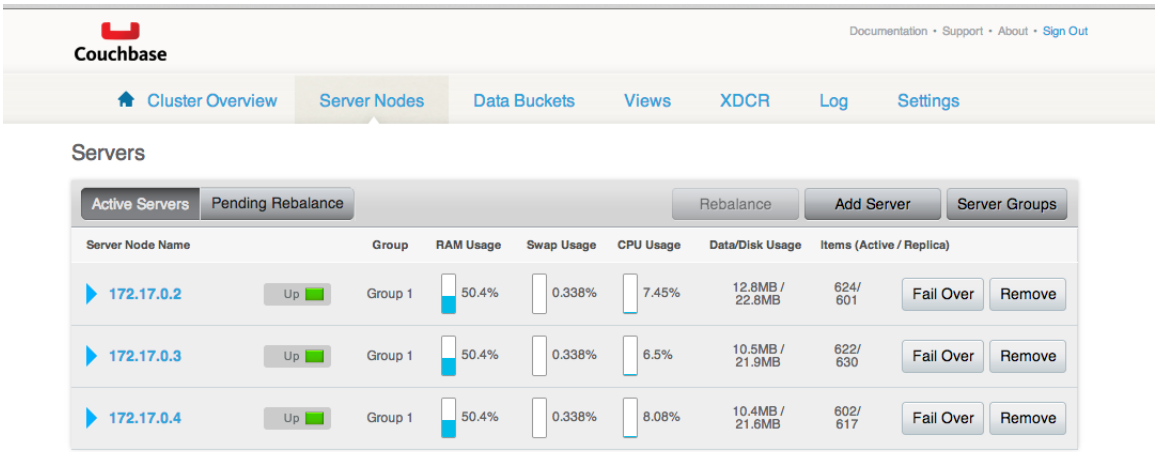

Figure 6.5: Couchbase cluster with three servers

a different internal IP address on the 172.16.0.0 - 172.31.255.255 private range reserved by IANA[10]. However, as the port has been mapped from each Docker container to the virtual machine, each node can be addressed on a different port of the virtual machine. The following section summarizes the steps executed for load balancing evaluation.

```
{
    "proxy": {
        "port": 1235
    },
    "configService": {
        "port": 9090
    },
    "dockerHost": {
        "ip": "192.168.50.5",
        "port": 2375
    },
    "cluster": {
        "containers": [
            {
                "image": "couchbase",
                "tag": "2.5.1",
                "name": "node_one",
                "portToProxy": 8091,
                "environment": [
                    "CLUSTER_INIT_USER=Administrator",
                    "CLUSTER_INIT_PASSWORD=password",
                    "SAMPLE_BUCKETS=\"beer-sample\""
                ],
                "portBindings": {
                    "8091/tcp": [
                        {
                            "HostIp": "0.0.0.0",
                            "HostPort": "8091"
                        }
                    ]
                }
            },
            {
```

[10] http://www.iana.org/assignments/iana-ipv4-special-registry/iana-ipv4-special-registry.xhtml

```
                "image": "couchbase",
                "tag": "2.5.1",
                "name": "node_two",
                "portToProxy": 9091,
                "environment": [
                    "CLUSTER_INIT_USER=Administrator",
                    "CLUSTER_INIT_PASSWORD=password"
                ],
                "links": [
                    "node_one:couchbase"
                ],
                "portBindings": {
                    "8091/tcp": [
                        {
                            "HostIp": "0.0.0.0",
                            "HostPort": "9091"
                        }
                    ]
                }
            },
            {
                "image": "couchbase",
                "tag": "2.5.1",
                "name": "node_three",
                "portToProxy": 10091,
                "environment": [
                    "CLUSTER_INIT_USER=Administrator",
                    "CLUSTER_INIT_PASSWORD=password"
                ],
                "links": [
                    "node_one:couchbase"
                ],
                "portBindings": {
                    "8091/tcp": [
                        {
                            "HostIp": "0.0.0.0",
                            "HostPort": "10091"
                        }
                    ]
                }
            }
        ],
        "version": "2.5.1"
    }
}
```

Proxy Configuration

The proxy was configured to forward HTTP requests to three nodes in Couchbase cluster each running in separate Docker containers (Figure 6.5) using round-robin load balancing. The start-up configuration was echoed back by the proxy as shown in the following terminal output.

```
2014/08/24 19:48:18.309757 config_parse.go:53:   NOTICE - Parsed config file:
{
   "proxy": {
      "port": 1235
   },
   "configService": {
```

```
      "port": 9090
   },
   "dockerHost": {
      "ip": "192.168.50.5",
      "port": 2375
   },
   "cluster": {
      "containers": [
         {
            "image": "couchbase",
            "tag": "2.5.1",
            "portToProxy": 8091,
            "name": "node_one",
            "environment": [
               "CLUSTER_INIT_USER=Administrator",
               "CLUSTER_INIT_PASSWORD=password",
               "SAMPLE_BUCKETS=\"beer-sample\""

            ],
            "portBindings": {
               "8091/tcp": [
                  {
                     "HostIp": "0.0.0.0",
                     "HostPort": "8091"
                  }
               ]
            }
         },

         {
            "image": "couchbase",
            "tag": "2.5.1",
            "name": "node_two",
            "portToProxy": 9091,
            "environment": [
               "CLUSTER_INIT_USER=Administrator",
               "CLUSTER_INIT_PASSWORD=password"
            ],
            "links": [
               "node_one:couchbase"
            ],
            "portBindings": {
               "8091/tcp": [
                  {
                     "HostIp": "0.0.0.0",
                     "HostPort": "9091"
                  }
               ]
            }
         },
         {
            "image": "couchbase",
            "tag": "2.5.1",
            "name": "node_three",
            "portToProxy": 10091,
            "environment": [
               "CLUSTER_INIT_USER=Administrator",
               "CLUSTER_INIT_PASSWORD=password"
            ],
            "links": [
               "node_one:couchbase"
```

```
            ],
            "portBindings": {
               "8091/tcp": [
                  {
                     "HostIp": "0.0.0.0",
                     "HostPort": "10091"
                  }
               ]
            }
         }
      ],
      "version": "2.5.1"
   }
}
as:
Proxy{
        Proxy Address:      0.0.0.0:1235
        ConfigService Port: 9090
        Proxied Servers:
                version: [2.5.1]
                [192.168.50.10:8091, 192.168.50.10:9091, 192.168.50.10:10091]
}
```

HTTP Requests

Simultaneous HTTP requests were sent to the running proxy using cURL[11].

```
curl -v \
http://Administrator:password@127.0.0.1:1235/pools/default/buckets/beer-sample
```

Load balancing proxy

When the proxy is configured to run with INFO level logging, it prints log entries to the terminal. The following log entries created by the proxy demonstrate that the HTTP requests were distributed between each node in the cluster using round-robin order.

```
2014/08/24 19:50:06.560279 proxy_context_cluster.go:104:
   INFO - Serving response 0 from ip: [192.168.50.10] port:  [8091]
   version: [2.5] mode: [INSTANT]

2014/08/24 19:52:34.576970 proxy_context_cluster.go:104:
   INFO - Serving response 1 from ip: [192.168.50.10] port:  [9091]
   version: [2.5] mode: [INSTANT]

2014/08/24 19:56:00.452493 proxy_context_cluster.go:104:
   INFO - Serving response 2 from ip: [192.168.50.10] port:  [10091]
   version: [2.5] mode: [INSTANT]

2014/08/24 19:56:07.619419 proxy_context_cluster.go:104:
   INFO - Serving response 3 from ip: [192.168.50.10] port:  [8091]
   version: [2.5] mode: [INSTANT]

2014/08/24 19:56:09.137194 proxy_context_cluster.go:104:
   INFO - Serving response 4 from ip: [192.168.50.10] port:  [9091]
   version: [2.5] mode: [INSTANT]
```

[11] http://curl.haxx.se/

```
2014/08/24 19:56:19.655983 proxy_context_cluster.go:104:
   INFO - Serving response 5 from ip: [192.168.50.10] port:  [10091]
   version: [2.5] mode: [INSTANT]
```

HTTP Requests & Responses

The following shows an HTTP request and the associated HTTP response received via the proxy from a node in the Couchbase cluster. Note the time zone used inside the Docker container was set to UTC (Coordinated Universal Time) whereas the time zone on the computer running the proxy was set to BST (British Summer Time). Therefore, terminal log output was 1 hour ahead of the "Date" header in the responses from Couchbase.

Request

```
GET /pools/default/buckets/beer-sample HTTP/1.1
Authorization: Basic QWRtaW5pc3RyYXRvcjpwYXNzd29yZA==
User-Agent: curl/7.30.0
Host: 127.0.0.1:1235
Accept: */*
```

Response

```
HTTP/1.1 200 OK
Set-Cookie: dynsoftup=369f06ad-2bbf-11e4-9034-28cfe9158b63;
Server: Couchbase Server
Pragma: no-cache
Date: Sun, 24 Aug 2014 18:50:06 GMT
Content-Type: application/json
Content-Length: 10972
Cache-Control: no-cache

{"name":"beer-sample","bucketType":"membase", ... }
```

6.2.2 Real Software Upgrades

The main feature of the implemented proxy described in Chapter 5 is its ability to support dynamic software update for continues delivery into production. The proxy is able to dynamically update an application to a new version. It then runs different versions of an application concurrently and it automatically monitors the behavior of the most updated version. If the updated version behaves incorrectly, the proxy seamlessly removes that version of the application and routes all requests to the previous stable version with no downtime. This section evaluates the aforementioned functionality of the proxy in details using real applications.

Dynamic software upgrade strategies described in Section 4.2.2 were applied on two versions of WordPress (version 3.9.1 and version 3.9.2) application. Each version of WorlPress application was run within a Docker container inside a virtual machine by the proxy as demonstrated in Section 6.1.1.

For each upgrade scenarios (i.e. INSTANT, SESSION, GRADUAL and CONCURRENT), the following actions were executed sequentially:

- Proxy was initially configured with WordPress application version 3.9.1.

- HTTP requests were sent to the proxy.
- Proxy's behavior and the responses received from the proxy were evaluated.
- An upgrade from WordPress version 3.9.1 to version 3.9.2 was applied by sending PUT HTTP requests to the proxy's configuration service
- Proxy's response to the upgrade was evaluated.
- HTTP requests were sent to the proxy and the responses were evaluated for the upgrade from WordPress version 3.9.1 to version 3.9.2.
- The Docker container running the new version of WordPress was stopped to simulate a failure and proxy's response to the failure was evaluated

INSTANT Upgrade

In an INSTANT upgrade the proxy forwards requests to the most updated version of the application. However, if the most updated version does not behave correctly, the proxy removes that version and forwards requests to the previous version of the application (Section 4.2.2). This section explains the process executed to evaluate the INSTANT upgrade mode of the proxy.

Proxy Initial Configuration - The proxy was configured to forward HTTP requests to a cluster running version 3.9.1 of the WordPress application. The start-up configuration echoed by proxy is shown in the following terminal output.

```
2014/08/25 20:22:24.922144 config_parse.go:53:   NOTICE - Parsed config file:
{
   "proxy": {
      "port": 1235
   },
   "configService": {
      "port": 9090
   },
   "dockerHost": {
      "ip": "192.168.50.5",
      "port": 2375
   },
   "cluster": {
      "containers": [
         {
            "image": "mysql",
            "tag": "latest",
            "name": "some-mysql",
            "environment": [
               "MYSQL_ROOT_PASSWORD=mysecretpassword"
            ],
            "volumes": [
               "/var/lib/mysql:/var/lib/mysql"
            ]
         },
         {
            "image": "wordpress",
            "tag": "3.9.1",
            "portToProxy": 8080,
            "name": "some-wordpress",
            "links": [
               "some-mysql:mysql"
            ],
            "portBindings": {
               "80/tcp": [
```

```
                    {
                        "HostIp": "0.0.0.0",
                        "HostPort": "8080"
                    }
                ]
            }
        }
    ],
    "version": "3.9.1"
  }
}
as:
Proxy{
        Proxy Address:      0.0.0.0:1235
        ConfigService Port: 9090
        Proxied Servers:    [version: 3.9.1] [192.168.50.5:8080]
}
```

HTTP Requests & Responses - HTTP requests were sent to the proxy using the following cURL syntax:

```
curl -v http://127.0.0.1:1235
```

When the proxy was run in the INFO level logging, its output to the terminal demonstrated that the HTTP requests were forwarded to the version 3.9.1 of the WordPress. The IP address and the version of the sever that proxy is serving responses from refers to WordPress version 3.9.1 demonstrated in the initial configuration.

```
2014/08/25 20:22:56.415432 proxy_context_cluster.go:114:
   INFO - Serving response 0 from [192.168.50.5] port: [8080]
   version: [3.9.1] mode: [INSTANT]
```

This was further confirmed by the response received from WordPress application via proxy. The "X-Powered-By" and "X-Pingback" headers refer to the PHP version and the IP address of WordPress version 3.9.1, respectively. Note that when a cluster has an INSTANT upgrade mode, the cookie in "Set-Cookie" header of the response associated with that cluster does not have an expiry date.

```
HTTP/1.1 200 OK
Set-Cookie: dynsoftup=24e90b08-2c8d-11e4-a649-28cfe9158b63;
Date: Mon, 25 Aug 2014 19:22:57 GMT
Server: Apache/2.4.9 (Debian)
X-Powered-By: PHP/5.5.12-2
X-Pingback: http://192.168.50.5:8080/xmlrpc.php
Vary: Accept-Encoding
Content-Length: 7467
Content-Type: text/html; charset=UTF-8

<!DOCTYPE html> ... </html>
```

PUT Request For INSTANT Upgrade - For an instant upgrade of the WordPress application from version 3.9.1. to version 3.9.2, a cluster containing new version was added to the proxy using the following PUT request:

```
curl -v \
```

```
http://127.0.0.1:9090/configuration/cluster -X PUT \
-H 'Content-Type: application/json' \
-d @config_for_curl/config_curl_docker_wordpress_INSTANT.json
```

The proxy acknowledged addition of the new cluster by outputting the following message in the terminal:

```
2014/08/25 20:25:16.080385 config_service.go:66:
   INFO - Received new cluster configuration:
   {
      "cluster": {
         "containers": [
            {
               "environment": [
                  "MYSQL_ROOT_PASSWORD=mysecretpassword"
               ],
               "image": "mysql",
               "name": "some-mysql-upgrade",
               "tag": "latest",
               "volumes": [
                  "/var/lib/mysql:/var/lib/mysql"
               ]
            },
            {
               "image": "wordpress",
               "links": [
                  "some-mysql:mysql"
               ],
               "name": "some-wordpress-upgrade",
               "portBindings": {
                  "80/tcp": [
                     {
                        "HostIp": "0.0.0.0",
                        "HostPort": "8081"
                     }
                  ]
               },
               "portToProxy": 8081,
               "tag": "3.9.2"
            }
         ],
         "version": "3.9.2",
         "upgradeTransition": {
            "mode": "INSTANT"
         }
      }
   }
```

The PUT request and its associated HTTP response is shown below. The response received from the proxy contained the id of the newly added cluster.

Request

```
PUT /configuration/cluster HTTP/1.1
User-Agent: curl/7.30.0
Host: 127.0.0.1:9090
Accept: */*
Content-Length: 126
```

```
Content-Type: application/x-www-form-urlencoded
```

Response

```
HTTP/1.1 202 Accepted
Date: Mon, 25 Aug 2014 19:25:16 GMT
Content-Length: 36
Content-Type: text/plain; charset=utf-8

8aedb902-2c8d-11e4-a649-28cfe9158b63
```

Evaluation of the INSTANT Upgrade - To check that the proxy applies the upgrade immediately and process the requests with the most updated version, HTTP requests were sent to the proxy and responses received from the proxy were analyzed.

The proxy's output to the terminal showed that the proxy is serving responses from version 3.9.2 of the WordPress application. In addition, the "X-Powered-By" and "X-Pingback" headers in the response recieved from proxy after the upgrade refered to the PHP version and the IP address of version 3.9.2 of WordPress application, respectively.

HTTP Request After INSTANT Upgrade

```
curl -v http://127.0.0.1:1235
```

Proxy Log Output After INSTANT Upgrade

```
2014/08/25 20:28:20.400747 proxy_context_cluster.go:114:
   INFO - Serving response 0 from ip: ip: [192.168.50.5] port: [8081]
   version: [3.9.2] mode: [INSTANT]
```

Response from WordPress Version 3.9.2

```
HTTP/1.1 200 OK
Set-Cookie: dynsoftup=8aedb902-2c8d-11e4-a649-28cfe9158b63;
Date: Mon, 25 Aug 2014 19:28:21 GMT
Server: Apache/2.2.22 (Debian)
X-Powered-By: PHP/5.4.4-14+deb7u12
X-Pingback: http://192.168.50.5:8081/xmlrpc.php
Vary: Accept-Encoding
Content-Length: 7467
Content-Type: text/html; charset=UTF-8

<!DOCTYPE html> ... </html>
```

Handling Incorrect Behavior - To assure that the proxy can seamlessly handle updates that cause incorrect behavior, the cluster running version 3.9.2 of WordPress was stopped and the behavior of the proxy was evaluated after sending new HTTP requests to the proxy. The following terminal output demonstrates that the proxy first tried to send requests to the cluster with the highest version. When proxy detected that the cluster is not behaving correctly, in this case it could not communicate with the cluster, the proxy removed the updated version of the cluster from the list of configured clusters and forwarded requests to the previous cluster version.

Proxy Log Output After WordPress Version 3.9.2 crashed

```
2014/08/25 20:32:59.897032 proxy_context_cluster.go:114:
   INFO - Serving response 1 from ip: ip: [192.168.50.5] port: [8081]
   version: [3.9.2] mode: [INSTANT]

2014/08/25 20:32:59.897433 stage_route.go:41:
   ERROR - Error communicating with server - dial tcp 192.168.50.5:8081:  connection refused

2014/08/25 20:32:59.899682 stage_route.go:42:
   WARNING - Removing cluster from configuration - version:  3.9.2 [192.168.50.5:8081]

2014/08/25 20:32:59.899707 proxy_context_cluster.go:114:
   INFO - Serving response 1 from [192.168.50.5] port: [8080]
   version: [3.9.1] mode: [INSTANT]
```

Response After WordPress Version 3.9.2 crashed

```
HTTP/1.1 200 OK
Set-Cookie: dynsoftup=24e90b08-2c8d-11e4-a649-28cfe9158b63;
Date: Mon, 25 Aug 2014 19:33:00 GMT
Server: Apache/2.4.9 (Debian)
X-Powered-By: PHP/5.5.12-2
X-Pingback: http://192.168.50.5:8080/xmlrpc.php
Vary: Accept-Encoding
Content-Length: 7467
Content-Type: text/html; charset=UTF-8

<!DOCTYPE html> ... </html>
```

SESSION Upgrade

In the SESSION upgrade the proxy generates cookies with a configured expiry time for requests forwarded to that cluster. An HTTP client returns the cookie as long as it is not expired. This insures that the client stays on the same cluster until it has not interacted with the proxy for the duration of the cookie. As with INSTANT update, if the cluster with SESSION upgrade does not behave correctly, it will be removed and requests will be forwarded to the previous version of the application (Section 4.2.2). This section explains the process executed to evaluate the proxy for SESSION upgrade.

Proxy Initial Configuration - The proxy was configured with a cluster running version 3.9.1 of WordPress application. The start-up configuration echoed by proxy is shown in the following terminal output.

```
2014/08/25 21:10:46.064526 config_parse.go:53:   NOTICE - Parsed config file:
{
   "proxy": {
      "port": 1235
   },
   "configService": {
      "port": 9090
   },
   "dockerHost": {
      "ip": "192.168.50.5",
      "port": 2375
   },
   "cluster": {
      "containers": [
```

```
         {
            "image": "mysql",
            "tag": "latest",
            "name": "some-mysql",
            "environment": [
               "MYSQL_ROOT_PASSWORD=mysecretpassword"
            ],
            "volumes": [
               "/var/lib/mysql:/var/lib/mysql"
            ]
         },
         {
            "image": "wordpress",
            "tag": "3.9.1",
            "portToProxy": 8080,
            "name": "some-wordpress",
            "links": [
               "some-mysql:mysql"
            ],
            "portBindings": {
               "80/tcp": [
                  {
                     "HostIp": "0.0.0.0",
                     "HostPort": "8080"
                  }
               ]
            }
         }
      ],
      "version": "3.9.1"
   }
}
as:
Proxy{
        Proxy Address:      0.0.0.0:1235
        ConfigService Port: 9090
        Proxied Servers:    version: [3.9.1] [192.168.50.5:8080]
}
```

HTTP Requests & Responses - HTTP requests were sent to the WordPress cluster via proxy using the following cURL syntax:

```
curl -v http://127.0.0.1:1235
```

Since no upgrade transition was defined for the initial configuration, INSTANT mode was assigned to the cluster with version 3.9.1 as shown in the following terminal output echoed by proxy. In addition, The output shows that the proxy is forwarding responses from WordPress with version 3.9.1.

```
2014/08/25 21:11:18.216192 proxy_context_cluster.go:114:
   INFO - Serving response 0 from [192.168.50.5] port: [8080]
   version: [3.9.1] mode: [INSTANT]
```

The "X-Powered-By" and "X-Pingback" headers received by the client also confirmed that the response has been sent by WordPress version 3.9.1. Note that the Set-Cookie header in the response is set and it has no expiry date.

```
HTTP/1.1 200 OK
Set-Cookie: dynsoftup=e6203784-2c93-11e4-88f5-28cfe9158b63;
Date: Mon, 25 Aug 2014 20:11:18 GMT
Server: Apache/2.4.9 (Debian)
X-Powered-By: PHP/5.5.12-2
X-Pingback: http://192.168.50.5:8080/xmlrpc.php
Vary: Accept-Encoding
Content-Length: 7467
Content-Type: text/html; charset=UTF-8

<!DOCTYPE html> ... </html>
```

PUT Request For SESSION Upgrade - A cluster running version 3.9.2 of WordPress application was added to the proxy with SESSION upgrade transition mode and 60 seconds session time out using the following PUT request:

```
curl -v \
http://127.0.0.1:9090/configuration/cluster -X PUT \
-H 'Content-Type: application/json' \
-d @config_for_curl/config_curl_docker_wordpress_SESSION.json
```

The proxy acknowledged addition of the new cluster by outputting the following message in terminal when it ran in the INFO logging level:

```
2014/08/25 21:14:10.079590 config_service.go:66:
   INFO - Received new cluster configuration:
   {
      "cluster": {
         "containers": [
            {
               "environment": [
                  "MYSQL_ROOT_PASSWORD=mysecretpassword"
               ],
               "image": "mysql",
               "name": "some-mysql-upgrade",
               "tag": "latest",
               "volumes": [
                  "/var/lib/mysql:/var/lib/mysql"
               ]
            },
            {
               "image": "wordpress",
               "links": [
                  "some-mysql:mysql"
               ],
               "name": "some-wordpress-upgrade",
               "portBindings": {
                  "80/tcp": [
                     {
                        "HostIp": "0.0.0.0",
                        "HostPort": "8081"
                     }
                  ]
               },
               "portToProxy": 8081,
               "tag": "3.9.2"
            }
```

```
        ],
        "version": "3.9.2",
        "upgradeTransition": {
            "mode": "SESSION",
            "sessionTimeout": 60
        }
    }
}
```

The PUT request and its associated HTTP response is shown below. The response contains cluster id of the newly added cluster.

Request

```
PUT /configuration/cluster HTTP/1.1
User-Agent: curl/7.30.0
Host: 127.0.0.1:9090
Accept: */*
Content-Length: 148
Content-Type: application/x-www-form-urlencoded
```

Response

```
HTTP/1.1 202 Accepted
Date: Mon, 25 Aug 2014 20:14:10 GMT
Content-Length: 36
Content-Type: text/plain; charset=utf-8

5fba791f-2c94-11e4-88f5-28cfe9158b63s
```

HTTP Requests Without Cookie - To check the proxy behaves correctly with SESSION upgrade, an HTTP request with no cookie was sent to the proxy. The proxy forwarded the HTTP request to the new cluster as shown in the following terminal output:

```
2014/08/25 21:17:22.866773 proxy_context_cluster.go:114:
   INFO - Serving response 0 from ip: ip: [192.168.50.5] port: [8081]
   version: [3.9.2] mode: [SESSION]
   session timeout [60] uuid [5fba791f-2c94-11e4-88f5-28cfe9158b63]
```

Analyzing headers in the response (*i.e.* "X-Powered-By" and "X-Pingback") also confirmed that responses were coming from the new cluster running version 3.9.2 of WordPress application. Note that the Set-Cookie header in the response from a cluster with SESSION upgrade mode has an expiry date.

```
HTTP/1.1 200 OK
Set-Cookie: dynsoftup=5fba791f-2c94-11e4-88f5-28cfe9158b63; Expires=Mon, 25
    Aug 2014 21:18:22 BST;
Date: Mon, 25 Aug 2014 20:17:23 GMT
Server: Apache/2.2.22 (Debian)
X-Powered-By: PHP/5.4.4-14+deb7u12
X-Pingback: http://192.168.50.5:8081/xmlrpc.php
Vary: Accept-Encoding
Content-Length: 7467
Content-Type: text/html; charset=UTF-8
```

```
<!DOCTYPE html> ... </html>
```

HTTP Requests With Cookie - When HTTP requests were sent to the proxy with cookies associated to an older cluster, the proxy forwarded those requests to the old cluster rather than the new one. This is demonstrated with the following terminal output echoed by proxy and the response received by the client.

HTTP request with cookie

```
curl -v 'http://127.0.0.1:1235/' \
-H 'Cookie:   dynsoftup=e6203784-2c93-11e4-88f5-28cfe9158b63;'
```

Proxy Log Output

```
2014/08/25 21:24:17.273012 proxy_context_cluster.go:114:
   INFO - Serving response 1 from [192.168.50.5] port: [8080]
   version: [3.9.1] mode: [INSTANT]
```

Response

```
HTTP/1.1 200 OK
Set-Cookie: dynsoftup=e6203784-2c93-11e4-88f5-28cfe9158b63; Expires=Mon, 25
    Aug 2014 21:24:17 BST;
Date: Mon, 25 Aug 2014 20:24:17 GMT
Server: Apache/2.4.9 (Debian)
X-Powered-By: PHP/5.5.12-2
X-Pingback: http://192.168.50.5:8080/xmlrpc.php
Vary: Accept-Encoding
Content-Length: 7467
Content-Type: text/html; charset=UTF-8

<!DOCTYPE html> ... </html>
```

Handling Incorrect Behavior - To insure the proxy can handle any errors occurring by the updated cluster, the Docker container running WordPress version 3.9.2 was stopped. HTTP requests subsequently were sent to the proxy and its behavior was screened. The proxy terminal output demonstrated that when the proxy fails to send requests to the most updated cluster, it removes that cluster and forwards the requests to the previous cluster version.

```
2014/08/25 21:35:10.120521 proxy_context_cluster.go:114:
   INFO - Serving response 1 from ip: ip: [192.168.50.5] port: [8081]
   version: [3.9.2] mode: [SESSION] session timeout [60]
   uuid [5fba791f-2c94-11e4-88f5-28cfe9158b63]

2014/08/25 21:35:10.120978 stage_route.go:41:
    ERROR - Error communicating with server - dial tcp 192.168.50.5:8081:  connection refused

2014/08/25 21:35:10.120999 stage_route.go:42:
    WARNING - Removing cluster from configuration - version:  3.9.2 [192.168.50.5:8081]

2014/08/25 21:35:10.121034 proxy_context_cluster.go:114:
   INFO - Serving response 3 from [192.168.50.5] port: [8080]
   version: [3.9.1] mode: [INSTANT]
```

The response received from WordPress application via proxy after the oldest version failed is shown below. Note that the "X-Pingback" header shows the network address of old version of WordPress application. In addition, Since proxy deleted the cluster with SESSION upgrade mode, the uuid cookie in the "Set-Cookie" header does not have an expiry date.

```
HTTP/1.1 200 OK
Set-Cookie: dynsoftup=e6203784-2c93-11e4-88f5-28cfe9158b63;
Date: Mon, 25 Aug 2014 20:35:10 GMT
Server: Apache/2.4.9 (Debian)
X-Powered-By: PHP/5.5.12-2
X-Pingback: http://192.168.50.5:8080/xmlrpc.php
Vary: Accept-Encoding
Content-Length: 7467
Content-Type: text/html; charset=UTF-8

<!DOCTYPE html> ... </html>
```

GRADUAL Upgrade

In the GRADUAL upgrade each client will be allocated a transition cookie. The transition cookie will be used to randomly create each client a number between 0 and 100. The random number will be used to define which client should be upgraded to the new version. If the randomly generated number is less than or equal to a defined threshold then that client will be moved to the latest cluster version (Section 5.2.2). This section summarizes the performance evaluation of the proxy when a GRADUAL upgrade of WordPress from version 3.9.1 to 3.9.2 was applied.

Proxy Initial Configuration - The initial configuration of the proxy is shown in the following terminal output echoed by proxy. Note that the proxy was initially configured with version 3.9.1 of WordPress application.

```
2014/08/28 11:05:30.745828 config_parse.go:53:   NOTICE - Parsed config file:
{
   "proxy": {
      "port": 1235
   },
   "configService": {
      "port": 9090
   },
   "dockerHost": {
      "ip": "192.168.50.5",
      "port": 2375
   },
   "cluster": {
      "containers": [
         {
            "image": "mysql",
            "tag": "latest",
            "name": "some-mysql",
            "environment": [
               "MYSQL_ROOT_PASSWORD=mysecretpassword"
            ],
            "volumes": [
               "/var/lib/mysql:/var/lib/mysql"
            ]
         },
         {
            "image": "wordpress",
```

```
            "tag": "3.9.1",
            "portToProxy": 8080,
            "name": "some-wordpress",
            "links": [
               "some-mysql:mysql"
            ],
            "portBindings": {
               "80/tcp": [
                  {
                     "HostIp": "0.0.0.0",
                     "HostPort": "8080"
                  }
               ]
            }
         }
      ],
      "version": "3.9.1"
   }
}
as:
Proxy{
        Proxy Address:      0.0.0.0:1235
        ConfigService Port: 9090
        Proxied Servers:    version: 3.9.1 [192.168.50.5:8080]
}
```

HTTP Requests & Responses - HTTP requests were sent to WordPress application via proxy using the following cURL syntax:

```
curl -v http://127.0.0.1:1235
```

The following proxy output confirms that the HTTP requests were forwarded to the WordPress application version 3.9.1. Since no upgrade mode was chosen for the cluster, the default mode *i.e.* INSTANT mode was chosen by proxy.

```
2014/08/28 11:06:01.661061 proxy_context_cluster.go:115:
   INFO - Serving response 0 from [192.168.50.5] port: [8080]
   version: [3.9.1] mode: [INSTANT]
```

"X-Pingback" header of the response received from the proxy further confirmed that proxy is receiving responses from WordPress application version 3.9.1.

```
HTTP/1.1 200 OK
Set-Cookie: dynsoftup=d7bf97fc-2e9a-11e4-8d7c-600308a8245e;
Date: Thu, 28 Aug 2014 10:06:10 GMT
Server: Apache/2.4.9 (Debian)
X-Powered-By: PHP/5.5.12-2
X-Pingback: http://192.168.50.5:8080/xmlrpc.php
Vary: Accept-Encoding
Content-Length: 7523
Content-Type: text/html; charset=UTF-8

<!DOCTYPE html> ... </html>
```

PUT Request for GRADUAL Upgrade - The following PUT request was sent to the proxy's configuration service for upgrading WordPress application from version 3.9.1 to version 3.9.2.

```
curl -v \
http://127.0.0.1:9090/configuration/cluster -X PUT \
-H 'Content-Type: application/json' \
-d @config_for_curl/config_curl_docker_wordpress_GRADUAL.json
```

The following output echoed by the proxy showed successful addition of the new cluster with GRADUAL upgrade transition mode running WordPress version 3.9.2.

```
2014/08/28 11:11:57.781817 config_service.go:66:
   INFO - Received new cluster configuration:
   {
      "cluster": {
         "containers": [
            {
               "environment": [
                  "MYSQL_ROOT_PASSWORD=mysecretpassword"
               ],
               "image": "mysql",
               "name": "some-mysql-upgrade",
               "tag": "latest",
               "volumes": [
                  "/var/lib/mysql:/var/lib/mysql"
               ]
            },
            {
               "image": "wordpress",
               "links": [
                  "some-mysql:mysql"
               ],
               "name": "some-wordpress-upgrade",
               "portBindings": {
                  "80/tcp": [
                     {
                        "HostIp": "0.0.0.0",
                        "HostPort": "8081"
                     }
                  ]
               },
               "portToProxy": 8081,
               "tag": "3.9.2"
            }
         ],
          "version": "3.9.2",
          "upgradeTransition": {
              "mode": "GRADUAL",
              "percentageTransitionPerRequest": 1
          }
      }
   }
```

A cluster id associated to the new cluster was returned by proxy.

```
PUT /configuration/cluster HTTP/1.1
User-Agent: curl/7.30.0
Host: 127.0.0.1:9090
```

```
Accept: */*
Content-Length: 167
Content-Type: application/x-www-form-urlencoded

HTTP/1.1 202 Accepted
Date: Thu, 28 Aug 2014 10:11:57 GMT
Content-Length: 36
Content-Type: text/plain; charset=utf-8

be709f8f-2e9b-11e4-8d7c-600308a8245e
```

Evaluation of the GRADUAL Upgrade - An HTTP request with transition cookie was sent to the proxy after the new cluster was added.

```
curl -v 'http://127.0.0.1:1235/' -H 'Cookie: transition=952c8557-2088-11e4-87
   e3-600308a8245e;'
```

The proxy used the transition cookie to randomly generate a number which determines when the client will be upgraded to the new version. When multiple HTTP requests were sent to the proxy using the same transition cookie, It was noticed that the upgrade occurred after three requests as it is shown in the following terminal output:

```
2014/08/28 11:15:21.019897 proxy_context_cluster.go:115:
   INFO - Serving response 0 from [192.168.50.5] port: [8080]
   version: [3.9.1] mode: [INSTANT]

2014/08/28 11:15:22.190838 proxy_context_cluster.go:115:
   INFO - Serving response 1 from [192.168.50.5] port: [8080]
   version: [3.9.1] mode: [INSTANT]

2014/08/28 11:15:22.988277 proxy_context_cluster.go:115:
   INFO - Serving response 2 from [192.168.50.5] port: [8080]
   version: [3.9.1] mode: [INSTANT]

2014/08/28 11:15:23.708418 proxy_context_cluster.go:115:
   INFO - Serving response 0 from ip: ip: [192.168.50.5] port: [8081]
   version: [3.9.2] mode: [GRADUAL] transition counter [4.00]
   percentage transition per request [1.00]

2014/08/28 11:15:24.563001 proxy_context_cluster.go:115:
   INFO - Serving response 1 from ip: ip: [192.168.50.5] port: [8081]
   version: [3.9.2] mode: [GRADUAL] transition counter [5.00]
   percentage transition per request [1.00]
```

Handling Incorrect Behavior - To insure the proxy forwards requests to the older version of the application if the new version behaved incorrectly, the Docker container running version 3.9.2 of the WordPress application was stopped. The behavior of the proxy was screened when new HTTP requests with transition cookie were sent to the proxy. The following terminal output shows that when the proxy failed connecting to the new version of the application, it removed the cluster from the configuration and forwarded requests to the previous cluster version.

```
2014/08/28 11:20:48.105660 proxy_context_cluster.go:115:
   INFO - Serving response 2 from ip: ip: [192.168.50.5] port: [8081]
   version: [3.9.2] mode: [GRADUAL] transition counter [8.00]
   percentage transition per request [1.00]
```

```
2014/08/28 11:20:48.106281 stage_route.go:28:
   ERROR - Error communicating with server - dial tcp 192.168.50.5:8081:  connection refused

2014/08/28 11:20:48.106299 stage_route.go:29:
   WARNING - Removing cluster from configuration - version:  3.9.2 [192.168.50.5:8081]

2014/08/28 11:20:48.106319 proxy_context_cluster.go:115:
   INFO - Serving response 3 from [192.168.50.5] port: [8080]
   version: [3.9.1] mode: [INSTANT]
```

The following response received after the failure of the cluster running version 3.9.2 showed that version 3.9.1 with address "192.168.50.5:8080" is processing the requests.

```
HTTP/1.1 200 OK
Set-Cookie: dynsoftup=ffa87add-2e9b-11e4-ae02-600308a8245e;
Date: Thu, 28 Aug 2014 10:20:56 GMT
Server: Apache/2.4.9 (Debian)
X-Powered-By: PHP/5.5.12-2
X-Pingback: http://192.168.50.5:8080/xmlrpc.php
Vary: Accept-Encoding
Content-Length: 7523
Content-Type: text/html; charset=UTF-8

<!DOCTYPE html> ... </html>
```

CONCURRENT Upgrade

In the CONCURRENT mode, the proxy sends HTTP requests to the old and new versions of an application and it monitors the responses arriving from the highest version of the application. If the latest version correctly accepts a new TCP connection and responds with an HTTP status code that is not in the 5xx range, for server errors, then the proxy forwards its response to the client. However, If the latest cluster does not accept a new TCP connection or responds with an HTTP status code in the 5xx range, the proxy drops its response and forwards the response from the older version of the application to the client. To evaluate that the CONCURRENT mode of the proxy works reliably, WordPress version 3.9.1 was upgraded to version 3.9.2. The upgrade process, it is summarized in this section. Proxy log output and the HTTP request and responses are used to demonstrate how the proxy behaves for the concurrent upgrade.

Proxy Initial Configuration - The following terminal output shows the initial configuration of the proxy. Proxy is configured with version 3.9.1 of WordPress application.

```
2014/08/29 12:39:55.615246 config_parse.go:56:   NOTICE - Parsed config file:
{
   "proxy": {
      "port": 1235
   },
   "configService": {
      "port": 9090
   },
   "dockerHost": {
      "ip": "192.168.50.5",
      "port": 2375
   },
   "cluster": {
      "containers": [
         {
            "image": "mysql",
```

```
            "tag": "latest",
            "name": "some-mysql",
            "environment": [
               "MYSQL_ROOT_PASSWORD=mysecretpassword"
            ],
            "volumes": [
               "/var/lib/mysql:/var/lib/mysql"
            ]
         },
         {
            "image": "wordpress",
            "tag": "3.9.1",
            "portToProxy": 8080,
            "name": "some-wordpress",
            "links": [
               "some-mysql:mysql"
            ],
            "portBindings": {
               "80/tcp": [
                  {
                     "HostIp": "0.0.0.0",
                     "HostPort": "8080"
                  }
               ]
            }
         }
      ],
      "version": "3.9.1"
   }
}
as:
Proxy{
    Proxy Address:      0.0.0.0:1235
    ConfigService Port: 9090
    Proxied Servers:    version: 3.9.1 [192.168.50.5:8080]
}
```

HTTP Request & Responses - HTTP requests were sent to the WordPress application via proxy using cURL.

```
curl -v http://127.0.0.1:1235
```

As it is shown in the following terminal output echoed by proxy, WordPress version 3.9.1 was processing the requests. Upgrade transition mode INSTANT was given to the initial configuration since no upgrade mode was specified.

```
2014/08/29 12:40:00.005587 proxy_context_cluster.go:115:
   INFO - Serving response 0 from [192.168.50.5] port: [8080]
   version: [3.9.1] mode: [INSTANT]
```

The "X-Pingback" header in the response received from proxy also confirmed that version 3.9.1 of the WordPress application with address "192.168.50.5:8080" is processing the requests.

```
HTTP/1.1 200 OK
Set-Cookie: dynsoftup=32afd70b-2f71-11e4-b59e-600308a8245e;
Date: Fri, 29 Aug 2014 11:40:00 GMT
Server: Apache/2.4.9 (Debian)
X-Powered-By: PHP/5.5.12-2
```

```
X-Pingback: http://192.168.50.5:8080/xmlrpc.php
Vary: Accept-Encoding
Content-Length: 7523
Content-Type: text/html; charset=UTF-8

<!DOCTYPE html> ... </html>
```

PUT Request for CONCURRENT Upgrade - To upgrade WordPress from version 3.9.1 to version 3.9.2, the following PUT request was sent to the proxy's configuration service:

```
curl -v \
http://127.0.0.1:9090/configuration/cluster -X PUT \
-H 'Content-Type: application/json' \
-d @config_for_curl/config_curl_docker_wordpress_CONCURRENT.json
```

The following terminal output showed successful addition of the cluster running version 3.9.2 of WordPress application in CONCURRENT upgrade transition mode:

```
2014/08/29 12:42:55.844840 config_service.go:66:
   INFO - Received new cluster configuration:
   {
      "cluster": {
         "containers": [
            {
               "environment": [
                  "MYSQL_ROOT_PASSWORD=mysecretpassword"
               ],
               "image": "mysql",
               "name": "some-mysql-upgrade",
               "tag": "latest",
               "volumes": [
                  "/var/lib/mysql:/var/lib/mysql"
               ]
            },
            {
               "image": "wordpress",
               "links": [
                  "some-mysql:mysql"
               ],
               "name": "some-wordpress-upgrade",
               "portBindings": {
                  "80/tcp": [
                     {
                        "HostIp": "0.0.0.0",
                        "HostPort": "8081"
                     }
                  ]
               },
               "portToProxy": 8081,
               "tag": "3.9.2"
            }
         ],
        "version": "3.9.2",
        "upgradeTransition": {
            "mode": "CONCURRENT"
         }
      }
   }
```

The response body received from the proxy contained a cluster id associated to the new cluster.

```
HTTP/1.1 202 Accepted
Date: Fri, 29 Aug 2014 11:42:55 GMT
Content-Length: 36
Content-Type: text/plain; charset=utf-8

9e1cb1e4-2f71-11e4-b59e-600308a8245e
```

Evaluation of the CONCURRENT Upgrade - To evaluate the behavior of the proxy after the CONCURRENT upgrade, HTTP requests were sent to the proxy.

```
curl -v http://127.0.0.1:1235
```

The proxy's log output showed that the proxy sent requests concurrently to both versions of the WordPress application (*i.e.* version 3.9.1 and 3.9.2).

```
2014/08/29 12:45:00.892252 proxy_context_cluster.go:115:
   INFO - Serving response 0 from ip: ip: [192.168.50.5] port: [8081]
   version: [3.9.2] mode: [CONCURRENT]

2014/08/29 12:45:00.892822 proxy_context_cluster.go:115:
   INFO - Serving response 1 from [192.168.50.5] port: [8080]
   version: [3.9.1] mode: [INSTANT]
```

However, the client only received a response from the most updated version as shown below. Note that the "X-Pingback" header contains the IP address of the WordPress application version 3.9.2.

```
HTTP/1.1 200 OK
Set-Cookie: dynsoftup=9e1cb1e4-2f71-11e4-b59e-600308a8245e;
Date: Fri, 29 Aug 2014 11:45:00 GMT
Server: Apache/2.2.22 (Debian)
X-Powered-By: PHP/5.4.4-14+deb7u12
X-Pingback: http://192.168.50.5:8081/xmlrpc.php
Vary: Accept-Encoding
Content-Length: 7523
Content-Type: text/html; charset=UTF-8

<!DOCTYPE html> ... </html>
```

Handling Incorrect Behavior - To insure that the proxy drops responses arriving from the newest version of the application when it is not behaving correctly the following evaluation was performed. Docker container running version 3.9.2 of the WordPress application was stopped and HTTP requests were consequently sent to the proxy using cURL.

```
curl -v http://127.0.0.1:1235
```

The log outputted by proxy showed that proxy concurrently forwards messages to both versions of the WorldPress application. However, since the latest version is not available, the proxy ignores the cluster running the latest version and forwards the respond received from version 3.9.1 to the client as shown below. Note the IP address demonstrated in the "X-Pingback" header of the response refers to version 3.9.1 of the WordPress application.

Proxy Log Output

```
2014/08/29 12:47:30.434366 proxy_context_cluster.go:115:
   INFO - Serving response 1 from ip: ip: [192.168.50.5] port: [8081]
   version: [3.9.2] mode: [CONCURRENT]

2014/08/29 12:47:30.434794 proxy_context_cluster.go:115:
   INFO - Serving response 2 from [192.168.50.5] port: [8080]
   version: [3.9.1] mode: [INSTANT]
```

Response

```
HTTP/1.1 200 OK
Set-Cookie: dynsoftup=9e1cb1e4-2f71-11e4-b59e-600308a8245e;
Date: Fri, 29 Aug 2014 11:47:30 GMT
Server: Apache/2.4.9 (Debian)
X-Powered-By: PHP/5.5.12-2
X-Pingback: http://192.168.50.5:8080/xmlrpc.php
Vary: Accept-Encoding
Content-Length: 7523
Content-Type: text/html; charset=UTF-8

<!DOCTYPE html> ... </html>
```

6.2.3 xxxxx Handling Incorrect Behavior

To ensure that the system is able to handle updates that cause incorrect behavior multiple scenarios will be tested. This test will cover situations where a single node, multiple nodes or all nodes in the updated application cluster behave incorrectly. Incorrect behavior will be modeled by either the application crashing or by the application taking too long to respond.

Lighttpd1 is a popular open-source web-server used by several high-traffic websites such as Wikipedia and YouTube. Despite its popularity, crash bugs are still a common occurrence in Lighttpd, as evident from its bug tracking database.2 Below we discuss one such bug, which our approach could successfully eliminate. In April 2009, a patch was applied3 to LighttpdâĂŹs code related to the HTTP ETag functionality. An ETag is a unique string assigned by a web server to a specific version of a web resource, which can be used to quickly determine if the resource has changed. The patch was a one-line change, which discarded the terminating zero when computing a hash representing the ETag. More exactly, line 47 in etag.c: for (h=0, i=0; i < etag->used; ++i) h = (h«5 ËĘ(h»27)ËĘ(eta->ptr[i]); was changed to: for (h=0, i=0; i < etag->used-1; ++i) h = (h«5 ËĘ(h»27)ËĘ(eta->ptr[i]); This correctly changed the

This correctly changed the way ETags are computed, but unfortunately, it broke the support for compression, whose implementation depended on the previous computation. More precisely, LighttpdâĂŹs support for HTTP compression uses caching to avoid re-compressing files which have not changed since the last access. To determine whether the cached file is still valid, Lighttpd internally uses ETags. Unfortunately, the code implementing HTTP compression did not consider the case when ETags are disabled. In this case, etags->used is 0, and when the line above is executed, etag->used-1 underflows to a very large value, and the code crashes while accessing etag->ptr[i]. Interestingly enough, the original code was still buggy (it always returns zero as the hash value, and thus it would never re-compress the files), but it was not vulnerable to a crash. The segfault was diagnosed and reported in March 20104 and fixed at the end of April 2010,5 more than one year

after it was introduced. The bottom line is that for about one year, users affected by this buggy patch essentially had to decide between (1) incorporating the new features and bug fixes added to the code, but being vulnerable to this crash bug, and (2) giving up on these new features and bug fixes and using an old version of Lighttpd, which is not vulnerable to this bug. Our approach provides users with a third choice; when a new version arrives, instead of replacing the old version, we run both versions in parallel. In our example, consider that we are using MX to run a version of Lighttpd from March 2009. When the buggy April 2010 version is released, MX runs it in parallel with the old one. As the two versions execute: âĂć As long as the two versions have the same external behaviour (e.g. they write the same values into the same files, or send the same data over the network), they are run side-by-side and MX ensures t

Chapter 7

Conclusions & Future Plans

xxxxxxxxx

xxxxxxxxx

xxxxxxxxx

Chapter 8

Appendices

Appendix I: Proxy REST API

The proxy provides a simple REST API to support dynamically updating the cluster configuration as follows:

- **PUT /configuration/cluster** - adds a new cluster configuration
- **GET /configuration/cluster/clusterId** - gets a single cluster configuration
- **GET /configuration/cluster** - gets all cluster configurations
- **DELETE /configuration/cluster/clusterId** - deletes a single cluster configuration

HTTP Response Codes

- 202 Accepted - a new cluster entity is successfully added or deleted
- 200 OK - cluster(s) entity is successfully returned
- 404 Not Found - cluster id is invalid
- 400 Bad Request - request syntax is invalid

PUT - /configuration/cluster

To add a new cluster make a PUT request to */configuration/cluster*. There are two supported request body formats depending on whether the cluster is using docker containers or pre-existing servers (i.e. IP and port combinations), as follows:

Request Body - Docker Container Configuration

```
{
    "cluster": {
        "containers": [
            {
                "image": "",
                "tag": "",
                "alwaysPull": false,
                "portToProxy": 0,
                "name": "",
                "workingDir": "",
                "entrypoint": [
                    ""
                ],
                "environment": [
```

```
                    ""
                ],
                "cmd": [
                    ""
                ],
                "hostname": "",
                "volumes": [
                    ""
                ],
                "volumesFrom": [
                    ""
                ],
                "portBindings": {
                    "": [
                        {
                            "hostIp": "",
                            "hostPort": ""
                        }
                    ]
                },
                "links": [
                    ""
                ],
                "user": "",
                "memory": 0,
                "cpuShares": 0,
                "lxcConf": [
                    {
                        "key": "",
                        "value": ""
                    }
                ],
                "privileged": false,
            }
        ],
        "version": ""
    }
}
```

Request Body - Existing Server Configuration

```
{
  "cluster": {
    "servers": [
      {
        "hostname": "",
        "port": 0
      }
    ],
    "version": "",
    "upgradeTransition": {
        "mode": ""  // allowed values are "INSTANT", "SESSION", "GRADUAL", "CONCURRENT"
        "sessionTimeout": 0  // only supported for a 'mode' value of "SESSION"
        "percentageTransitionPerRequest": 0   // only supported for "GRADUAL" mode
      }
    }
}
```

cluster.containers

Type: 'Array' Default value: '[]'

specifies the list of docker containers in the cluster.

cluster.containers[i].image

Type: 'String' Default value: 'undefined'

name of the image to create each container from.

cluster.containers[i].tag

Type: 'String' Default value: 'latest'

the tag for the image, defaulting to 'latest'.

cluster.containers[i].alwaysPull

Type: 'Boolean' Default value: 'false'

whether to automatically check for new image versions.

cluster.containers[i].portToProxy

Type: 'Number' Default value: '0'

the port the proxy routes HTTP request to.

cluster.containers[i].name

Type: 'String' Default value: 'undefined'

the name of the container, defaulting to an auto-generated unique name.

cluster.containers[i].workingDir

Type: 'String' Default value: 'undefined'

the current working directory inside the container's root file system.

cluster.containers[i].entrypoint

Type: 'Array' Default value: '[]'

the container's entrypoint.

cluster.containers[i].environment

Type: 'Array' Default value: '[]'

the environment variables that are set in the running container.

cluster.containers[i].cmd

Type: 'Array' Default value: '[]'

the command to be executed when running the container.

cluster.containers[i].hostname

Type: 'String' Default value: 'undefined'

the container's hostname.

cluster.containers[i].volumes

Type: 'Array' Default value: '[]'

mount volumes from either the host or from another docker container.

cluster.containers[i].volumesFrom

Type: 'Array' Default value: '[]'

mount all the volumes defined for one or more other docker containers, including control over whether the volumes are mounted in read-write or read-only mode.

cluster.containers[i].portBindings

Type: 'Array' Default value: '[]'

bind ports from within the container to one or more host port and IP combination, using format <port>/tcp or <port>/udp as the key.

cluster.containers[i].portBindings[i].hostIp

Type: 'String' Default value: 'undefined'

the host ip address to bind the port to.

cluster.containers[i].portBindings[i].hostPort

Type: 'String' Default value: 'undefined'

the host port to bind the port to.

cluster.containers[i].links

Type: 'Array' Default value: '[]'

link the container to another Docker container.

cluster.containers[i].user

Type: 'String' Default value: 'undefined'

set the user user id and group id of the executing process running inside the container.

cluster.containers[i].memory

Type: 'Number' Default value: '0'

restrict the container's memory (in bytes).

cluster.containers[i].cpuShares

Type: 'Number' Default value: '0'

restrict the container's cpu share using relative weight compared to other containers.

cluster.containers[i].lxcConf

Type: 'Array' Default value: '[]'

add custom LXC options for the container.

cluster.containers[i].lxcConf.key

Type: 'String' Default value: 'undefined'

key (or name) for the custom LXC options for the container.**cluster.containers[i].lxcConf.value**

Type: 'String' Default value: 'undefined'

value for the custom LXC options for the container.

cluster.containers[i].privileged

Type: 'Boolean' Default value: 'false'

give extended privileges to the container.

cluster.servers

Type: 'Array' Default value: '[]'

This value specifies the list of servers in the cluster

cluster.servers[i].ip

Type: 'String' Default value: 'undefined'

This value specifies the ip address or hostname of a server in the cluster

cluster.servers[i].port

Type: 'Number' Default value: 'undefined'

This value specifies the port of a server in the cluster

cluster.version

Type: 'String' Default value: '0.0'

This value specifies the cluster version. If no version is specified, the version defaults to '0.0'. The version value is sorted using a string sort so care must be taken when using multi digit version numbers as '13' will be sorted before '3' to resolve this always use '03' for '3'.

cluster.upgradeTransition

Type: 'Object' Default value: "INSTANT'

This value allows the configuration of the upgrade transition. If no 'upgradeTransition' is specified, the upgrade transition mode defaults to 'INSTANT'.

cluster.upgradeTransition.mode

Type: 'String' Default value: 'SESSION'

This value specifies the upgrade transition mode and support the following values: 'INSTANT', 'SESSION', 'GRADUAL', 'CONCURRENT'.

cluster.upgradeTransition.sessionTimeout

Type: 'Number' Default value: 'undefined'

This value specifies the timeout period assigned to the 'SESSION' transition mode.

cluster.upgradeTransition.percentageTransitionPerRequest

Type: 'Number' Default value: 'undefined'

This value specifies the transition percentage associated with each request in the 'GRADUAL' transition mode.

Response Body

A cluster id is returned representing the new cluster entity that has been added.

Example

Request

For example the following JSON would set up a new cluster with two 'servers' and 'SESSION' upgrade transition:

```
{
  "cluster": {
    "servers": [
      {
        "hostname": "127.0.0.1",
        "port": 1036
      },
      {
        "hostname": "127.0.0.1",
        "port": 1038
      }
    ],
    "version": "1.1",
    "upgradeTransition": {
      "mode": "SESSION",
      "sessionTimeout": 60
    }
  }
}
```

To send this request with 'cURL' use the following syntax:

```
curl \
http://127.0.0.1:9090/configuration/cluster -X PUT \
--data '{"cluster": {"servers":[{"hostname": "127.0.0.1", "port": 1036},{"
    hostname": "127.0.0.1", "port": 1038}],"version": "1.1","upgradeTransition
    ": { "mode": "SESSION", "sessionTimeout": 60 }}}'
```

Response

```
HTTP/1.1 202 Accepted
Date: Sat, 16 Aug 2014 19:54:21 GMT
```

```
Content-Length: 36
Content-Type: text/plain; charset=utf-8

1dcbb083-257f-11e4-bcbc-600308a8245e
```

GET - /configuration/cluster/clusterId

To get a single cluster configuration make a GET request to */configuration/cluster/clusterId*.

Response Body

```
{
  "cluster": {
    "servers": [
      {
        "hostname": "",
        "port": 0
      }
    ],
    "upgradeTransition": {
      "mode": ""
      "sessionTimeout": 0  // only returned when 'mode' is "SESSION"
      "percentageTransitionPerRequest": 0  // only returned when 'mode' is "GRADUAL"
    },
    "uuid": "",
    "version": ""
  }
}
```

Example

Request

For example the following 'curl' request would get the cluster configuration with cluster id '1dcbb083-257f-11e4-bcbc-600308a8245e':

```
curl \
http://127.0.0.1:9090/configuration/cluster/\
1dcbb083-257f-11e4-bcbc-600308a8245e -X GET
```

Response

```
{
  "cluster": {
    "servers": [
      {
        "hostname": "127.0.0.1",
        "port": 1036
      },
      {
        "hostname": "127.0.0.1",
        "port": 1038
      }
     ],
    "upgradeTransition": {
```

```
        "mode": "SESSION",
        "sessionTimeout": 60
    },
    "uuid": "016ca2cd-2585-11e4-ab5c-600308a8245e",
    "version": "1.1"
  }
}
```

For example the response when using curl is as follows:

```
HTTP/1.1 200 OK
Date: Sat, 16 Aug 2014 20:37:42 GMT
Content-Length: 206
Content-Type: text/plain; charset=utf-8

{
    "cluster": {
        "servers": [
            {
                "hostname": "127.0.0.1",
                "port": 1036
            },
            {
                "hostname": "127.0.0.1",
                "port": 1038
            }
        ],
        "upgradeTransition": {
            "mode": "SESSION",
            "sessionTimeout": 60
        },
        "uuid": "016ca2cd-2585-11e4-ab5c-600308a8245e",
        "version": "1.1"
    }
}
```

GET - /configuration/cluster

To get all the cluster configurations make a GET request with no cluster id */configuration/cluster/*.

Response Body

```
[
  {
    "cluster": {
      "servers": [
        {
          "hostname": "",
          "port": 0
        }
       ],
      "upgradeTransition": {
        "mode": ""
        "sessionTimeout": 0  // only returned when 'mode' is "SESSION"
        "percentageTransitionPerRequest": 0  // only returned when 'mode' is "
           GRADUAL"
      },
      "uuid": "",
```

```
      "version": ""
    }
  },
  {
    "cluster": {
      "servers": [
        {
          "hostname": "",
          "port": 0
        },
        {
          "hostname": "",
          "port": 0
        }
       ],
      "upgradeTransition": {
        "mode": "CONCURRENT"
      },
      "uuid": "",
      "version": ""
    }
  }
]
```

Example

Request

For example the following 'curl' request would get a list of all cluster configurations

```
curl http://127.0.0.1:9090/configuration/cluster/ -X GET
'''
##### Response

\begin{lstlisting}[language=json]
[
  {
    "cluster": {
      "servers": [
        {
          "hostname": "127.0.0.1",
          "port": 1036
        },
        {
          "hostname": "127.0.0.1",
          "port": 1038
        }
      ],
      "upgradeTransition": {
        "mode": "SESSION",
        "sessionTimeout": 60
      },
      "uuid": "1f6a0854-2608-11e4-ab79-600308a8245e",
      "version": "1.1"
    }
  },
  {
    "cluster": {
```

```
      "servers": [
        {
          "hostname": "127.0.0.1",
          "port": 1037
        },
        {
          "hostname": "127.0.0.1",
          "port": 1039
        }
      ],
      "upgradeTransition": {
        "mode": "CONCURRENT"
      },
      "uuid": "01386f1f-2608-11e4-ab79-600308a8245e",
      "version": "1.1"
    }
  },
  {
    "cluster": {
      "servers": [
        {
          "hostname": "127.0.0.1",
          "port": 1034
        },
        {
          "hostname": "127.0.0.1",
          "port": 1035
        }
      ],
      "upgradeTransition": {
        "mode": "INSTANT"
      },
      "uuid": "ffde36ce-2607-11e4-ab79-600308a8245e",
      "version": "1"
    }
  }
]
```


For example the response when using curl is as follows:


```
HTTP/1.1 200 OK
Date: Sun, 17 Aug 2014 12:28:55 GMT
Content-Length: 583
Content-Type: text/plain; charset=utf-8

[
    {
        "cluster": {
            "servers": [
                {
                    "hostname": "127.0.0.1",
                    "port": 1036
                },
                {
                    "hostname": "127.0.0.1",
                    "port": 1038
                }
            ],
            "upgradeTransition": {
```

```
                "mode": "SESSION",
                "sessionTimeout": 60
            },
            "uuid": "1f6a0854-2608-11e4-ab79-600308a8245e",
            "version": "1.1"
        }
    },
    {
        "cluster": {
            "servers": [
                {
                    "hostname": "127.0.0.1",
                    "port": 1037
                },
                {
                    "hostname": "127.0.0.1",
                    "port": 1039
                }
            ],
            "upgradeTransition": {
                "mode": "CONCURRENT"
            },
            "uuid": "01386f1f-2608-11e4-ab79-600308a8245e",
            "version": "1.1"
        }
    },
    {
        "cluster": {
            "servers": [
                {
                    "hostname": "127.0.0.1",
                    "port": 1034
                },
                {
                    "hostname": "127.0.0.1",
                    "port": 1035
                }
            ],
            "upgradeTransition": {
                "mode": "INSTANT"
            },
            "uuid": "ffde36ce-2607-11e4-ab79-600308a8245e",
            "version": "1"
        }
    }
]
```


DELETE - /configuration/cluster/clusterId

To delete a single cluster configuration make a DELETE request to */configuration/cluster/clusterId*.

Example

Request

For example the following cURL request would delete the cluster configuration with id '1dcbb083-257f-11e4-bcbc-600308a8245e':

```
curl \
http://127.0.0.1:9090/configuration/cluster/\
1dcbb083-257f-11e4-bcbc-600308a8245e -X DELETE
```

Response

For example the response when using curl is as follows:

```
HTTP/1.1 202 Accepted
Date: Sat, 16 Aug 2014 21:28:38 GMT
Content-Length: 0
Content-Type: text/plain; charset=utf-8
```

Chapter 9

Bibliography

[1] Orso A, Rao A, Harrold M. J , *A Technique for Dynamic Updating of Java Software*, CSM Proceedings of the International Conference on Software, 2002.

[2] Segal M. E, Frieder O, *On-the-fly program modification: Systems for dynamic updating.* IEEE Software, 1993.

[3] Thakur A, *Analysis of failures in the Tandem NonStop-UX Operating System.*, Proceedings., Sixth International Symposium on Software Reliability Engineering, 1995.

[4] Ohba M, *Software reliability analysis models.*, IBM Journal of Research and Development, 1984.

[5] Speirs, N.A, Barrett, P. A, *Using passive replicates in Delta-4 to provide dependable distributed computing.*, Fault-Tolerant Computing, 1989. Speirs, N.A, Barrett, P. A

[6] Acharya S , Zdonik S. B, *An Efficient Scheme for Dynamic Data Replication.*, Tech Report, 1993.

[7] Pokorny J, *NoSQL databases: a step to database scalability in web environment.*, International Journal of Web Information Systems, 2013.

[8] Sahai A, Calton P, Jung G, Wu Q, Yan W, Swint G. S, *Towards Automated Deployment of Built-to-Order Systems.*, Ambient Networks, 2005

[9] Humble J, Farley D, *Continuous Delivery: Reliable Software Releases Through Build, Test, and Deployment Automation.*, Addison-Wesley Professional Publisher, 2010.

[10] Apostolopoulos G, Aubespin D, Peris V, Pradhan P, Saha D, *Design, implementation and performance of a content-based switch.*, Nineteenth Annual Joint Conference of the IEEE Computer and Communications Societies, 2000.

[11] Arnold J, Kaashoek M. F, *Ksplice: automatic rebootless kernel updates.*, EuroSys, 2009.

[12] Altekar G, Bagrak I, Burstein P, Schultz A, *OPUS: Online patches and updates for security.*, USENIX Security, 2005.

[13] Makris K Ryu K. D, *Dynamic and Adaptive Updates of Non-Quiescent Subsystems in Commodity Operating System Kernels.*, EuroSys, 2007.

[14] Chen H, Yu J, Hang C, Zang B, Yew P. C, *Dynamic software updating using a relaxed consistency model.*, IEEE Transactions on Software Engineering, 2011.

[15] Neamtiu I, Hicks M, Stoyle G, Oriol M, *Practical dynamic software updating for C.*, PLDI, 2006.

[16] Baumann A, Appavoo J, Silva D. D, Kerr J, Krieger O, Wisniewski R. W, *Providing dynamic update in an operating system.* USENIX ATC, 2005.

[17] Makris K, Bazzi R, *Immediate Multi-Threaded Dynamic Software Updates Using Stack Reconstruction.*, USENIX ATC, 2009.

[18] Hayden C. M, Smith E. K, Hicks M ,Foster J. S, *State transfer for clear and efficient runtime upgrades.*, HotSWUp, 2011.

[19] Hayden C. M, Smith E. K, Denchev M, Hicks M, Foster J. K, *"Kitsune: Efficient, general-purpose dynamic software updating for C"*, OOPSLA, 2012.

[20] Altekar G, Bagrak L, Burstein P, and Schultz A, *OPUS: Online patches and updates for security.*, USENIX Security Symp, 2005.

[21] Baumann A, Appavoo J, Wisniewski R. W, Silva D. D, Krieger O, Heiser G, *Reboots are for hardware: Chal- lenges and solutions to updating an operating system on the fly.*, USENIX Annual Tech. Conf., 2007.

[22] Neamtiu L, Hicks M, Stoyle G, Oriol M, *Practical dynamic software updating for C.*, ACM SIGPLAN Conf. on Programming Language Design and Implementation, 2006.

[23] Neamtiu L, Hicks M, *Safe and timely updates to multi-threaded programs.* ACM SIGPLAN Conf. on Programming Language Design and Implementation, 2009.

[24] Stoyle G, Hicks M, Bierman G, Sewell P, Neamtiu L, Mutandis M, *Safe and predictable dynamic software updating.*, ACM Trans. Program. Lang. Syst., 29(4), 2007.

[25] Hayden C M, Smith E. K, Hicks M, Foster S. J, *State transfer for clear and efficient runtime updates.*, In Proc. of the Third Int'l Workshop on Hot Topics in Software Upgrades, 2011.

[26] Giuffrida C, Kuijsten A, Tanenbaum A. S, *Enhanced operating system security through efficient and fine-grained address space randomization.*, USENIX Security Symp., 2012.

[27] Neamtiu L, Hicks M, Foster J. S, Pratikakis P, *Contextual effects for version-consistent dynamic software updating and safe concurrent programming.*, ACM SIGPLAN Conf. on Programming Language Design and Implementation, 2008.

[28] Kramer J, Magee J, *The evolving philosophers problem: Dynamic change management.*, IEEE Trans. Softw. Eng., 1990.

[29] Vandewoude Y, Ebraert P, Berbers Y, D'Hondt T, *Tranquility: A low disruptive alternative to quiescence for ensuring safe dynamic updates.*, IEEE Trans. Softw. Eng., 2007.

[30] Cook J. E, Dage J. A, *"Highly reliable upgrading of components"*, ICSE Conf. on Proceedings of the 21st international conference on Software engineering, 1999.

[31] Berger E. D, Zorn B, *DieHard: probabilistic memory safety for unsafe languages.*, ACM SIGPLAN Conf. on Programming Language Design and Implementation, 2006.

[32] Kaushik Veeraraghavan, Chen M. P, Flinn J, Narayanasamy S, *detecting and surviving data races using complementary schedules*, SOSP, 2011.

[33] Hosek P, Cadar C, *Safe software updates via multi-version execution.*, Int'l Conf. on Software Eng., 2013.

[34] Cadar C, Hosek P, *Multi-version software updates.*, In Proc. of the Fourth Int'l Workshop on Hot Topics in Software Upgrades, 2012.

[35] Chen L, Avizienis A, *"N-version programming: A fault-tolerance approach to reliability of software operation"*, in FTCS, 1978.

[36] Mosharaf Kabir Chowdhury N. M, Boutaba R. b, *A survey of network virtualization*, Computer Networks, 2010.

[37] Bressoud T. C, Schneider F. B, *Hypervisor-based fault tolerance.*, ACM Transactions on Computer System, 1996.

[38] Tholeti B. P, *Hypervisors, virtualization, and the cloud: Learn about hypervisors, system virtualization, and how it works in a cloud environment*, IBM, 2011.

[39] Vaughan-Nichols S. J, *New Approach to Virtualization Is a Lightweight.*, Computer, 2006.

[40] Sahoo J, *Virtualization: A Survey on Concepts, Taxonomy and Associated Security Issues.*, Computer and Network Technology, 2010.

[41] Merkel, D, *Docker: Lightweight Linux Containers for Consistent Development and Deployment.*, Linux Journal, 2014.

[42] Vinoski S, *Advanced Message Queuing Protocol*, IEEE Internet Computing, 2006.

[43] Hapner M, Burridge R, Sharma R, Fialli J, Stout K, *Java Message Service*, In Oracle America, Inc., 2012.

[44] Raj A, Kumar S. P, *"Branch Sequencing Based XML Message Broker Architecture,"*, IEEE 23rd International Conference on Data Engineering, 2007.

[45] Vassar C, Poughkeepsie, N.Y, *A File Structure for the Complex, the Changing, and the Indeterminate*, 20th National Conference, New York, Association for Computing Machinery, 1965.

[46] Yin Z, Yuan D, Zhou Y, Pasupathy S, and Bairavasundaram L, *"How do fixes become bugs?"*, ESEC/FSE, 2011.

[47] Boehm, B. W, *Improving software productivity*, IEEE Computer, 1987a.

[48] Boehm, B. W, Victor R. B, *Software Defect Reduction Top 10 List*, Computer, v.34, 2001.