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

# Contents

# 1 Abstract

## 2 Introduction

Reliable software is important in many sectors such as E-commerce, healthcare, banking, air traffic control and nuclear control.

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. However, software upgrade itself may cause unreliability by introducing new errors, and security vulnerabilities. Many different systems and techniques have been proposed to support reliable software upgrade. Currently, these systems are either specific for particular programming languages or are complex, heavyweight and expensive. In addition, no existing system or technique is considered robust enough to be used widely for fully automated continuous delivery of upgrades into production.

This report summarizes different approaches and techniques used for the past few decades to overcome software reliability. In addition, a simple lightweight system is proposed to support reliable dynamic software update via a fully automated process suitable for continuous delivery of applications written in any programming language.

This paragraph will be updated when the report is finished

# 3 Overview

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 will describe different approaches and technologies used to achieve and maintain software reliability, specifically to support the activities mentioned above.

## 3.1 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.

## 3.2 Software Update

Once faults, such as bugs and security vulnerabilities, have been detected software must be updated to remove these faults. 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, updating or patching software is not risk free 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**

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

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

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 4.1.

## 3.3 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.

### 3.3.1 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 [2] or RabbitMQ [3] can be used to allow nodes to pull requests (Section 5.2).

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 [4], Citrix NetScaler [5]) 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.

### 3.3.2 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 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

[2] http://activemq.apache.org/

[3] http://www.rabbitmq.com/

[4] https://f5.com/

[5] http://www.citrix.com/products/netscaler-application-delivery-controller/overview.html

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.

### 3.3.3 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.

### 3.3.4 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. As 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) [6]. 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.

## 3.4 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.

### 3.4.1 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

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

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 [7], Chef [8], Ansible [9] and Salt [10].

### 3.4.2 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.

### 3.4.3 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. In addition, 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.

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

# 4 Related Work

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. This section summarizes different approaches that have been proposed over the past forty years for highly reliable dynamic software upgrade.

## 4.1 Dynamic Software Update Systems

In the past decades several systems have been developed to support dynamic software updates. 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.

### 4.1.1 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 reconstitution 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.

### 4.1.2 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 âĂIJslopâĂİ 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*.

## 4.2 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]).

### 4.3 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.

# 5 Background

This section explains and justifies the technologies that will be used in the implementation of the dynamic software update system proposed in section 7.

## 5.1 Efficient Cluster Management

As described in section 3.3 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.

### 5.1.1 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 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 [11].

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 [12] or VirtualBox [13]. 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 1 ).

**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 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) [14]. 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

[11] http://www.markus-gattol.name/ws/lvm.html
[12] http://www.vmware.com/
[13] https://www.virtualbox.org/
[14] https://www.appeagle.com/ecommerce-news/ecommerce-is-on-the-rise-in-2013/

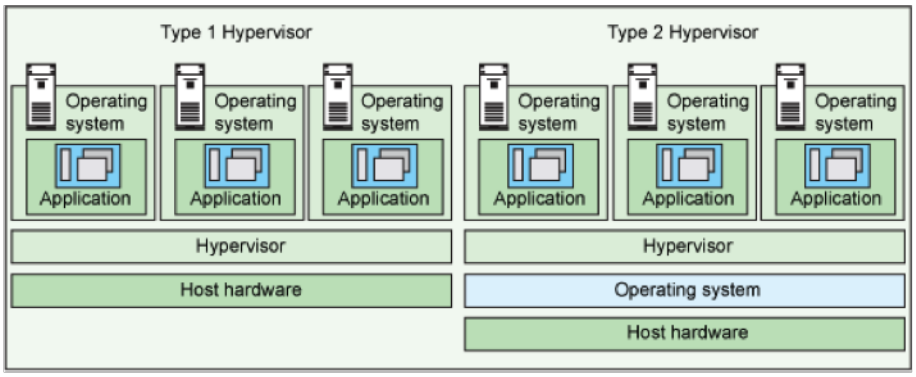


Figure 1: Hypervisor Type 1 vs. Type 2 [38]

each container as a separate host. Container virtualization allows installation of several different 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].

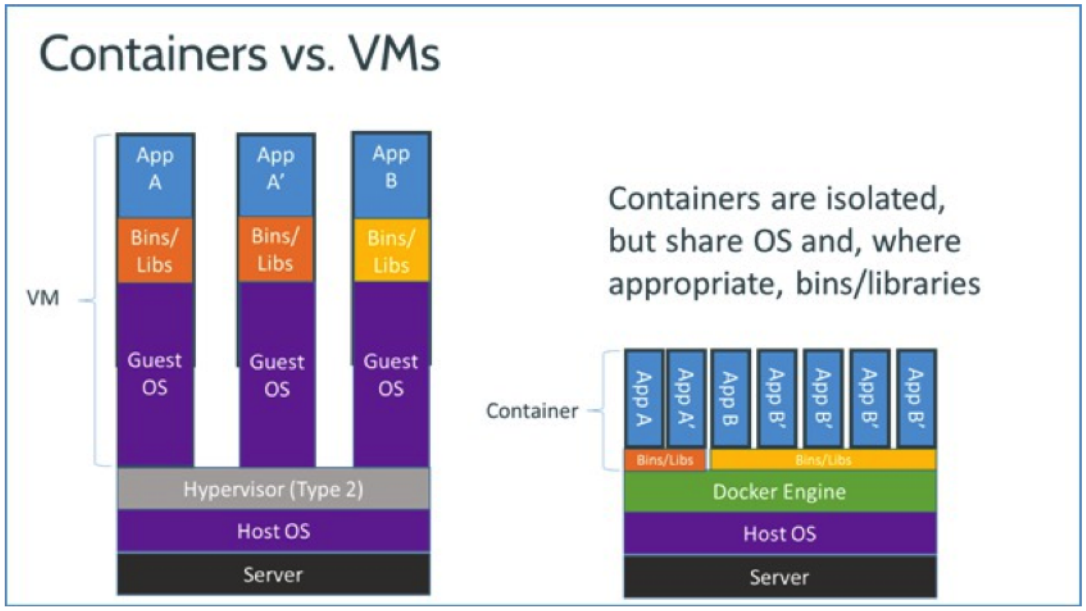


Figure 2: Containers vs. Traditional Virtual Machines [14]

***Linux Containers***

Container virtualization has been developed independently for different operating systems such as Linux OpenVZ [15], Solaris Containers [16], FreeBSD Jails [17]. A popular example is Linux Containers (LXC) [18] 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 [19], SELinux [20], 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

[14] https://www.docker.io/the_whole_story/
[15] http://openvz.org/Main_Page
[16] http://www.oracle.com/technetwork/server-storage/solaris/containers-169727.html
[17] http://www.freebsd.org/cgi/man.cgi?jail
[18] https://linuxcontainers.org/
[19] https://wiki.ubuntu.com/AppArmor
[20] http://selinuxproject.org/page/Main_Page

- 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].

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 [21]. This has proven to be a significantly useful feature of these containers for seamless software upgrade (Section 4.3). 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 [22]. 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 [23] clusters or on a service provider's infrastructure such as Digital Ocean [24].

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. 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 [25] 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 (Figure 3).

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.

[21] https://linuxcontainers.org/
[22] https://www.docker.io/the_whole_story/
[23] https://www.openstack.org/
[24] https://www.digitalocean.com/
[25] http://git-scm.com/

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 [26].

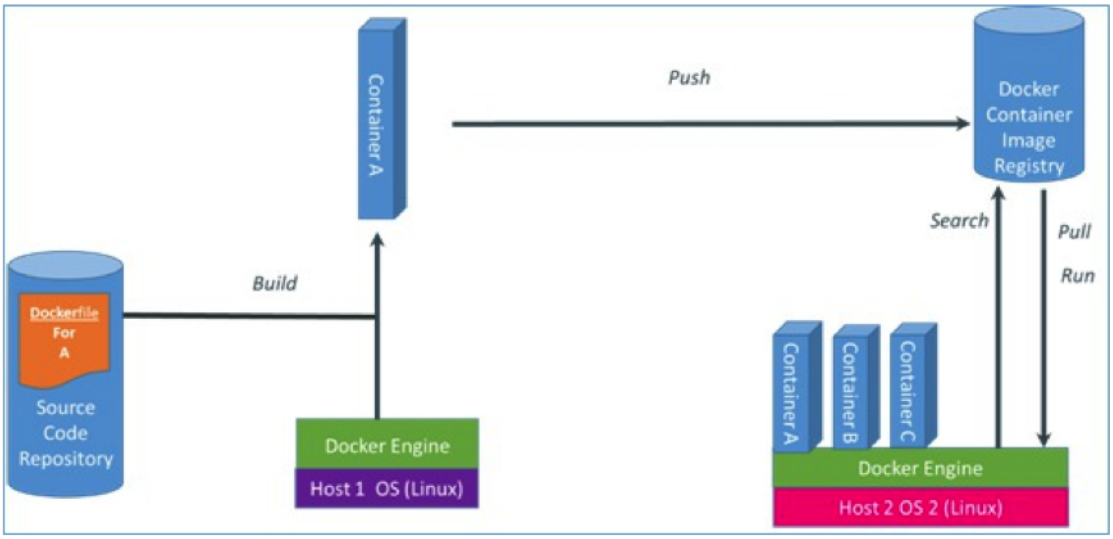


Figure 3: Basic Docker Function [26]

## 5.2 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) [27] are the most common messaging standards.

### 5.2.1 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]. 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.

[26] https://www.docker.io/the_whole_story/
[27] http://zeromq.org/

### 5.2.2 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 [28].

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`.

More recently, a ZeroMQ alternative, called nanomsg, has been proposed by the same team who developed ZeroMQ [29]. Similar to ZeroMQ, nanomsg is aimed to make the networking layer fast, scalable, and easy to use; however, it has been reported to be more lightweight than ZerMQ. In addition, in ZeroMQ each individual object is managed exclusively by a single thread. This strategy can cause issues such as inability to implement request resending in REQ/REP protocol and PUB/SUB subscriptions not being applied while application is doing other work. However, in nanomsg the objects are not tightly bound to particular threads as a result the aforementioned issues do not exist.

## 5.3 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 [30] 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).

## 5.4 HTTP Requests

A HTTP request has the following format:

```
<Request> ::=   <Request-Line>
                <general-headers>
```

[28] http://zeromq.org/
[29] http://nanomsg.org/
[30] http://oauth.net/

```
                |<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 .

<*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: */*
```

### 5.5 HTTP Responses

A 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://192.168.50.40/xmlrpc.php
Vary: Accept-Encoding
Content-Length: 7467
Content-Type: text/html; charset=UTF-8

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

# 6 Design

To manage the ever-increasing size of E-commerce and web related services large-scale clusters are deployed to virtualized operating systems running on hypervisors. However, hypervisors have a significant overhead when all that is required is a simple way to run clustered instances of applications. 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 EC2 [31].

In addition to the use of virtualization, continuous delivery and automated deployments are critical to reduce the cost of maintaining large clusters of software that require regular updates and feature releases. Although many companies use automated deployment tools, continuous delivery all the way into production is extremely rare due to the significant risk imposed by applying automated updates directly into production.

## 6.1 Dynamic Software Update System

To support reliable dynamic software updates a simple lightweight system has been developed suitable for continuous delivery into production. The system runs the updated version concurrently with previous stable versions. The updated version is monitored automatically and if the updated version behaves incorrectly, all requests will be seamlessly routed to the previous stable version with no downtime.

Four mechanisms are supported to enable different reliable update strategies depending on the specific software requirements.

Instant Update - 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. 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.

Session Update - This update mechanism only switches new sessions to the updated version. A new session is identified by no requests being received from a client within a configured time period. As with Instant Update, the non-updated version however remains running and will be immediately available to process requests if the updated system does not behave correctly. This approach is useful when a low risk update is being made that provides features, which are important to make available to users as rapidly as possible. For example, a low risk update that is providing a new profit making service.

Gradual Update - This mechanism switches new sessions gradually to the updated version over multiple days or weeks. This approach is particularly useful for risky or complex updates that have a high potential to introduce system instability. This technique is also less risky since the new software version incrementally receives a greater percentage of requests.

Concurrent Update - In this mechanism the updated and non-updated versions run concurrently and process identical requests. In this strategy, both the old and the new versions handle all requests and the characteristics of each response is compared to ensure the update is behaving correctly.

In all four mechanisms the behavior of the updated software is automatically monitored to confirm the application is behaving correctly. The status code and ability to receive incoming requests is used to verify that the updated application is behaving correctly. Figure 4 demonstrates the component interactions within the proposed dynamic software update system. The focus for the dynamic software update system is on HTTP requests and responses. This focus allows the use of HTTP headers to identify user sessions. In addition, it allows efficient comparison between large responses without the need to buffer the entire response body.

[31] https://www.appeagle.com/ecommerce-news/ecommerce-is-on-the-rise-in-2013/

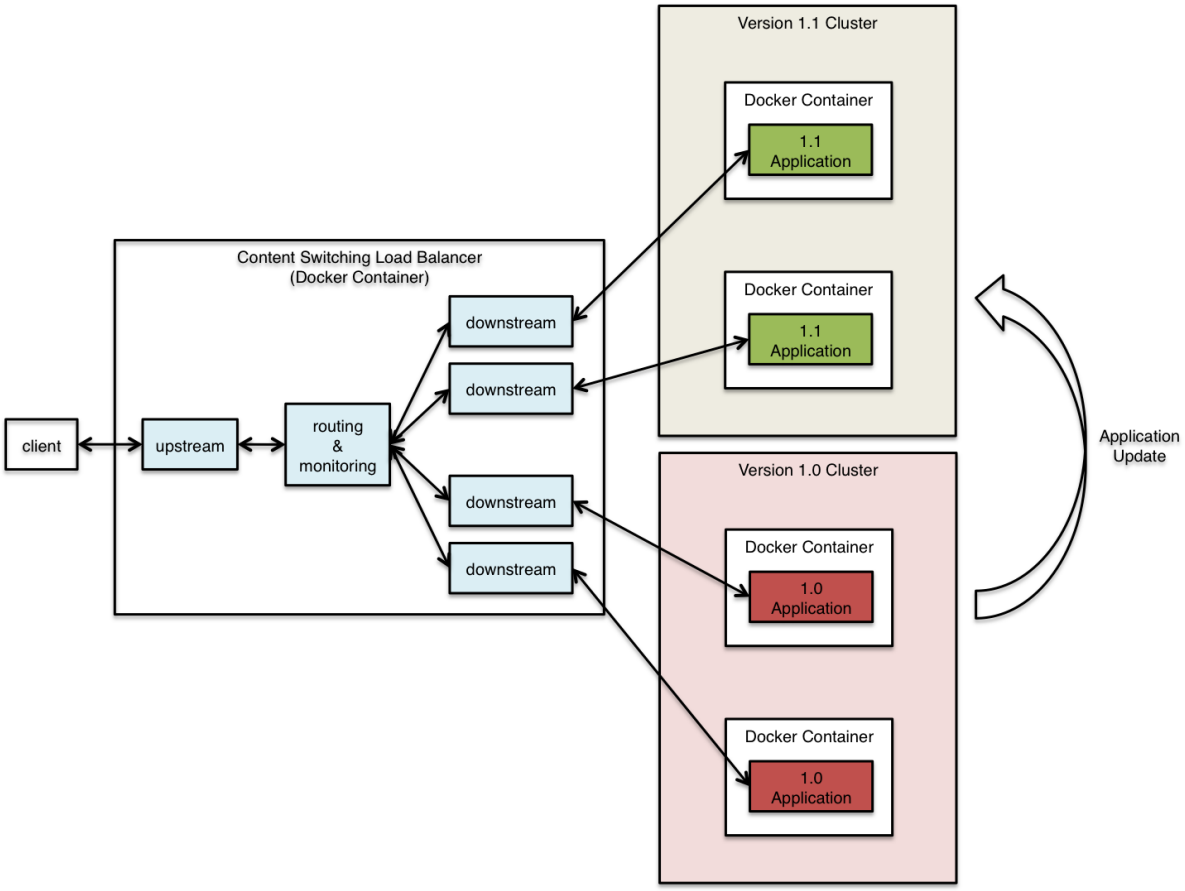


Figure 4: Component interactions within the proposed dynamic software update system

## 6.2 Technology Stack

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

- Go [32] programming language for building the proxy *i.e.* the content switching load balancer. Go is chosen because of its simplicity, efficiency, scalability, highly concurrency and garbage collection. Other fast programming languages such as C$^{++}$ and Java was 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 or nanomsg as message frameworks that support highly efficient load balancing and routing of messages without the need for any middleware.

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

# 7 Implementation

To produce an efficient and reliable dynamic software update system several versions of a dynamic software update proxy was implemented using Go programming language as described in this section.

## 7.1 Message Based Proxy Using ZeroMQ

The first proxy developed to support dynamic software update was a HTTP reverse proxy based on ZeroMQ messaging system. A custom Dockerfile was written that creates a Docker container running an example web service using Netty [33]. The proxy uses different types of ZeroMQ sockets such as STREAM, REQ, REP and it is arranged as shown in Figure 5. 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.

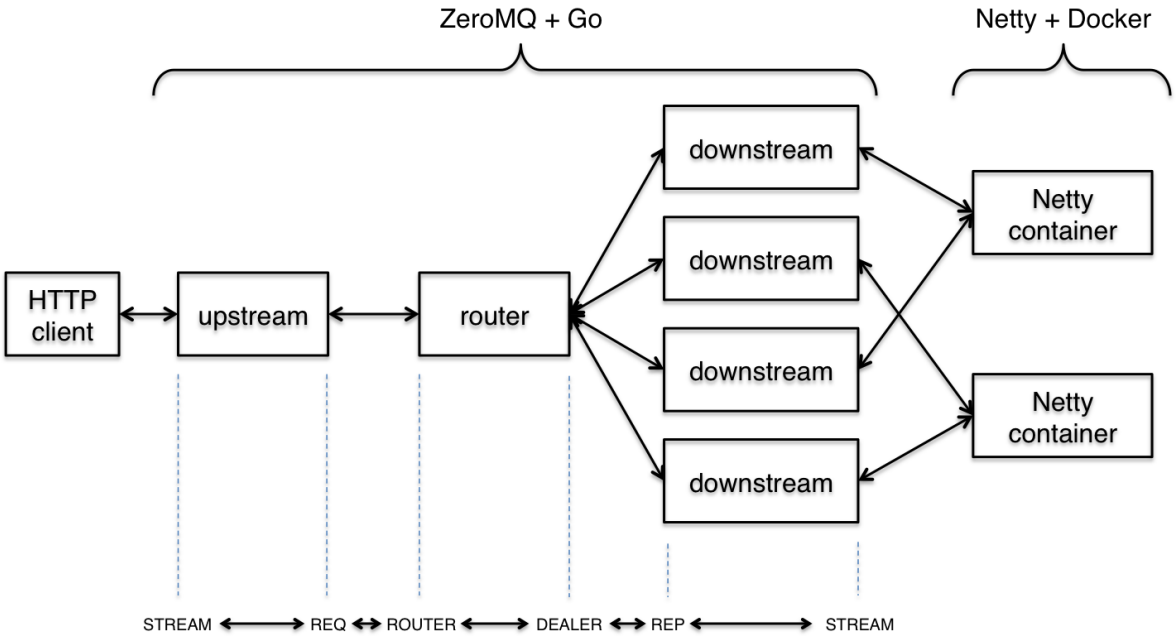


Figure 5: Message Based Proxy Using ZeroMQ

[33] http://netty.io/

### 7.1.1 Performance Evaluation

Detailed analysis of the reverse HTTP proxy showed that when the HTTP requests are chunked (Section 5.4), 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 such as nanomsg 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.

## 7.2 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.

### 7.2.1 Socket Based Proxy With Content Counting

The first version of the socket based proxy was designed as demonstrated in figure 6. The proxy consists of a listener that listens in an *Accept* loop for new TCP connections.

When a new connection is received, a TCP connection between the client and the proxy is created over which the HTTP request read from the socket in *Read* loop. The routing component creates another connection between the proxy and the server. The HTTP request is 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.

**Content Counting**
The content counting proxy uses the HTTP format to detect the end of each request and response 5.3. 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 5.4). 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 5.3).

**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 5.3). 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.

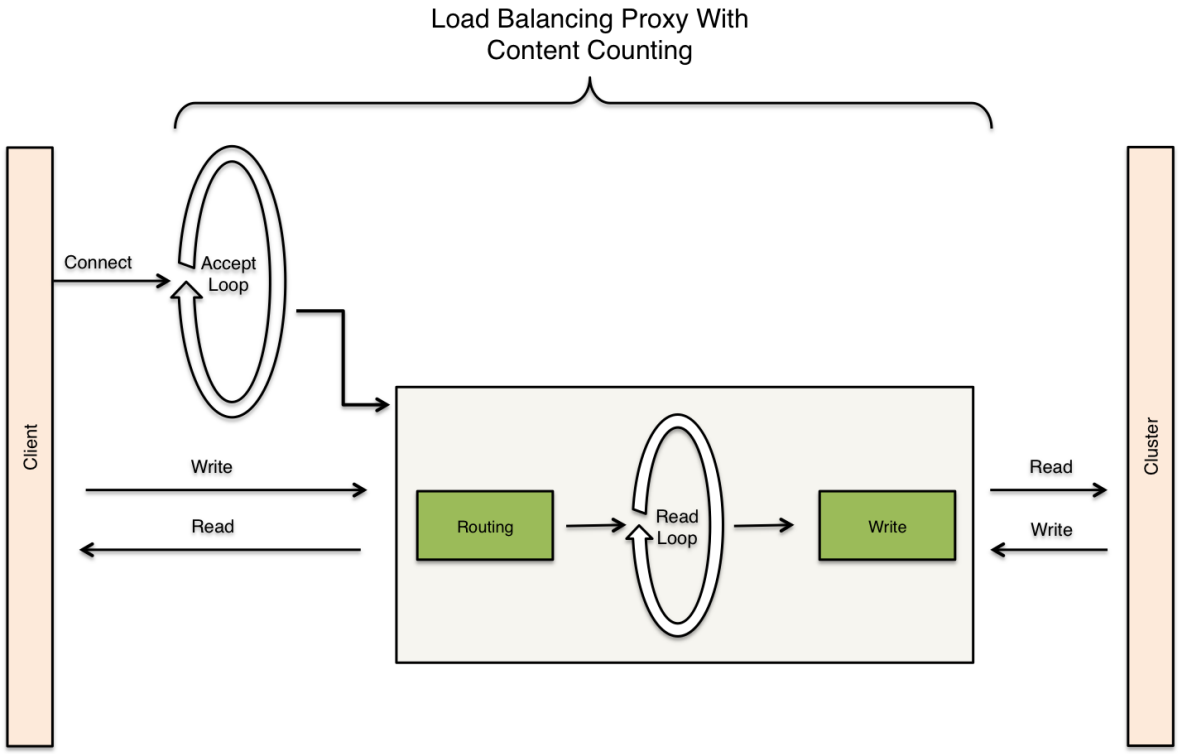


Figure 6: Socket Based Proxy With Content Counting

**Performance Evaluation**

To evaluate the performance of the aforementioned proxy a HTTP benchmarking tool called *wrk* [34] 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 [35] and kqueue [36].

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
```

[34] https://github.com/wg/wrk
[35] http://man7.org/linux/man-pages/man7/epoll.7.html
[36] http://www.freebsd.org/cgi/man.cgi?query=kqueue&sektion=2

```
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 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.

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.

### 7.2.2 Socket Based Staged Proxy With End Of File Signal

The final version of the socket based proxy was implemented using a Stage Based Design (SBD) as demonstrated in figure 7. The SBD approach was used to improve the code design, simplicity and to support good test coverage. Additionally, the SBD also promoted encapsulation, separation-of-concerns and allowed inversion-of-control and dependency-injection which 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 7.2.1 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 7 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 occured in the content counting proxy 7.2.1, 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 7 and their functionality is explained below.

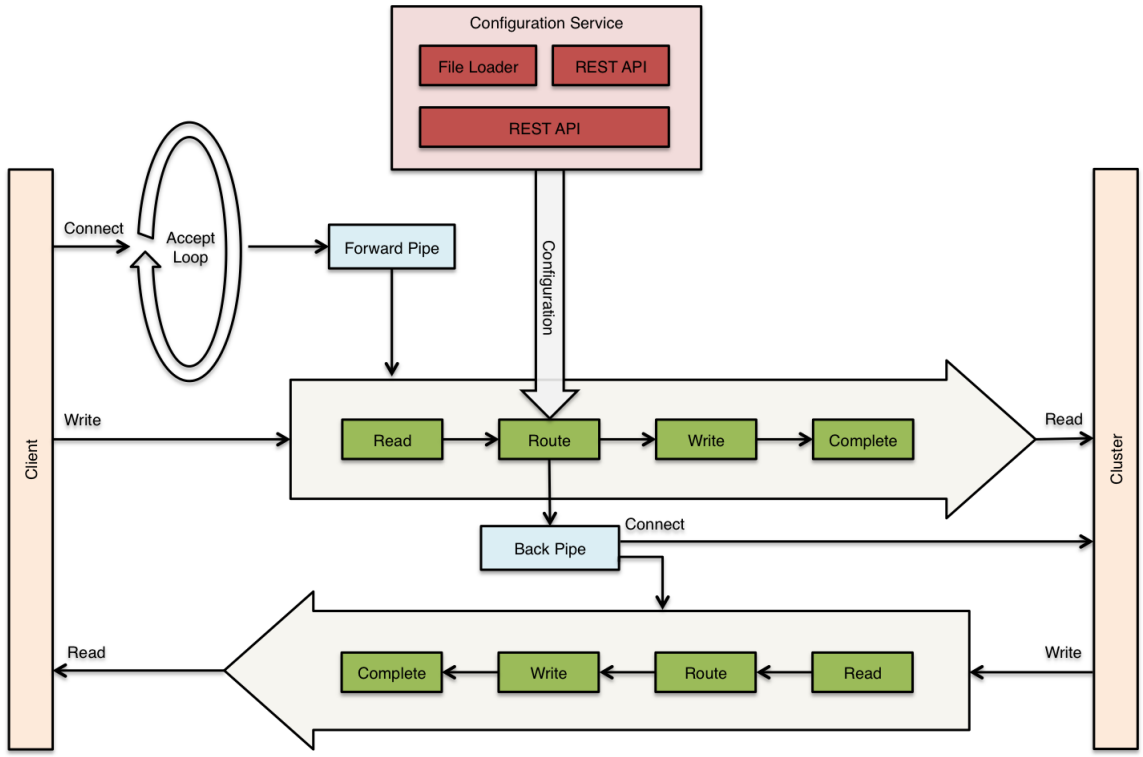


Figure 7: Socket Based Staged Proxy Design Supporting Dynamic Software Update

### Accept Loop

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

### 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
}
```

**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 7) 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 7.2.1, which is very inefficient and unreliable.

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

Routing in *Forward Pipe* - When a new request arrives in the *Route* stage, the *Route* checks the upgrade transition mode of the most updated cluster and forwards the request chunks to the correct instance of the cluster accordingly. Currently the proxy supports four different upgrade transition modes; INSTANT, SESSION, GRADUAL and CONCURRENT (Section 6.1). The *Route* stage distributes the requests using the modes according to the rules summarized below:

- **INSTANT** - If the upgrade transition mode in the most updated version of the cluster is defined as INSTANT, *Route* stage forwards the requests to the latest cluster. If no upgrade transition is defined in the cluster configuration, INSTANT mode will be the default mode (Section 7.5.1).

- **SESSION** - When upgrade transition mode of a cluster is defined as SESSION, the proxy will generate a session cookie with a configured expiry time for each request forwarded to that cluster. The session cookie will be added as a header to the HTTP response using Set-Cookie. If the request arrived in *Route* stage has an assigned cookie and it is not timed out, *Route* stage forwards the request to the cluster associated with the cookie, otherwise requests will be forwarded according to the upgrade transition mode of the cluster with highest version number. If the cluster configuration has an upgrade transition but the mode is not defined, SESSION mode will be assigned to that cluster.

- **GRADUAL** - When a client connects to a cluster with GRADUAL transition mode, a transition cookie will be added to responses associated to that client. The transition cookie refers to a percentage determining when the client should switch to the new version of the system. As an example, if the transition percentage is 15%, this means that the *Route* stag forwards the first 15% of the requests to the old version before forwarding the requests to the new version.

- **CONCURRENT** - When the upgrade transition mode is set to CONCURRENT, the *Route* stage forwards requests concurrently to the two most updated cluster versions.

If the user decide to delete the updated version of the cluster or an error occurs while routing the requests to a cluster, the request will be forwarded to the older version of the cluster. If routing to all versions failed, the process stops and returns an appropriate error to the client. 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 differently according to the configuration of the cluster which the response is originated from. If the upgrade transition mode in the cluster is defined as INSTANT, the response will be sent to the client with no further modifications. If the mode is set to SESSION or GRADUAL, *Route* stage sets the session cookie or the transition cookie to the response respectively before forwarding it to the client. Finally, if the mode is defined as CONCURRENT, the response from each cluster version will be parsed and HTTP headers such as status code, Transfer-Encoding, Content-Length and the response delays are compared. If the characteristics of the response from the new cluster is identical to the old one, only the response from the new version will be send to the client. Otherwise the response from the new version will be dropped and the old version's response will be used.

**Write Stage**
*Write* stage writes received requests/responses to an underlying data stream. The writing process stops if any errors such as *ErrShortWrite* encountered while writing to a 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. This proved to be essential for generating a fast and reliable dynamic software update system.

**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 the received response and creates *Read*, *Route*, *Write* and *Complete* stages for forwarding the response to the client.

## 7.3 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.

## 7.4 Command Line Interface

The proxy runs 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
                  },
                  "cluster": {
                      "servers":[
                          {"hostname": "127.0.0.1", "port": 1034},
```

```
                        {"hostname": "127.0.0.1", "port": 1035}
                    ],
                    "version": "1.0"
                }
            }

  -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"
```

## 7.5 REST API

The proxy provides a simple REST API (Figure 7) 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

### 7.5.1 PUT /configuration/cluster

The PUT request to */configuration/cluster* is used to add a new cluster to the proxy. The format of the request body is as follows:

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

The JSON fields are used for the following reasons:

- **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 10.1 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.

### 7.5.2 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 10.1.

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

### 7.5.3 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 10.1 shows an example of a DELETE request and the response received from the proxy.

## 7.6 Dockerising the proxy

Talk about how I dockerised the proxy

why didn't use fleet system and coreOS?

## 7.7 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 A FIGURE FOR TESTING XXXXXXXXXXXXXXXXXXXXXXXXXXXXX

### 7.7.1 Unit Testing

These tests focus on low level checking of the functionality of each proxy component. Inversion-of-control and dependency-injection (Section 7.2.2) allowed dependencies to be injected into each component and be mocked. The unit tests cover both positive and negative cases resulting in overall of xxxxxxx line coverage and xxxxxxx branch coverage.

XXXXXXXX MAYBE ADD A SCREEN SHOT XXXXXXXXXXXXXXXXXX

### 7.7.2 Integration Testing

These testes focus on the integration between different components of the REST API and the poxy. 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 analyzed. 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 or when a cluster configuration is deleted from the proxy.

### 7.7.3 System Testing

The aim of these tests is to check that the 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 are executed by starting the proxy with an initial configuration. Then HTTP requests is fired at the proxy and the test checks that the proxy is distributing the requests to the cluster of servers correctly. Afterwards, the four different upgrade modes described in section 6.1 are applied via the REST API and the behavior of the proxy is tested. Finally, the proxy is tested when clusters are deleted from the proxy.

# 8 Evaluation

The system to support dynamic software update will be evaluated as described below.

## 8.1 Qualitative Analysis

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.
Automated Software Update - To prove the suitability of the proposed system for continuous delivery, application updates will be performed using both a bash script [37] and a puppet manifest [38]. Puppet has been chosen since it is the most widely used automated deployment tool. In addition bash has been chosen since all automated deployment tools have the ability to run bash commands.

## 8.2 Quantitative Analysis

Load Test - A load test will be used to measure the performance as the number of requests increases. This test will demonstrate the delay incurred by the content switching load balancer and the maximum number of requests that the system can handle.
Saturation Test - This test will be performed to measure the performance of the system when it is has been exposed to a moderate number of requests over a prolonged period of time. This test will demonstrate that the system can run for a prolonged period of time without any degradation, such as memory leaks.
Update Speed -Rapid Updates will be performed to measure how quickly the requests will be transferred to the updated version of the software.
Software Recovery - Bugs will be introduced to the new version of the software to calculate the speed of switching to the non-updated version.

[37] http://www.tldp.org/LDP/abs/html/
[38] https://puppetlabs.com/

# 9 Conclusions & Future Plans

xxxxxxxxx

xxxxxxxxx

xxxxxxxxx

# 10 Appendices

## 10.1 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*.

***Request Body***

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

**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": 0
  }
}
```

**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-60030
    8a8245e -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": 0
    }
  },
  {
    "cluster": {
      "servers": [
        {
          "hostname": "",
          "port": 0
        },
        {
          "hostname": "",
          "port": 0
```

```
        }
       ],
      "upgradeTransition": {
        "mode": "CONCURRENT"
      },
      "uuid": "",
      "version": 0
    }
  }
]
```

### 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,firstnumber=1]
[
  {
    "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":"01
    386f1f-2608-11e4-ab79-600308a8245e","version":"1.1"}},{"cluster":{"servers
    ":[{"hostname":"127.0.0.1","port":1034},{"hostname":"127.0.0.1","port":103
    5}],"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-60030
    8a8245e -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
```

# 11 References


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

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

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

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

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

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

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

[8] Akhil Sahai, Calton Pu, Gueyoung Jung, Qinyi Wu, Wenchang Yan, Galen S. Swint. *Towards Automated Deployment of Built-to-Order Systems.*, Ambient Networks, 2005

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

[10] George Apostolopoulos, David Aubespin, Vinod Peris, Prashant Pradhan, Debanjan Saha *Design, implementation and performance of a content-based switch.*, Nineteenth Annual Joint Conference of the IEEE Computer and Communications Societies, 2000.

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

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

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

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

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

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

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

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

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

[20] Gautam Altekar, Ilya Bagrak, Paul Burstein, and Andrew Schultz. *OPUS: Online patches and updates for security.*, USENIX Security Symp, 2005.

[21] Andrew Baumann, Jonathan Appavoo, Robert W. Wisniewski, Dilma Da Silva, Orran Krieger, and Gernot Heiser. *Reboots are for hardware: Chal- lenges and solutions to updating an operating system on the fly.*, USENIX Annual Tech. Conf., 2007.

[22] Iulian Neamtiu, Michael Hicks, Gareth Stoyle, and Manuel Oriol. *Practical dynamic software updating for C.*, ACM SIGPLAN Conf. on Programming Language Design and Implementation, 2006.

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

[24] Gareth Stoyle, Michael Hicks, Gavin Bierman, Peter Sewell, and Iulian Neamtiu. Mutatis mutandis: *Safe and predictable dynamic software updating.*, ACM Trans. Program. Lang. Syst., 29(4), 2007.

[25] ] C. M Hayden, E. K Smith, M. Hicks, and J. S Foster. *State transfer for clear and efficient runtime updates.*, In Proc. of the Third IntâĂŹl Workshop on Hot Topics in Software Upgrades, pages 179 - 184, 2011.

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

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

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

[29] Yves Vandewoude, Peter Ebraert, Yolande Berbers, and Theo DâĂŹHondt. *Tranquility: A low disruptive alternative to quiescence for ensuring safe dynamic updates.*, IEEE Trans. Softw. Eng., 2007.

[30] Jonathan E. Cook and Jeffrey A. Dage. *âĂIJHighly reliable upgrading of componentsâĂİ*, ICSE Conf. on Proceedings of the 21st international conference on Software engineering, 1999.

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

[32] Kaushik Veeraraghavan, Peter M. Chen, Jason Flinn, and Satish Narayanasamy. *detecting and surviving data races using complementary schedules*, SOSP, 2011.

[33] Petr Hosek and Cristian Cadar. *Safe software updates via multi-version ex- ecution.*, IntâĂŹl Conf. on Software Eng., pages 612-621, 2013.

[34] Cristian Cadar and Petr Hosek. *Multi-version software updates.*, In Proc. of the Fourth IntâĂŹl Workshop on Hot Topics in Software Upgrades, 2012.

[35] Liming. Chen and Algirdas Avizienis. *âĂIJN-version programming: A fault-tolerance approach to reliability of software operationâĂİ*, in FTCS, 1978.

[36] N.M. Mosharaf Kabir Chowdhurya,1, Raouf Boutaba b. *A survey of network virtualization*, Computer Networks, 2010.

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

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

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

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

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

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

[43] Mark Hapner, Rich Burridge, Rahul Sharma, Joseph Fialli, Kate Stout. *Java Message Service*, In Oracle America, Inc., 2012.

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

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