Tatyana Krylosova

Implementing Container-based Virtualization in a Hybrid Cloud

Helsinki Metropolia University of Applied Sciences
Bachelor of Engineering
Information Technology
Thesis
24 April 2014
The goal of the project was to set up a hybrid cloud infrastructure where cloud bursting could be potentially implemented for application and system deployment using lightweight container-based virtualization. The project aimed to explore and discuss the features, benefits and challenges of cloud computing and particularly of the hybrid cloud approach.

The hybrid cloud concept implies the use of public cloud services alongside the internal facilities. Google Compute Engine was used as a public cloud service providing Virtual Private Cloud functionality. Interconnections between the clouds were created using OpenVPN, a functional and secure open source VPN solution. The implementation of container-based virtualization in the hybrid cloud was tested with Docker, the new promising software facilitating flexible application deployment.

The result of this project was a hybrid cloud test environment combining the internal resources at Metropolia premises and Google Compute Engine cloud. The project confirmed that Docker containers and OpenVPN tunnels could be successfully used to create a hybrid cloud infrastructure with the potential of further enhancements through cloud bursting mechanisms. In the future, the containers management can be automated by employing additional tools, such as OpenStack Nova, which, in combination with Docker, would provide a powerful tool for application development and deployment in a cross-cloud environment.

### Keywords
- cloud computing, hybrid cloud, Google Compute Engine, OpenVPN, Docker, container-based virtualization
## Contents

### Abbreviations

### 1 Introduction 1

### 2 Cloud Computing 2

#### 2.1 Definition of Cloud Computing 2

#### 2.2 Service Delivery Models 3

#### 2.3 Cloud Types 5

#### 2.4 Advantages and Disadvantages of Private and Public Clouds 6

#### 2.5 Cloud Bursting 7

### 3 Virtual Private Cloud 9

#### 3.1 Virtual Private Cloud Overview 9

#### 3.2 Google Compute Engine Infrastructure 10

### 4 Container-based Virtualization 11

#### 4.1 Container-based Virtualization Overview 11

#### 4.2 Benefits and Challenges of Container-based Virtualization 13

#### 4.3 Open Source Docker Container Virtualization 14

### 5 Proof of Concept Implementation of a Hybrid Cloud Infrastructure 17

#### 5.1 Task Definition 17

#### 5.2 Interconnection between Two Private Clouds 18

##### 5.2.1 Network Preparation 18

##### 5.2.2 VPN Server Configuration 19

##### 5.2.3 VPN Client Configuration 22

#### 5.3 Creating a Virtual Machine on the Google Compute Engine Cloud 23

#### 5.4 Docker Setup and Configuration 26

#### 5.5 Hybrid Cloud Configuration with a Full Mesh VPN Topology 28

### 6 Results and Discussion 30

### 7 Conclusion 33

### References 35

### Appendices
Appendix 1. Server.conf Configuration File
Appendix 2. Dockerfile for Running SSH Service
<table>
<thead>
<tr>
<th>Abbreviation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ACL</td>
<td>Access Control List</td>
</tr>
<tr>
<td>API</td>
<td>Application Programming Interface</td>
</tr>
<tr>
<td>CA</td>
<td>Certificate Authority</td>
</tr>
<tr>
<td>CPU</td>
<td>Central Processing Unit</td>
</tr>
<tr>
<td>GCE</td>
<td>Google Compute Engine</td>
</tr>
<tr>
<td>IaaS</td>
<td>Infrastructure as a Service cloud delivery model</td>
</tr>
<tr>
<td>IP</td>
<td>Internet Protocol</td>
</tr>
<tr>
<td>IT</td>
<td>Information Technology</td>
</tr>
<tr>
<td>NIST</td>
<td>National Institute of Technology and Standards</td>
</tr>
<tr>
<td>OS</td>
<td>Operating System</td>
</tr>
<tr>
<td>PaaS</td>
<td>Platform as a Service cloud delivery model</td>
</tr>
<tr>
<td>PKI</td>
<td>Public Key Infrastructure</td>
</tr>
<tr>
<td>RAM</td>
<td>Random Access Memory</td>
</tr>
<tr>
<td>SaaS</td>
<td>Software as a Service cloud delivery model</td>
</tr>
<tr>
<td>SCP</td>
<td>Secure Copy</td>
</tr>
<tr>
<td>SSH</td>
<td>Secure Shell</td>
</tr>
<tr>
<td>SSL</td>
<td>Secure Sockets Layer</td>
</tr>
<tr>
<td>TCP</td>
<td>Transmission Control Protocol</td>
</tr>
<tr>
<td>TLS</td>
<td>Transport Layer Security</td>
</tr>
<tr>
<td>UDP</td>
<td>User Datagram Protocol</td>
</tr>
<tr>
<td>VLAN</td>
<td>Virtual Local Area Network</td>
</tr>
<tr>
<td>VM</td>
<td>Virtual Machine</td>
</tr>
<tr>
<td>VPC</td>
<td>Virtual Private Cloud</td>
</tr>
<tr>
<td>VPN</td>
<td>Virtual Private Network</td>
</tr>
</tbody>
</table>
1 Introduction

Computing technologies are changing and evolving rapidly. Nowadays, data is often stored on the servers grouped into clusters and sharing storage. This approach provides multiple benefits including high-availability, fault tolerance and more efficient use of resources. The availability and growth of high-capacity networks and expansion of low-cost computers and mobile devices support the delivery of computer resources and applications as a service over a network. The widespread application of hardware virtualization and service-oriented architecture lead to advancements in cloud computing.

The goal of the project is to set up a hybrid cloud infrastructure where cloud bursting could be potentially implemented for application and system deployment using lightweight container-based virtualization. The project aims to explore and discuss the features, benefits and challenges of cloud computing and particularly of the hybrid cloud approach.

Cloud computing brings new opportunities into many areas of computing and employs different virtualization technologies. In a variety of cloud types and delivery models, each user can choose the approach which will be the most effective for a given task. A hybrid cloud is considered to be the most flexible and resilient cloud model, as well as potentially the most appropriate cloud technology for educational purposes.

I have chosen this topic because cloud computing is one of the most important trends in the IT world. Knowledge of cloud technologies and practical experience in configuration of a cloud environment is important for network engineers and system administrators and benefits a professional career in these fields. It may also introduce new technologies and take advantage of their benefits for the education process in Helsinki Metropolia University of Applied Sciences.

The scope of this project is limited to building an infrastructure that combines private and public clouds and employs a container-based virtualization technique. However, the project does not include the application development or deployment of any specific application in the hybrid cloud, nor does it cover automation of cross cloud deployment services.
2 Cloud Computing

2.1 Definition of Cloud Computing

Cloud computing is becoming more and more popular in the IT industry. Rapid development of virtualization and distributed computing and propagation of access to the high-speed Internet have accelerated the growth of cloud technologies. Many people use cloud services on an everyday basis without even realizing it. Cloud computing comes in many forms and serves different purposes, which leads to its different definitions.

According to the National Institute of Technology and Standards (NIST) definition, “cloud computing is a model for enabling ubiquitous, convenient, on-demand network access to a shared pool of configurable computing resources (e.g., networks, servers, storage, applications, and services) that can be rapidly provisioned and released with minimal management effort or service provider interaction” [1]. In simpler terms, cloud computing is the delivery of different kind of IT resources as a service over a network. Most often the resources are provided by distant centralized data centers. Although some definitions refer to cloud computing as Internet-based computing, these resources are not necessarily hosted on the Internet. For end users, it makes no difference where the data are physically stored or where servers are physically located, and how they are run and administered.

Certain characteristics differentiate cloud services from traditional hosting. Among the main distinct features of cloud computing are on-demand self-service, “pay per use” approach, broad network access, location-independent resource pooling and high scalability and elasticity. Cloud infrastructure provides on-demand “pay per use” self-service. Computing capabilities, such as server time and network storage, can be provisioned without requiring human interaction with a service provider. Cost of the services is usually calculated by the minute or the hour. [1]

Broad network access means that cloud resources are available over a network and accessed through standard mechanisms used by thin or thick client platforms. Consumers can access cloud services using desktop computers, laptops, tablets, and mobile phones. A service is fully managed by the provider, and typically, in order to use
computing capabilities a consumer needs only a personal computer and access to the Internet. [1]

With location-independent resource pooling, the provider’s computing resources are pooled to serve multiple consumers. Resources, such as storage, processing power, memory, and network bandwidth, are dynamically assigned depending on the consumer demand. A consumer has no knowledge or control over the exact location of the provided resources, but it may be possible to specify location at more general level and choose in which part of the world or which country the resources are hosted. [1]

Scalability and elasticity are important features of cloud services. Capabilities in the cloud can be elastically provisioned and released, manually or automatically, which provides high scalability and allows to rapidly adjust to the customer needs. The capabilities available for provisioning often look unlimited for a consumer. A consumer can provision as much of a service as needed and can make changes any time. [1]

Cloud computing is a wide concept, and each of its parts has its own specifics. A variety of cloud types and implementations serve diverse consumer needs. Different service delivery models and different types of clouds will be discussed in the following subsections.

2.2 Service Delivery Models

In cloud computing, the resources are delivered as services. Although a wide variety of resources can be offered within the cloud concept, the most common service delivery models are Software as a Service (SaaS), Platform as a Service (PaaS), and Infrastructure as a Service (IaaS).

In Software as a Service cloud model (SaaS), the consumer uses the provider’s applications running on a cloud infrastructure. The service provider hosts both the application and the data, and the end user can use the service with various client devices through a thin client interface, such as a web browser, or a program interface. The underlying cloud infrastructure including network, servers, operating systems, storage and application capabilities is controlled by a cloud service provider. However, the end user often can adjust limited application settings. [1; 2] Well-known examples of SaaS are
Gmail and GoogleDocs. This cloud model is widely used and covers many services from web-based email systems to database solutions.

Platform as a Service cloud model (PaaS) is defined as a set of software and product development tools hosted on the provider’s infrastructure. On this infrastructure, the consumer deploys self-created or acquired applications using programming languages, libraries, services, and tools supported by the provider. The underlying cloud infrastructure including network, servers, operating systems, or storage is managed and controlled by the service provider, but the customer can control the deployed applications and configuration settings for the application-hosting environment. [2; 3] Well-known examples of PaaS are Microsoft Azure and Google App Engine.

In Infrastructure as a Service cloud model (IaaS), the service provider delivers virtual server instances and Application Programming Interface (API) to create, access, configure and remove them. The consumer can provision processing power, storage, network components, and other computing resources and run software (operating systems and applications) based on individual preference. The underlying cloud infrastructure is managed by a service provider, but the consumer has control over the operating system, storage, installed services and applications, and partly networking components. [1; 2] Well-known examples of IaaS are Google Compute Engine and Amazon EC2.

Figure 1. Level of abstraction and control for cloud service delivery models. Reprinted from Meier (2010) [4].
The cloud service delivery models discussed above offer different levels of infrastructure control. Software as a Service gives the least control over infrastructure, while Infrastructure as a Service gives complete control of the operating system and application platform. Platform as a Service partly combines the simplicity of SaaS and the flexibility of IaaS. Figure 1 shows the level of abstraction and control characterizing different service delivery models. The decision on which cloud model to use depends on each unique situation.

2.3 Cloud Types

Based on different deployment models, four types of cloud computing services can be defined: private, community, public, and hybrid clouds. The division is based on who owns, manages, and accesses cloud resources.

A private cloud gives an ability to create a cloud infrastructure which is provisioned for internal customers of a single organization. The resources are owned and managed by the organization itself or a third party, and may be hosted internally or externally. A community cloud is an extended private cloud. It is a cloud infrastructure shared by several organizations and supporting a specific community with common requirements. The resources are owned and managed by one or more of the organizations in the community or a third party, and may be hosted internally or externally. [1] [3]

The infrastructure of a public cloud represents a publicly available pool of resources. “The resources are dynamically provisioned to the general public over the Internet by using web applications or web services”. Public clouds are owned and managed by a cloud service provider and hosted on its premises. [3] A hybrid cloud is a combination of private and public clouds. The clouds remain unique entities, but they are bound together into a single pool of resources that enables data and application portability. Programs and data can be moved easily from a private cloud system to the public cloud and back. If the private cloud is out of capacity, the additional resources can become quickly available and workloads can be provisioned on the public cloud. [1] [3]
2.4 Advantages and Disadvantages of Private and Public Clouds

Cloud computing implementations bring multiple benefits into IT infrastructure, but also raise specific challenges. Both private and public clouds have their own advantages and disadvantages. The decision about the best suitable cloud type is based on different factors. Companies need to consider many issues including security and availability requirements, workload variation, cost and the specifics of applications.

Private cloud implementations offer a number of advantages. Among the most significant benefits of private clouds are control over infrastructure, security, compliance and customization. Organizations have better control over their data because the hardware resides at their location. The ability to monitor and maintain all the data and physical infrastructure gives organizations complete oversight of their data. Since private clouds belong to a single organization, the hardware, data storage, and network can be designed to provide high levels of security. Also, enterprises can follow the compliance standards required for their industries. The control over the hardware, network and storage performance allows the company to achieve a high level of customization. [5]

Along with advantages listed above, a private cloud architecture brings disadvantages, such as higher cost, maintenance complexity and capacity ceiling. Organizations need to own the hardware, storage and networking resources and maintain all the resources which makes private clouds expensive and complex to deploy and maintain. Private clouds require system administrators, which leads to higher administration costs. The organization also has the risk of data loss due to physical damage of the system and has to implement proper measures to ensure high availability and fault tolerance of the resources. Another limitation is a capacity ceiling. [5] The amount of the resources that can be provisioned is limited by the company's data center capabilities.

In terms of scalability, flexibility, simplicity of use and price, public cloud service usually outmatches private clouds. With public clouds, the company can instantly provision virtually unlimited resources, scale them up or down to meet the demand, and pay only for the resources in use. The company does not have to purchase and maintain hardware because all servers are virtual and hosted on the service provider infrastructure. Creating and reconfiguring a server can take only minutes, and high-availability of services is a service provider's responsibility. [5]
Companies also need to keep in mind the disadvantages of public clouds: lack of control, speed limitation, lack of compliance, and perceived weaker security. Since providers are in charge of the data systems, some organizations may feel that they do not have enough control over their personal data. Public cloud services are delivered over the Internet, which means that the data transfer rate is limited to the Internet connection speed. It may become a problem especially for an organization that stores and transfers large amounts of data. Public cloud providers may not follow all the regulatory compliance required by an organization. Weaker security is viewed by some companies as the main disadvantage in public cloud service. Most of service providers have high level security measures, but customers with sensitive personal information often cannot trust this information to a third party. [5]

A hybrid cloud is the combination of private and public clouds. With a hybrid approach, the company can take advantage of the scalability and cost-effectiveness of public cloud and assure security of critical applications keeping them in a private cloud [5]. By enabling data and application portability, a hybrid cloud can ensure better performance and scalability. The disadvantage of a hybrid cloud is complexity. The company has to ensure that all necessary services can communicate with each other. To summarise this comparison of private and public clouds, it is important to keep in mind that the company does not necessarily have to choose between public and private cloud services. In many cases, creating an hybrid cloud may be the best solution.

2.5 Cloud Bursting

Cloud bursting is a technique used in hybrid cloud environments that combines internal resources with public cloud resources leased on a “pay per use” basis. More specifically, cloud bursting is an application deployment model which allows to run an application in a private cloud and burst it into a public cloud in case of increased demand for computing resources. [6] [7] By introducing cloud bursting, a hybrid cloud can overcome some challenges of private and public cloud services.

The main advantage of cloud bursting is that the organization pays for extra computing capabilities only when they are needed. The graph in figure 2 shows an organization demand of computing resources (number of servers). When the resources of the private site consisting of five servers are exhausted, cloud bursting will occur to respond to the demand spikes.
Cloud bursting is recommended for high performance, non-critical applications that handle non-sensitive information. “An application can be deployed locally and then burst to the cloud to meet peak demands, or the application can be moved to the public cloud to free up local resources for business-critical applications”. Cloud bursting can be implemented more easily and works more reliably for applications that do not depend on a complex infrastructure and do not need to be integrated with other internal applications, components and systems. [7]

Common cases where a hybrid cloud and cloud bursting can benefit include using public cloud as a fail-over platform and handling peak loads. Maintaining an unused infrastructure in order to provide business continuity is expensive. A redundant infrastructure can be built in a more cost-effective way using standby public cloud resources. Thus, a hybrid cloud is a reasonable choice for implementing fail-over strategy. Also, many companies encounter situations when, for a short time, an application has a higher workload than normal. [10] In this case, a hybrid cloud is again a cheaper option compared to maintaining the abandoned resources in the local premises.

Moving components to a hybrid cloud is associated with a few challenges, including data security, adjustments and reprogramming. Despite the cost efficiency and agility provided by hybrid clouds, many enterprises trust internal security measures more than the ones provided by a cloud provider. They are concerned about control over their infrastructure and confidentiality of their data, and fear that the violation might happen intentionally or by accident. In addition, one of the main priorities of cloud service pro-
The threats associated with hybrid clouds can be mitigated by using Virtual Private Cloud (VPC) and VPN solutions. Virtual Private Cloud allows customers to create their own virtual cloud within public cloud. The examples of VPC services offered by the biggest cloud service providers are Amazon VPC and Google Compute Engine. With VPC, the customer can control the virtual network environment, including IP addressing, creation of subnets, and configuration of firewalls. VPC can be connected with the internal IT infrastructure using a VPN connection. The Virtual Private Cloud and Google's implementation of it will be discussed in more detail in Chapter 3.

OpenVPN is an open source VPN solution which can be used to provide secure data exchange between public and private clouds. It uses OpenSSL-based encryption for securing the data and creates a secure tunnel for data exchange. It also supports multiple ways of authentication before establishing a secure connection, including verification of certificates, username/password credentials, smart cards, and firewall access control policies.

Cloud bursting typically relies on virtualization technologies to encapsulate applications in the form that can be relatively easily transitioned between different hardware platforms. One of the new technologies simplifying the deployment of applications is Docker container virtualization, which will be discussed in Chapter 4.
cloud solution is provided within a public cloud provider’s infrastructure” [11]. The VPC concept is similar to Virtual Private Network, but applied to cloud computing. In the same way as a VPN provides secure data transfer over the Internet, a VPC provides secure data transfer between an enterprise and a public cloud provider. Customers’ data stored in this type of cloud are isolated from each other. This can be done using security policies and implementing encryption, tunneling, private IP addressing or allocating a unique VLAN to each customer. [12]

Major cloud vendors offer virtual private cloud implementations along with their public cloud services. Amazon VPC is the most popular VPC implementation. It supports provision of logically isolated resources in the Amazon Web Services Cloud. The customer has full control over the virtual network environment and, for example, can select the IP address range, create subnets or configure route tables and network gateways. The network configuration is easily customizable. Multiple layers of security are supported, including security groups and network access control lists. Also, it is possible to set up a VPN connection between the customer’s internal datacenter and the Amazon cloud and use the VPC as an extension of the internal datacenter. [13]

3.2 Google Compute Engine Infrastructure

Google Compute Engine (GCE) is an IaaS cloud service launched by Google. GCE enables users to run virtual machines on the Google infrastructure. It is run on the global infrastructure that also runs Google’s search engine, Gmail, YouTube and other services. Google's infrastructure can easily facilitate launching of large compute clusters. Without any initial investment, users can run up to thousands of virtual CPUs on a system. [14] GCE offers easy and fast system deployment, and high scalability and performance.

Google does not use the term “Virtual Private Cloud” referring to the Google Compute Engine functionality, but it incorporates the idea of a VPC in network configuration of GCE. Customers can create a virtual network, private, public subnets, firewalls, routes, gateways, access control lists (ACL) before or after launching the VMs.

Google Compute Engine offers the capabilities to create virtual machines with various configurations, store data in the persistent block storage and manage network access
to virtual machines. GCE uses KVM as a hypervisor. Instances can run the Linux images provided by Google (Debian and CentOS). Premium operating system images (Red Hat Enterprise Linux and SUSE) are available for an additional fee. Another premium operating system Windows Server 2008 R2 is available in a limited preview [15]. It is also possible to import or build an image and run it on Compute Engine. GCE lets users specify the virtual machine properties, such as the number of CPU cores and the amount of memory. VMs in the same network can communicate with each other. They can also communicate with the rest of the world through the Internet. [14]

GCE service is delivered on a “pay per use” basis. All machine types are charged for a minimum of 10 minutes, after which, instances are charged in one-minute increments [14]. The price of a standard VM hosted in Europe with one virtual CPU core and 3.75 GB of memory is 0.077 USD. The price of the provisioned disk space is 0.04 USD per GB per month. [16] Using the additional features such as load balancing, protocol forwarding or static IP addresses will lead to extra costs.

All GCE resources belong to the global, regional, or zonal plane. For example, considering that static IP addresses are a regional resource, only resources that are part of the same region can use the static addresses of that region. If a region falls offline, it will not affect any other regions because all of them are isolated from each other. [14] This functionality can be employed to build high-availability systems by spreading the resources across different regions or zones.

4 Container-based Virtualization

4.1 Container-based Virtualization Overview

Nowadays, hypervisor-based virtualization is the most popular virtualization technology. It is flexible and works for nearly any guest operating system. However, there is another approach called container-based virtualization (or operating system virtualization). In some cases, container-based virtualization can be considered a feasible choice and offer performance benefits.

A container-based system “provides a shared, virtualized OS image consisting of a root file system and a safely shared set of system libraries and executables” [17]. Each VM
(referred to as a container) can be treated as a regular operating system: it can be booted, rebooted or shut down. Resources such as disk space, CPU and memory are assigned to each VM when it is created, but also they can be dynamically increased or decreased when a VM is running. Applications and users see the VM as a separate host. The virtualization layer ensures that the container does not interfere with other containers. [17]

Container-based virtualization allows to run a kernel with several different virtual machines installed on top of it. Unlike hypervisor-based virtualization, a container-based approach does not run a virtual machine as a complete operating system instance. Container-based VMs are rather partial instances of the operating system. This virtualization type is mostly used in Linux environments. [19] The architecture of container-based virtualization is shown in figure 3.

Figure 3. Container-based virtualization architecture. Reprinted from viux.com [18].

The container virtualization is built upon the standard host operating system and uses it as a base. Implementations of container-based virtualization are developed for different operating systems. For example, Parallels Virtuozzo virtualization can be set up on a Windows or a Linux host. LXC and OpenVZ are implementations for Linux.

Container virtualization does not run an entire guest OS, but instead it isolates the guest systems and it does not virtualize the hardware. A patched kernel and user tools are needed to run the virtual containers. The kernel isolates processes of the different containers and performs resource management. It is important to remember that
whereas all the virtual machines use the same kernel, they have their own file system, processes, memory and devices. [20]

4.2 Benefits and Challenges of Container-based Virtualization

Container-based virtualization has its own benefits and challenges which have to be considered when making a decision on the virtualization approach. Among the main benefits are performance and scalability of container-based solutions. However, the specifics of the container maintenance and guest OS limitations might imply challenges for this type of virtualization deployment.

The main difference between container-based and hypervisor-based virtualization is that container-based virtualization does not use complete virtual machines. This means that it does not cause the overhead of running completely installed operating systems. With partial virtualization, the container functions as a virtual machine, and users of the guest systems probably will not distinguish a system running in the container from a system running on bare metal or under a hypervisor. However, the container runs as an isolated application within the host operating system. Thus, container-based virtualization eliminates the need to duplicate certain OS functionality. For example, it does not duplicate hardware calls, and one operating system is responsible for managing all hardware access. [19]

Container-based virtualization brings advantages in terms of performance and scalability. While hypervisor virtualization usually has limits on how much CPU and memory resources can be assigned to a guest OS, the container-based solutions should be able to address all the CPUs and RAM available for the host kernel. [20] Since all of the containers share the same operating system, they are significantly smaller than VMs. A container can be created and launched faster than a full virtual machine. Containers also offer better performance for the applications they contain, because they eliminate the overhead of running software through the hypervisor. [21] Type of workload has to be considered when choosing the right virtualization approach. A container-based solution might be a better option, if there is a need to deploy dozens or hundreds of Linux guests. Containers benefit hosting environments and scenarios where a large number of Linux instances need to be consolidated [20].

The challenges of container-based virtualization include the container maintenance and
guest OS limitations. With container-based virtualization, installing a guest OS is not as straightforward as with hypervisor solutions where it can be installed from a CD or DVD. Running the container requires creating first a container template or using an available one. [20] Saving permanent changes to the container configuration might also require specific actions.

The choice of the operating system in a container-based solution is limited. Linux and Windows cannot be run together. With Linux virtualization, Linux guests are running on a Linux host. However, they are not limited to a specific distribution. For example, it is possible to run Debian and CentOS templates on top of a Ubuntu host. [20] The mentioned limitations make container-based virtualization not suitable for some systems. Container-based virtualization offers an approach to virtualization which has its own advantages and disadvantages. The main benefits of this approach include improved performance and scalability. Container-based virtualization solutions work well for the systems where only one operating system family is needed.

4.3 Open Source Docker Container Virtualization

Docker is an open-source engine which allows to “easily create lightweight, portable, self-sufficient containers from any application”. Any application and its dependencies can be packaged up as a container. The container built and tested on a developer laptop can run virtually anywhere. Docker containers can encapsulate any payload and run consistently on any Linux servers. They can be run on bare metal servers or on virtual machines, on OpenStack clusters or on a service provider’s infrastructure. [21]

Docker can be used for:

- automating the packaging, deployment and testing of web applications;
- deploying and scaling web apps, databases and backend services in a service-oriented environment;
- building custom lightweight PaaS environments. [21]

Docker is now under development and is not recommended to be used in production. A full production-ready release of the software is planned for April 2014. In the future, the Docker developers plan to introduce the services for indexing and signing images, and for creating private registries for Docker images. They also plan to offer commercial
support for Docker and are considering the release of an enterprise edition of the software. [22]

Docker offers new opportunities for developers, because it allows simple environment isolation and repeatability. Developers “can create a run-time environment once, package it up, and then run it again on any other machine” [21]. This environment is isolated from the underlying host. Thus, Docker eliminates inconsistencies between development, test, production, and customer environments and “improves the speed and reliability of continuous deployment and continuous integration systems”. [21] The technical features of Docker are explained in table 1.

Table 1. Docker Features. Data gathered from Docker Homepage [21].

<table>
<thead>
<tr>
<th>Feature</th>
<th>Short Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>File system isolation</td>
<td>The container runs each process in a completely separate root file system.</td>
</tr>
<tr>
<td>Resource isolation</td>
<td>System resources, such as CPU and memory, can be allocated differently to each process container, using cgroups.</td>
</tr>
<tr>
<td>Network isolation</td>
<td>The container runs each process in its own network namespace, with its own virtual interface and IP address.</td>
</tr>
<tr>
<td>Copy-on-write</td>
<td>Root file systems are created using copy-on-write, which makes deployment fast, memory-cheap and disk-cheap.</td>
</tr>
<tr>
<td>Logging</td>
<td>The standard streams (stdout/stderr/stdin) of each process container are collected and logged for real-time or batch retrieval.</td>
</tr>
<tr>
<td>Change management</td>
<td>Changes to a container’s file system can be committed to a new image and re-used to create more containers.</td>
</tr>
<tr>
<td>Interactive shell</td>
<td>Docker can allocate a pseudo-tty and attach to the standard input of any container.</td>
</tr>
</tbody>
</table>

Docker facilitates an easy way to build, modify, publish, search, and run containers. A Docker container carries both an application and all of its dependencies. Containers can be created manually or automatically using a DockerFile script. A modified Docker image can be saved to a new container using the Commit command. Also, containers can be pushed to a Central Registry and retrieved (pulled) from there using the Pull
function. They can be run, started and stopped and can reside on the internal servers or public instances. [21] Figure 4 shows the Docker infrastructure and the functions that can be applied to containers.

Figure 4: Basic Docker Functions. Reprinted from Docker Homepage [21].

With Docker, a new application on a host needs only its binaries or libraries, but not a new guest operating system. Several copies of the same application can be run using the shared binaries. When modifying an application, only the differences have to be copied. Thus, Docker provides efficiency and makes it easy to update applications. [21] One of the important Docker benefits is its simplicity. Docker eliminates the complexity of working with LXC containers and provides an API that can help administrators to inject containers into a larger scripted environment. Container-based virtualization might require complex configuration and extensive command line use. Docker aims to provide an interface that abstracts the complex details away. [22]

As explained above, containers can encapsulate entire applications and application environments, and can be managed through a Docker API. Therefore, Docker also introduces new opportunities for implementing PaaS cloud services, since it offers an easy way to deploy applications across different cloud providers. Docker can be used in combination with OpenStack cloud computing software to provide a powerful tool for cross cloud application development and deployment.
5 Proof of Concept Implementation of a Hybrid Cloud Infrastructure

5.1 Task Definition

The project aimed to set up a hybrid cloud infrastructure comprising internal Metropolia resources and a Google Compute Engine cloud. More specifically, the project was intended to implement the lightweight container-based virtualization using Docker, verify the possibility and feasibility of Docker integration into a hybrid cloud environment and examine its potential for distributed application and system deployment. The successful setup of such system would open up the opportunities for implementing cloud bursting and deployment of a new cloud infrastructure.

The project was divided into four stages:

- Interconnection between two private clouds
- Creating a virtual machine on the Google Compute Engine cloud
- Docker setup and configuration
- Hybrid cloud setup with a full mesh VPN topology.

The first stage of the project aimed to build interconnection between two private clouds. The project started with preparing the private cloud infrastructure. The further tasks included setup and testing of the VPN tunnel between the private clouds, as well as routing configuration if needed. The outcome of this stage had to be the full connectivity between the devices in both clouds.

The second stage covered creating and configuring a virtual machine hosted on Google's Compute Engine infrastructure. The virtual machine had to be accessible via SSH and had to permit incoming VPN connections on an agreed port. This VM would be used in the next stages for hosting Docker containers and acting as a VPN server accessible from anywhere on the Internet.

In the third stage of the project, Docker had to be installed on the virtual machines in the public and one of the private clouds. The addressing scheme was important consideration for a hybrid cloud setup. It had to guarantee the network consistency and connectivity. Therefore, the Docker instances had to be configured to use different IP addresses. A few simple containers were to be run for testing purposes.
The forth stage aimed to build a hybrid cloud infrastructure connecting two private clouds and the GCE cloud together. The gateways of each cloud had to represent a full mesh VPN topology. The VM hosted in the Google cloud would be configured as a VPN server, and both private VPN gateways would act as a VPN server and a VPN client. The certificate distribution and validity issues had to be considered when configuring the VPN. The desired outcome of this stage was defined as a hybrid cloud environment running lightweight containers and providing full connectivity for all its hosts including Docker containers and VPN clients.

5.2 Interconnection between Two Private Clouds

5.2.1 Network Preparation

The goal of this stage was to create a VPN tunnel between two private clouds hosted in Metropolia premises. In a simplified case, each cloud was represented by two virtual machines. In the first step, Ubuntu 12.04 was installed as the operating system on each virtual machine. Figure 5 displays the network diagram with initial IP configuration.

![Figure 5. Initial Network Configuration.](image)

VM1 and VM4 comprised the first private cloud. VM1 had two network adapters, and it was connected to the Labnet, by means of which it had access to the Internet. In this simple network topology, VM1 acted as a default gateway for VM4. VM2 and VM3 comprised the second private cloud and fulfilled the same roles as VM1 and VM4 correspondingly. After establishing a VPN tunnel, the local networks 172.16.1.0/24 and 192.168.1.0/24 had to be able to communicate to each other. Additionally, in order to
provide the Internet access for VM3 and VM4, Iptables NAT was configured on VM1 and VM2.

The next step was to choose the best suited VPN software. The VPN solution had to work on Linux virtual machines, be available on free-of-charge basis and meet the requirements of functionality, security and proper documentation. OpenVPN was chosen as a VPN implementation for this project. OpenVPN is open source software distributed under the GNU General Public License. It can be used for creating secure point-to-point or site-to-site connections in routed or bridged configurations. It can be configured to use any UDP or TCP port and bypass firewalls blocking standard VPN communication. OpenVPN provides an easy way to set up and use a Public Key Infrastructure (PKI) with SSL/TLS certificates for authentication and key exchange between the VPN server and clients. OpenVPN client implementations are available for all Linux distributions, OS X and Windows OS. [23] Therefore, OpenVPN was considered to be the best solution for implementing VPN tunnels between the two private clouds or the private and public clouds.

There are two editions of OpenVPN software: OpenVPN Access Server and OpenVPN Community Edition. OpenVPN Access Server is commercial software with GUI and some advanced features. Its drawback is that it allows only two client connections for free. Based on this limitation, OpenVPN Community Edition was considered to be a better choice for the project.

OpenVPN provides different authentication methods including pre-shared keys, username-password credentials and SSL/TLS certificates. In SSL/TLS mode, an SSL session is established with bidirectional authentication, which means that the client must authenticate the server certificate and the server must authenticate the client certificate [24]. Pre-shared static keys are simpler in use, but the SSL/TLS mode provides better security and therefore was chosen for this VPN implementation.

5.2.2 VPN Server Configuration

OpenVPN was installed from the Ubuntu repositories on VM1. The SSL/TLS infrastructure was configured using Easy-RSA package. The first step in building an OpenVPN configuration was to establish a PKI consisting of a separate certificate (a public key)
and a private key for the server and each of the clients, and a master Certificate Authority (CA) certificate and key, which were used to sign all the server and client certificates [25]. The list of commands used to install and configure OpenVPN is provided in listing 1.

```
apt-get install openvpn
mkdir /etc/openvpn/easy-rsa/
cp -r /usr/share/doc/openvpn/examples/easy-rsa/2.0/*
/etc/openvpn/easy-rsa/
l n -s openssl-1.0.0.cnf openssl.cnf
```

Listing 1. OpenVPN Installation.

Next, the file /etc/openvpn/easy-rsa/vars was edited to adjust the settings for the certificates. It contains such variables as KEY_COUNTRY, KEY_PROVINCE, KEY_CITY, KEY_ORG, and KEY_EMAIL. These variables reflect the CA. In the process of key generation, host names had to be entered. Ubuntu-vm1 was the host name of VM1 (the VPN server), ubuntu-vm2 was the host name VM2 (the VPN client). After that, the following commands generated the master CA certificate and key, Diffie Hellman parameters, and a certificate and private key for the server and the clients (see listing 2).

```
cd /etc/openvpn/easy-rsa/
source vars
./clean-all
./build-ca
./build-key-server ubuntu-vm1
./build-dh
./build-key ubuntu-vm2
cd keys/
cp ubuntu-vm1.crt ubuntu-vm1.key ca.crt dh1024.pem /etc/openvpn/
```

Listing 2. PKI Setup.

OpenVPN provides sample configuration files for a server and a client. The server configuration file /usr/share/doc/openvpn/examples/sample-config-files/server.conf.gz was copied to /etc/openvpn/ directory and unpacked. Then, the /etc/openvpn/server.conf file was edited to make sure that OpenVPN would use the correct certificates and keys.
OpenVPN offers a few possible topology choices. These are available options as values to the `--topology` parameter in `--dev tun` mode. By default, it uses “net30” which is the old topology supporting Windows clients running 2.0.9 or older. In this topology, each client is allocated a virtual /30 subnet, which takes four IP addresses per client and four IP addresses for the server. According to the documentation, the recommended topology for modern servers is “subnet” where addressing is done by normal subsequent assigning of the IP addresses from the dedicated subnet for all clients. In this project, the topology was changed to “subnet”.

In order to provide consistency across all VPN infrastructure including the future connections to the Google Compute Engine Cloud, the OpenVPN server was configured to listen on port TCP 443. It used subnet 172.16.63.0 255.255.255.0, from which the first address the server took for itself. VPN addressing is shown in figure 6. The OpenVPN server could also push routes to the client to allow it to reach the server’s private subnets. In this case, the OpenVPN server pushed a route to the subnet 192.168.1.0/24. After VPN connection was established, the client (VM2) could access network 192.168.1.0/24. The changes applied to the server configuration file are shown in listing 3.

```
port 443
proto tcp
server 172.16.63.0 255.255.255.0
topology subnet
push "route 192.168.1.0 255.255.255.0"
```

Listing 3. Additional configuration parameters of the OpenVPN server.
OpenVPN can assign specific IP addresses to specific clients and provide routing for private subnets behind a client. Subdirectory "ccd" is used for client-specific configuration files. Since ubuntu-vm2 client had 172.16.1.0/24 subnet behind it, a route to this network was added to server.conf (route 172.16.1.0 255.255.255.0 172.16.64.253). According to the documentation, identifying a subnet address and a subnet mask is enough, but this configuration did not work properly with subnet topology. As a solution, a gateway address was added, which brought up the necessity of a static IP address for this client. Listing 4 shows the content of the created file ccd/ubuntu-vm2 containing a route and a statically assigned IP address. This configuration required using “ubuntu-vm2” client name when generating the client key and certificate. The OpenVPN server configuration file is provided in appendix 1.

```
  iroute 192.16.1.0 255.255.255.0
  ifconfig-push 172.16.64.253 255.255.255.0
```

Listing 4. The content of the ccd/ubuntu-vm2 file.

VPN clients can connect to the server using unique certificate/key pairs or they can all use the same pair which simplifies the key/certificate generation and distribution process, but adversely impacts security. For allowing more than one client to use the same pair, "duplicate-cn" option can be uncommented in the server configuration file. If this option is not enabled and the same certificate for several clients is used, then OpenVPN will assign the same IP address to these clients when they connect.

After the described preparation, the OpenVPN server was started using "service openvpn start" command. Log messages were displayed using “cat /var/log/syslog | grep VPN” command. The commands “ifconfig tun0” or “ip address” could be applied to check whether OpenVPN had created a tun0 interface and whether it used correct IP addresses.

5.2.3 VPN Client Configuration

The same OpenVPN package was installed and configured on the client machine ubuntu-vm2. The client.conf sample configuration file was copied to /etc/openvpn/ and modified to match the server configuration. The client certificate and private key and the CA certificate were copied to the client's directory /etc/openvpn using Secure Copy
(SCP) method based on SSH protocol. Changing permissions to the client private key on the server allowed it to be copied via SCP with non-root credentials (chmod 644 ubuntu-vm2.key). The client certificates and keys were only required on the client machine and were removed from the server for better security. Listing 5 shows modifications applied to the client.conf file.

```plaintext
remote 10.94.128.3 443
proto tcp
cert ubuntu-vm2.crt
key ubuntu-vm2.key
```

Listing 5. Modifications applied to the client.conf file.

The tun0 interface was created after the OpenVPN service had been started. In order to use the Ubuntu routing capabilities on VM1 and VM2, IP forwarding was enabled by making changes to the file /etc/sysctl.conf (a line containing “net.ipv4.ip_forward = 1” was uncommented).

As a result of this stage, a VPN tunnel between two private clouds was configured. In addition, the routing options configuration provided full connectivity for all hosts on the network. The connection between the client and the server was verified using ping. The client received the 172.16.63.254 IP address and had to be able to ping the server IP address 172.16.63.1. Furthermore, as a result of enabled routing and pushing routes via VPN, VM3 from the client private network was also able to ping VM4 from the server private network.

5.3 Creating a Virtual Machine on the Google Compute Engine Cloud

The Google Compute Engine resources can be managed using the command-line tool or the Compute Engine web console. The GCE web interface provides an easy way to create VM instances. When a VM instance is created, there will be no way to stop it normally for some period of time and then start it again. If the VM operation needs to be interrupted, the snapshots will have to be used. For this project, a VM instance with
1 virtual CPU and 0.6 GB memory was created. It used 10 GB of persistent disk space and run Debian Wheezy. The web interface for creating a VM is shown in figure 7.

Figure 7. Creating a new instance in the GCE cloud.

The f1-micro machine belongs to shared-core machine types and is designed for tasks that do not require many resources but have to stay active for long periods of time. The f1-micro machine offers bursting capabilities that allow instances to use an additional physical CPU for a short time. This machine type is available for considerably less price than the more powerful machine types. Google Compute Engine provides graphical tools for monitoring the CPU load and Disk input/output for VM instances. The VM resources were sufficient for the testing. In this stage, the VM did not run any configured services. Later in the project, it would run Docker software for container virtualization, which is not demanding on resources, especially when it runs only a few simple containers. The CPU load and Disk input/output graphs are shown in figure 8.
For further configuration, gcutil tool was installed on the laptop running Ubuntu. The command “curl https://dl.google.com/dl/cloudsdk/release/install_google_cloud_sdk.bash | bash” installs gcutil. This tool runs on UNIX-based operating systems and requires to have Python 2.6.x or 2.7.x installed on the computer. Google Compute Engine uses OAuth2 to authenticate and authorize access. Before using gcutil, it was necessary to get authenticated to the Google Cloud platform by running “gcloud auth login”. The VM instance used in this project was called vpngw1. The following command, which could always be retrieved from a web interface, was used to connect to the VM: gcutil --service_version="v1" --project="thesis-ha-w2014-n01-taty" ssh --zone="europe-west1-a" "vpngw1".

It is also possible to connect to the VM instance via other regular SSH tools. For example, openssh package can be used on Linux systems. It requires to explicitly identify the SSH keys that have to be used for the connection. During the authentication process with “gcloud auth login” command, the keys are created automatically, and stored locally in the ~/.ssh directory. A public key is also copied to the metadata of the GCE project (the user must have read/write permissions to the project). For establishing successful SSH connection, SSH/SCP tools are run with -i option pointing at the existing private key. The command starts with “ssh -i .ssh/google_compute_engine” and is followed by the IP address or other parameters as normal.
By default, every project uses a default network. All incoming traffic from outside a network is blocked if no appropriate firewall rule exists. The default network has the following firewall rules: the default-allow-internal rule, which allows network connections of any protocol and port between instances on the same network, and the default-ssh rule, which allows TCP connections over port 22 from any source to any instance on the network. To allow incoming VPN traffic, a new rule was added to permit a TCP connection over port 443.

As a result of this project stage, a virtual machine instance in the Google Compute Engine Cloud was created. The available options and the specifics of VM operation in the Google Cloud were explored. The firewall was configured to allow incoming connections over TCP port 443 to support the incoming VPN connection.

5.4 Docker Setup and Configuration

According to the project plan, Docker had to be installed on VM4 in the private cloud and on vpngw1 in the Google cloud. The commands used to install Docker on VM4 running Ubuntu 12.04 are shown in listing 6.

```bash
sudo apt-key adv --keyserver keyserver.ubuntu.com -recv-keys \ 36A1D7869245C8950F966E92D8576A8BA88D21E9
sudo sh -c "echo deb http://get.docker.io/ubuntu docker main \ > /etc/apt/sources.list.d/docker.list"
sudo apt-get update
sudo apt-get install lxc-docker
```


IP forwarding was turned on on VM4 to enable routing between the Docker subnet and other networks. According to the documentation, the IP addresses of the containers are local to the host system and the container ports are not reachable by the outside world. However, in practice the containers could be accessed from outside through their IP addresses and ports. The challenge is that the IP address of the container may change every time it starts.
In order to run Docker on multiple hosts and let all the containers to communicate with each other, the consistent addressing and routing configuration was needed. Docker created a bridge interface and the IP addresses from a certain default subnet were assigned to all created containers. To assign a custom IP subnet to the Docker containers, it was necessary to create a new bridge interface and configure the Docker service to use this interface. Listing 7 shows the commands used to create and configure a new bridge on VM4.

```
service docker stop
ifconfig docker0 down
ip link add bridge0 type bridge
ip addr add 172.16.65.129/25 dev bridge0
ip link set bridge0 up
```

Listing 7. Creating a bridge interface.

After creating a bridge interface, the Docker startup configuration file had to be modified in order to use a new bridge. It was done using the following command: `echo "DOCKER_OPTS="-b=bridge0:"
" >> /etc/default/docker`. Next, the Docker service was started using the command “service docker start”.

Docker can read instructions from a text file called Dockerfile to automate the steps of creating an image. For testing purposes, a simple container running SSH daemon service was set up. The Dockerfile content is listed in Appendix 2. The file with the name Dockerfile was located in the home directory of the current user. The new image named ubuntu-ssh was built using this file. Then, the container named test1 was run using the ubuntu-ssh image. The container ID could be found out using the “docker ps” command, while the “docker inspect” command returned the container IP address which could be used for SSH connection. The commands used to run the container and examine its IP settings are shown in listing 8.

```
docker build -rm -t ubuntu-ssh .
docker run -d -t -P --name test1 ubuntu-ssh
docker ps
docker inspect <container_id> | grep IP
```

Listing 8. Running a container.
The -d option enables detached mode. It means that the container runs in the background and is not listening to the command line commands. Also, all input/output operations have to be done via the network. The -t option allocates a pseudo-tty. It is needed to enable the command line interface over SSH. The -P option publishes all exposed to the host interfaces container ports. For example, the container TCP port 22 can be exposed as port 49155 on the host. The --name option allows to set a name for a container. If a name is not set, it will be generated automatically.

The first created container got an IP address of 172.16.65.130. To make the container accessible from outside, the route to the Docker subnet was added on VM1. VM1 was a gateway for VM4 connecting it with outside world. The following command added the route: ip route add 172.16.65.128/25 via 192.168.1.2. It allowed VM1 to route packets destined for subnet 172.16.65.128/25, used by Docker.

The Docker configuration on the GCE cloud is similar to the configuration for the private cloud. The latest Docker version was installed and configured using the commands “curl get.docker.io | bash” and “update-rc.d docker defaults”. Two containers running the SSH daemon service were created using Dockerfile. The output of the “docker ps” command, which returns information about running containers, is shown in figure 9.

![Figure 9. List of the running Docker containers.](image)

Docker running on the VM in the Google Cloud used the 172.16.65.0/25 network. To allow communication between the Docker containers running in the private and in the public clouds, a VPN tunnel between clouds had to be set up.

5.5 Hybrid Cloud Configuration with a Full Mesh VPN Topology

A hybrid cloud had to connect the private clouds and the GCE cloud with full mesh VPN topology. In order to implement this scheme, the OpenVPN server was configured on the VM in the GCE cloud and on each of the private cloud gateways. The connec-
tions were set up in a similar way with the connection between the private cloud described in subsection 5.2. VM1 and VM2 operated as VPN servers and VPN clients at the same time. The network diagram is shown in figure 10.

![Figure 10. The hybrid cloud network diagram.](image)

Each OpenVPN server runs its own Certification Authority. To simplify the VPN infrastructure and to allow a client to connect to any of the three VPN servers using the same certificate retrieved from one of the servers, the CA certificates of each server were stacked into one file and distributed across the servers. The command “cat ca1.crt ca2.crt ca3.crt > stacked.crt” combined the three CA certificates into one file. Next, each of three servers was configured to use the stacked.crt file to check the validity of the client certificate.

All the necessary routes were pushed across VPN links. For example, the VM hosted in the Google Cloud pushed to its clients the routes to the Docker subnet 172.16.65.0/25. With help of OpenVPN routing options, this VM was also configured to route all traffic destined for subnet 172.16.1.0/24 to VM2 and traffic for subnets
192.168.1.0/24 and 172.16.65.128/25 to VM1. The VPN servers VM1 and VM2 were configured to handle routing in the similar way.

Thus, all the hosts including the Docker containers could communicate with each other. Figure 11 shows an example of SSH connection to one of the Docker containers running in the GCE cloud. This connection was initiated from a laptop connected via VPN over the Internet to the VM in the Google Cloud.

```
studenti@Tedge2:~$ ssh root@172.16.65.3
root@172.16.65.3's password:
Welcome to Ubuntu 12.04 LTS (GNU/Linux 3.2.0-4-amd64 x86_64)
* Documentation: https://help.ubuntu.com/
Last login: Mon Apr  7 13:36:19 2014 from 172.16.64.2
root@35747fbadc7e:~# ls
```

Figure 11. SSH connection to the Docker container.

By means of Ubuntu routing capabilities and configuring the OpenVPN routing options, connectivity was achieved between all the network elements including hypervisor-based virtual machines and Docker containers. Containers were run on top of the standard virtual machines and provided benefits in performance and scalability.

6 Results and Discussion

The project was divided into four stages, and all of them were completed. As the starting point of the project, two private clouds were interconnected with an OpenVPN tunnel. The Google Compute Engine Cloud setup followed the private clouds VPN configuration. Docker was installed, configured and tested on the virtual machines in the public and one of the private clouds. A few simple containers running SSH service were created for testing purposes.

The consistent addressing scheme allowed to configure routing for the whole test network. The subnets 172.16.65.0/25 and 172.16.65.128/25 were used for Docker containers on vpngw1 and VM4 respectively. The subnets 172.16.62.0/24, 172.16.63.0/24, 172.16.64.0/24 represented VPN connections. As a result of OpenVPN routing configuration on VPN servers and manual configuration for subnet 172.16.65.128/25 on VM1,
all hosts in the hybrid cloud network could communicate with each other. In particular, it was important to confirm that the containers running in the Google cloud were able to communicate with the containers running in the private cloud on VM4.

For security and confidentiality reasons, the communication between two or more clients of the same VPN server was blocked. This is a default OpenVPN policy. If the interaction between clients is necessary, it can be configured through the server.conf text file by uncommenting the line “client-to-client”. This configuration will also require the additional routing settings because VPN servers will need to push more routes to their clients.

The result of this project is a hybrid cloud test environment combining the internal resources at Metropolia premises and the Google Compute Engine cloud. The infrastructure components are connected together as a full mesh VPN topology. Three OpenVPN servers have been deployed. The server located in the Google Compute Engine cloud has a public IP address and accepts VPN connections from anywhere on the Internet, given that a client provides a valid key and certificate. The servers at the edge of private clouds accept connections from the clients located in the Metropolia Labnet network. The network diagram is shown in figure 12.
Due to time limitations, the compatibility of the OpenVPN server with other VPN clients, such as built-in GUI VPN clients for Ubuntu, Chrome OS, Mac OS or Windows, was not tested. It is known that the OpenVPN server does not support a web browser as a VPN client. OpenVPN clients can run on all widely used modern operating systems. The key pair has to be generated and copied to the client host before it can connect to the server. The distribution of keys was also beyond the scope of this project. Theoretically, the key distribution process for OpenVPN can be simplified using scripts or third-party software.

The VPN configuration uses TCP port 443 in order to bypass a firewall. The TCP connection is normally more reliable, but it may cause problems in the case of VPN. Usually UDP is recommended for OpenVPN connections because it is faster and does not cause excessive overhead. It is important especially when a VPN tunnel carries VoIP traffic or other data sensitive to delay. However, the VPN performance issued were not tested in this project.
Docker containers represent a convenient way to develop, deploy and distribute applications. Containers can be started, stopped, modified and saved to another image. They can be accessed from outside through their IP addresses as regular computers, although their IP addresses may change every time they start, which makes their usage inconvenient for some purposes. Docker is still under development and not recommended for use in production, but it offers promising opportunities for application deployment and building custom lightweight PaaS environments. The application development and deployment using Docker containers was beyond the scope of this project.

In the created cloud environment, new Docker containers can be run in either private or public infrastructure, which provides flexibility. The deployment of containers across the clouds can be automated by employing additional tools, such as OpenStack Nova. The combination of Docker and OpenStack provides a powerful tool for application development and deployment in a cross-cloud environment.

7 Conclusion

The goal of the project was to set up a hybrid cloud infrastructure where cloud bursting could be potentially implemented in the future using lightweight container-based virtualization as an approach to flexible application and system deployment. A hybrid cloud as the most flexible and resilient cloud model, was of particular interest to this project. The main purpose of hybrid cloud testing was to consider the possibilities of creating a new flexible and cost-efficient cloud infrastructure for educational purposes.

In the process of project work the main features, benefits and challenges of cloud computing and container-based virtualization were investigated and discussed. The project result was a hybrid cloud infrastructure comprising internal Metropolia resources and a Google Compute Engine cloud. This hybrid cloud was implemented as a proof of concept testing. The important result of the project was building the container-based virtualization on the virtual machines hosted in the different clouds. The project confirmed that as long as all components are installed and properly configured, a hybrid cloud
setup with OpenVPN and Docker will represent the facilitative environment for implementing cloud bursting. Cloud bursting, in its turn, could optimize the resource usage.

The project gave me a better understanding of one of the most popular technologies nowadays, cloud computing. A hybrid cloud is one the most interesting cloud models, and by configuring it, not only did I learn more about cloud implementations, but also gained practical experience of VPN configuration. As another project outcome, I became familiar with Docker container-based virtualization which is nowadays considered one of the new promising technologies in application deployment. I expect that the knowledge and practical experience that I gained while working on this project can benefit my career.

The project did not involve the application development or deployment of any specific application in the hybrid cloud. It also did not cover the automation of Docker containers’ deployment on multiple hosts. In order to be able to give more accurate estimations on the feasibility of a hybrid cloud infrastructure and container-based virtualization, it would be useful to study how the containers’ deployment across the clouds could be done automatically with the help of additional software. In particular, it looks promising to explore and test Docker and OpenStack functionality in combination. This investigation would provide a better understanding of whether the deployed model of a hybrid cloud is worth using as the cloud infrastructure in Helsinki Metropolia University of Applied Sciences.
References


Server.conf Configuration File

# Which local IP address should OpenVPN
# listen on? (optional)
;local a.b.c.d

# Which TCP/UDP port should OpenVPN listen on?
# If you want to run multiple OpenVPN instances
# on the same machine, use a different port
# number for each one. You will need to
# open up this port on your firewall.
port 443

# TCP or UDP server?
proto tcp
;proto udp

# "dev tun" will create a routed IP tunnel,
# "dev tap" will create an ethernet tunnel.
# Use "dev tap0" if you are ethernet bridging
# and have precreated a tap0 virtual interface
# and bridged it with your ethernet interface.
# If you want to control access policies
# over the VPN, you must create firewall
# rules for the the TUN/TAP interface.
# On non-Windows systems, you can give
# an explicit unit number, such as tun0.
# On Windows, use "dev-node" for this.
# On most systems, the VPN will not function
# unless you partially or fully disable
# the firewall for the TUN/TAP interface.
;dev tap
dev tun

# Windows needs the TAP-Win32 adapter name
# from the Network Connections panel if you  
# have more than one. On XP SP2 or higher,  
# you may need to selectively disable the  
# Windows firewall for the TAP adapter.  
# Non-Windows systems usually don't need this.
:dev-node MyTap

# SSL/TLS root certificate (ca), certificate  
# (cert), and private key (key). Each client  
# and the server must have their own cert and  
# key file. The server and all clients will  
# use the same ca file.

# See the "easy-rsa" directory for a series  
# of scripts for generating RSA certificates  
# and private keys. Remember to use  
# a unique Common Name for the server  
# and each of the client certificates.

# Any X509 key management system can be used.  
# OpenVPN can also use a PKCS #12 formatted key file  
# (see "pkcs12" directive in man page).
ca ca_stacked.crt
 certify ubuntu-vm1_server.crt
key ubuntu-vm1_server.key  
# This file should be kept secret

# Diffie hellman parameters.
# Generate your own with:
# openssl dhparam -out dh1024.pem 1024  
# Substitute 2048 for 1024 if you are using  
# 2048 bit keys.
dh dh1024.pem

# Configure server mode and supply a VPN subnet  
# for OpenVPN to draw client addresses from.
# The server will take 10.8.0.1 for itself,
# the rest will be made available to clients.
# Each client will be able to reach the server
# on 10.8.0.1. Comment this line out if you are
# ethernet bridging. See the man page for more info.
server 172.16.63.0 255.255.255.0
topology subnet

# Maintain a record of client <-> virtual IP address
# associations in this file. If OpenVPN goes down or
# is restarted, reconnecting clients can be assigned
# the same virtual IP address from the pool that was
# previously assigned.
ifconfig-pool-persist ipp.txt

# Configure server mode for ethernet bridging.
# You must first use your OS’s bridging capability
# to bridge the TAP interface with the ethernet
# NIC interface. Then you must manually set the
# IP/netmask on the bridge interface, here we
# assume 10.8.0.4/255.255.255.0. Finally we
# must set aside an IP range in this subnet
# (start=10.8.0.50 end=10.8.0.100) to allocate
# to connecting clients. Leave this line commented
# out unless you are ethernet bridging.
;server-bridge 10.8.0.4 255.255.255.0 10.8.0.50 10.8.0.100

# Configure server mode for ethernet bridging
# using a DHCP-proxy, where clients talk
# to the OpenVPN server-side DHCP server
# to receive their IP address allocation
# and DNS server addresses. You must first use
# your OS’s bridging capability to bridge the TAP
# interface with the ethernet NIC interface.
# Note: this mode only works on clients (such as
# Windows), where the client-side TAP adapter is
# bound to a DHCP client.

:server-bridge

# Push routes to the client to allow it
# to reach other private subnets behind
# the server. Remember that these
# private subnets will also need
# to know to route the OpenVPN client
# address pool (10.8.0.0/255.255.255.0)
# back to the OpenVPN server.
push "route 192.168.1.0 255.255.255.0"
push "route 172.16.65.128 255.255.255.128"

# To assign specific IP addresses to specific
# clients or if a connecting client has a private
# subnet behind it that should also have VPN access,
# use the subdirectory "ccd" for client-specific
# configuration files (see man page for more info).

# EXAMPLE: Suppose the client
# having the certificate common name "Thelonious"
# also has a small subnet behind his connecting
# machine, such as 192.168.40.128/255.255.255.248.
# First, uncomment out these lines:
client-config-dir ccd
route 172.16.1.0 255.255.255.0 172.16.63.254
# Then create a file ccd/Thelonious with this line:
# iroute 192.168.40.128 255.255.255.248
# This will allow Thelonious' private subnet to
# access the VPN. This example will only work
# if you are routing, not bridging, i.e. you are
# using "dev tun" and "server" directives.
# EXAMPLE: Suppose you want to give
# Thelonious a fixed VPN IP address of 10.9.0.1.
# First uncomment out these lines:
;client-config-dir ccd
;route 10.9.0.0 255.255.255.252
# Then add this line to ccd/Thelonious:
#   ifconfig-push 10.9.0.1 10.9.0.2

# Suppose that you want to enable different
# firewall access policies for different groups
# of clients. There are two methods:
# (1) Run multiple OpenVPN daemons, one for each
#     group, and firewall the TUN/TAP interface
#     for each group/daemon appropriately.
# (2) (Advanced) Create a script to dynamically
#     modify the firewall in response to access
#     from different clients. See man
#     page for more info on learn-address script.
;learn-address ./script

# If enabled, this directive will configure
# all clients to redirect their default
# network gateway through the VPN, causing
# all IP traffic such as web browsing and
# and DNS lookups to go through the VPN
# (The OpenVPN server machine may need to NAT
# or bridge the TUN/TAP interface to the internet
# in order for this to work properly).
;push "redirect-gateway def1 bypass-dhcp"

# Certain Windows-specific network settings
# can be pushed to clients, such as DNS
# or WINS server addresses. CAVEAT:
# http://openvpn.net/faq.html#dhcpcaveats
# The addresses below refer to the public
# DNS servers provided by opendns.com.
;push "dhcp-option DNS 208.67.222.222"

# Uncomment this directive to allow different
# clients to be able to "see" each other.
# By default, clients will only see the server.
# To force clients to only see the server, you
# will also need to appropriately firewall the
# server's TUN/TAP interface.
;client-to-client

# Uncomment this directive if multiple clients
# might connect with the same certificate/key
# files or common names. This is recommended
# only for testing purposes. For production use,
# each client should have its own certificate/key
# pair.
#
# IF YOU HAVE NOT GENERATED INDIVIDUAL
# CERTIFICATE/KEY PAIRS FOR EACH CLIENT,
# EACH HAVING ITS OWN UNIQUE "COMMON NAME",
# UNCOMMENT THIS LINE OUT.
duplicate-cn

# The keepalive directive causes ping-like
# messages to be sent back and forth over
# the link so that each side knows when
# the other side has gone down.
# Ping every 10 seconds, assume that remote
# peer is down if no ping received during
# a 120 second time period.
keepalive 10 120

# For extra security beyond that provided
# by SSL/TLS, create an "HMAC firewall"
# to help block DoS attacks and UDP port flooding.
#
# Generate with:
# openvpn --genkey --secret ta.key
#
# The server and each client must have
# a copy of this key.
# The second parameter should be '0'
# on the server and '1' on the clients.
;tls-auth ta.key 0 # This file is secret

# Select a cryptographic cipher.
# This config item must be copied to
# the client config file as well.
;cipher BF-CBC # Blowfish (default)
;cipher AES-128-CBC # AES
;cipher DES-EDE3-CBC # Triple-DES

# Enable compression on the VPN link.
# If you enable it here, you must also
# enable it in the client config file.
comp-lzo

# The maximum number of concurrently connected
# clients we want to allow.
;max-clients 100

# It's a good idea to reduce the OpenVPN
# daemon's privileges after initialization.
#
# You can uncomment this out on
# non-Windows systems.
;user nobody
;group nogroup
# The persist options will try to avoid
# accessing certain resources on restart
# that may no longer be accessible because
# of the privilege downgrade.
persist-key
persist-tun

# Output a short status file showing
# current connections, truncated
# and rewritten every minute.
status openvpn-status.log

# By default, log messages will go to the syslog (or
# on Windows, if running as a service, they will go to
# the "\Program Files\OpenVPN\log" directory).
# Use log or log-append to override this default.
# "log" will truncate the log file on OpenVPN startup,
# while "log-append" will append to it. Use one
# or the other (but not both).
;log openvpn.log
;log-append openvpn.log

# Set the appropriate level of log
# file verbosity.
#
# 0 is silent, except for fatal errors
# 4 is reasonable for general usage
# 5 and 6 can help to debug connection problems
# 9 is extremely verbose
verb 3

# Silence repeating messages. At most 20
# sequential messages of the same message
# category will be output to the log.
;mute
Dockerfile for Running SSH Service
Modified from Docker Homepage [26].

# ssdh
#
# VERSION               0.0.1

FROM ubuntu:12.04
MAINTAINER Thatcher R. Peskens "thatcher@dotcloud.com"

# make sure the package repository is up to date
RUN echo "deb http://archive.ubuntu.com/ubuntu precise main universe" > /etc/apt/sources.list
RUN apt-get update

RUN apt-get install -y openssh-server
RUN mkdir /var/run/sshd
RUN echo 'root:scre1234' | chpasswd

EXPOSE 22
CMD /usr/sbin/sshd -D