TAMPERE UNIVERSITY OF TECHNOLOGY

VILLE SEPPÄNEN

ELASTIC BUILD SYSTEM IN A HYBRID CLOUD ENVIRONMENT

Master of Science Thesis

Examiners: professor Jarmo Harju
and professor Tommi Mikkonen
Examiners and topic approved in
Faculty meeting on 12 January 2011

ABSTRACT

TAMPERE UNIVERSITY OF TECHNOLOGY
Master's Degree Programme in
Signal Processing and Communications Engineering
SEPPÄNEN, VILLE: Elastic Build System in a Hybrid Cloud Environment
Master of Science Thesis, 49 pages, 3 Appendix pages
November 2011
Major: Communications Networks and Protocols
Examiners: Professor Jarmo Harju, Professor Tommi Mikkonen
Keywords: Cloud computing, computing cluster, distributed system, software build, OBS, cloudbursting, build system, virtualization

Linux-based operating systems such as MeeGo consist of thousands of modular packages. Compiling source code and packaging software is an automated but computationally heavy task. Fast and cost-efficient software building is one of the requirements for rapid software development and testing. Meanwhile, the arrival of cloud services makes it easier to buy computing infrastructure and platforms over the Internet. The difference to earlier hosting services is the agility; services are accessible within minutes from the request and the customer only pays per use.

This thesis examines how cloud services could be leveraged to ensure sufficient computing capacity for a software build system. The chosen system is Open Build Service, a centrally managed distributed build system capable of building packages for MeeGo among other distributions. As the load on a build cluster can vary greatly, a local infrastructure is difficult to provision efficiently, thus virtual machines from the cloud could be acquired temporarily to accommodate the fluctuating demand. Main issues are whether cloud could be utilized safely and whether it is time-efficient to transfer computational jobs to an outside service.

A MeeGo-enabled instance of Open Build Service was first set up in-house, running a management server and a server for workers which build the packages. A virtual machine template for cloud workers was created. Virtual machines created from this template would start the worker program and connect to the management server through a secured tunnel. A service manager script was then implemented to monitor jobs and the usage of workers and to make decisions whether new machines from the cloud should be requested or idle ones terminated. This elasticity is automated and is capable of scaling up in a matter of minutes. The service manager also features cost optimizations implemented with a specific cloud service (Amazon Web Services) in mind.

The latency between the in-house and the cloud did not prove to be insurmountable, but as each virtual machine from the cloud has a starting delay of three minutes, the system reacts fairly slowly to increasing demand. The main advantage of the cloud usage is the seemingly infinite number of machines available, ideal for building a large number of packages that can be built in parallel. Packages may need other packages during building, which inhibits the system from building all packages in parallel. Powerful workers are needed to quickly build larger bottleneck packages.

Finding the balance between the number and performance of workers is one of the issues for future research. To ensure high availability, improvements should be made to the service manager and a separate virtual infrastructure manager should be used to utilize multiple cloud providers. In addition, mechanisms are needed to keep proprietary source code on in-house workers and to ensure that malicious code cannot be injected into the system via packages originating from open development communities.

TIIVISTELMÄ

TAMPEREEN TEKNILLINEN YLIOPISTO
Signaalinkäsittelyn ja tietoliikennetekniikan koulutusohjelma
SEPPÄNEN, VILLE: Elastic Build System in a Hybrid Cloud Environment
Diplomityö, 49 sivua, 3 liitesivua
Marraskuu 2011
Pääaine: Tietoliikenneverkot ja protokollat
Tarkastajat: professori Jarmo Harju, professori Tommi Mikkonen
Avainsanat: pilvipalvelu, pilvilaskenta, hajautettu järjestelmä, ohjelmistokehitys, OBS, ohjelmistopaketointi, virtualisointi

Linux-pohjaiset käyttöjärjestelmät kuten MeeGo koostuvat tuhansista modulaarisista ohjelmistopaketeista. Lähdekoodin kääntäminen ja paketointi ovat automaattisia, mutta laskennallisesti raskaita tehtäviä. Ohjelmistojen nopea ja kustannustehokas rakentaminen (*software building*) on edellytys nopealle kehitys- ja testaustyölle, ja siten myös ohjelmistoyrityksen kilpailukyvylle. Samalla pilvipalveluiden yleistyminen mahdollistaa erilaisten infrastruktuurien ja ohjelmistoalustojen helpon ostamisen tilapäiseen käyttöön Internetin yli. Erona aiempiin vuokrauspalveluihin on ketteryys; palvelut ovat käytettävissä muutaman minuutin varoitusajalla ja asiakas maksaa vain käytöstä.

Tässä diplomityössä tutkitaan, miten pilvipalveluita voitaisiin hyödyntää ohjelmistojen rakennusjärjestelmän kapasiteetin varmistamiseksi. Käytettävä järjestelmä on Open Build Service, keskitetysti hallittu hajautettu paketointijärjestelmä, joka kykenee rakentamaan paketteja muun muassa MeeGolle. Palvelun hetkellinen kuormitus voi vaihdella suuresti, jolloin paikallisen infrastruktuurin kapasiteettia on vaikea mitoittaa etukäteen. Tällöin väliaikaista apua voitaisiin vuokrata pilvipalveluista virtuaalikoneina. Tutkimuksessa selvitetään erityisesti, onko pilvipalvelua mahdollista hyödyntää turvallisesti, ja onko laskennan siirtäminen ulkopuoliseen palveluun kustannustehokasta.

Työssä pystytettiin Open Build Service käyttäen kahta yrityksen sisäistä palvelinta: yksi hallinnoiva palvelin ja yksi palvelin paikallisille rakentajille. Pilvessä rakentamista varten tehtiin virtuaalikoneen mallipohja, josta luotavat virtuaalikoneet käynnistävät rakentajaohjelman ja ottavat yhteyttä hallinnoivaan palvelimeen salatun tunnelin läpi. Työssä kehitettiin skripti, joka valvoo resurssien käyttöä ja tarvetta, ja tekee tämän pohjalta päätöksiä lisäkoneiden käynnistämisestä tai joutilaiden koneiden sammuttamisesta pilvipalvelussa. Järjestelmä mukautuu kuormaan automaattisesti ja lisäkapasiteetti on käytettävissä muutamassa minuutissa. Skriptiin on toteutettu palveluntarjoajan (Amazon Web Services) hinnoitteluun liittyviä optimointeja.

Viive paikallisen klusterin ja pilven välillä ei koitunut ylitsepääsemättömästi. Jokaisen pilvirakentajan käynnistys vie kuitenkin kolme minuuttia, joten järjestelmä reagoi hitaasti pieniin työmäärän muutoksiin. Tärkein etu pilvipalvelun käytöstä on näennäisen loputon määrä koneita, jolloin suuri määrä rinnakkain rakennettavia paketteja ei ruuhkauta palvelua. Paketit kuitenkin usein riippuvat toisista paketeista estäen rinnakkaisen rakentamisen. Näitä pullonkauloja varten on pystytettävä tehokkaampia rakentajia.

Jatkotutkimuksissa tulisi selvittää optimi rakentajien määrän ja tehokkuuden välillä rakennusaikojen lyhentämiseksi. Palvelun tuotantokäyttötasoisen saatavuuden varmistamiseksi tarvitaan tuki useiden pilvipalveluiden hyödyntämiseksi. Tämä tuki vaatii infrastruktuurinhallintaohjelmistoja ja parannuksia skriptin päätöksentekoon. Lisäksi järjestelmään tarvitsee kehittää mekanismeja suljetun koodin paketoimisen rajoittamiseksi yrityksen sisälle sekä mahdollisten avoimen koodin rakentamisen kautta tulevien hyökkäysten torjumiseksi.

PREFACE

This thesis was done to Tieto Corporation and was supported by TEKES as part of the Cloud Software program of TIVIT (Finnish Strategic Centre for Science, Technology and Innovation in the field of ICT). The target audience for this thesis is people who are interested in utilizing cloud services or in setting up their own instance of Open Build Service (OBS). Cloud management software issues have been discussed with peers in the Cloud Software program.

Open Build Service aspects have been discussed with and reviewed by the OBS community, and I would like to thank Sami Anttila and the rest of the OBS community for providing valuable feedback. The service manager implemented in this thesis has been released under the GNU General Public License in the source code repository of the build service. MeeGo-related parts have been discussed with the MeeGo community. I would like to dedicate special thanks to Thomas Rücker for his insight into both MeeGo and OBS related aspects.

I want to thank my professors Jarmo Harju and Tommi Mikkonen as well as my instructor Jussi Nurminen for guiding me during my work. Last but not least, I thank my spouse Tanja for encouraging me during brief times of despair. Overall I enjoyed making my thesis of a very current topic: a distributed computational system that spans over several stakeholders.

> "*The Open Build Service makes package building a community effort.*
> *That means fun, fun, fun and divided pain ;-)"*
> - Thomas Schmidt in the Open Build Service FAQ

Tampere, October 17th 2011

_____

Ville Seppänen

TABLE OF CONTENTS

1 Introduction 1
2 Cloud services 3
2.1 About the cloud in general 3
2.1.1 Essence of cloud computing 3
2.1.2 Infrastructure as a service 4
2.1.3 Elasticity – shifting the risks of provisioning 5
2.1.4 Challenges in the cloud 6
2.2 Hybrid cloud and cloudbursting 8
2.3 Managing cloud services 9
3 Building software packages 13
3.1 Software building 13
3.2 Open Build Service 14
3.3 MeeGo and its build services 17
4 Extending MeeGo building to the cloud 20
4.1 Planning the overall architecture 20
4.2 Infrastructure and build service setup 22
4.2.1 Local build host and build server 22
4.2.2 Cloud build hosts 23
4.3 Implementing a service manager 25
4.4 Example of the complete system 31
5 Evaluation 33
5.1 Build measurements and analysis 33
5.2 Cloud-readiness of Open Build Service 37
5.3 Suitability of Amazon Web Service for building 38
5.4 Future work 39
5.4.1 Improving system robustness 39
5.4.2 Improving build efficiency 39
5.4.3 Ensuring system security 41
5.4.4 Easing disk image configuration 42
5.5 Related work 42
6 Conclusions 43
References 45
Appendix 1: AWS API example 50
Appendix 2: OBS API example 51
Appendix 3: List of built packages 52

ABBREVIATIONS AND TERMS

AMI	Amazon Machine Image - a VM image used in EC2.
API	Application Programming Interface.
Appliance	A preconfigured combination of an application and OS.
AWS	Amazon Web Services, IaaS hosting services provided by Amazon.
Build host	A network host that runs one or more workers for building.
EBS	Elastic Block Store, part of AWS.
EC2	Elastic Computing Cloud, part of AWS.
CBH	Cloud Build Host, a VM instance in the cloud running OBS worker(s), see also LBH.
CLI	Command Line Interface.
Cloudbursting	Extending a local cluster to the cloud dynamically when necessary.
Cluster	Group of nodes that are managed remotely and work towards a common goal.
CPU	Central Processing Unit.
DHCP	Dynamic Host Configuration Protocol.
GPL	GNU General Public License, a license for free software.
Guest OS	An OS running inside a VM.
Host OS	An OS running directly on physical hardware in contrast to running in a VM. Not to be confused with “Network host”.
HTTP	Hypertext Transfer Protocol.
HTTPS	HTTP Secure.
Hybrid Cloud	A combination of local/internal and external resources.
Hypervisor	Virtual machine manager, can run as a software on host OS or as a lightweight host OS itself.
IaaS	Infrastructure-as-a-Service.
Instance	A VM in EC2.
I/O	Input/Output.
IP	Internet Protocol.
ISP	Internet Service Provider.
KVM	Kernel-based Virtual Machine.
LBH	Local Build Host, a build host running in-house of an organization, in contrast to running in the cloud (see CBH).
Libcloud	A library to unify an interface to cloud providers and VIMs.
MeeGo	A Linux-based open-source OS for mobile devices.
Network host	Any virtual or non-virtual computer that is connected to a network and has been assigned with a host IP address. Not to be confused with Host OS.
NFS	Network File System.

Node	A single machine (physical or virtualized) that is part of a cluster.
OBS	Open Build Service, a distributed software build system.
OS	Operating System.
OSC	The command line client of OBS.
OVF	Open Virtualization Format.
PaaS	Platform-as-a-Service.
Power host	A build host in OBS that has more computing power than regular build hosts.
Power package	A large software package that many other packages depend on; should be built on a power host.
QoS	Quality-of-Service.
RAID	Redundant Array of Independent Disks.
RAM	Random-Access Memory.
REST	Representational State Transfer.
RPM	RPM Package Manager, a software package management system.
Runtime disk image	A disk volume of a specific VM, originating from a template disk image.
S3	Simple Storage Service, part of AWS.
SaaS	Software-as-a-Service.
SLA	Service Level Agreement.
SLP	Service Location Protocol.
SOAP	Simple Object Access Protocol.
Spec file	A recipe file that describes how a package should be built.
Tag	EC2 mechanism that allows the user to attach arbitrary property-value pairs to instances.
Template disk image	A disk image that will serve as a base for multiple VMs.
URL	Uniform Resource Locator.
User Data	EC2 mechanism that allows the user to pass arbitrary shell commands or other data to new instances via the EC2 API.
VIM	Virtual Infrastructure Manager, software that manages VM clusters.
VM	Virtual Machine.
VPC	Virtual Private Cloud.
VPN	Virtual Private Network.
Worker	Component of OBS, a program that builds software packages.
XML	Extensible Markup Language.

1 INTRODUCTION

Operating systems have grown in complexity over the last ten years. Modern Linux-based operating systems consist of thousands of modular software packages. Rapid development requires that software builds are quickly testable or usable. Software packages are created with automated build tools which require a lot of computational power. Furthermore, the building needs of software developers may vary greatly, causing irregular load spikes on the build system. Overprovisioning the build cluster permanently to handle even the largest temporary spikes would be costly, while underprovisioning would lengthen build times during load spikes.

Cloud services, which provide software or virtual infrastructure as a service, have emerged, offering easily obtainable, utility-like computing power over the Internet. Users can request resources from the cloud as necessary and pay based on the actual usage. Cloud services can be used as an extension to locally hosted infrastructure, even dynamically based on the utilization level, termed hybrid clouds and cloudbursting respectively.

MeeGo is an open-source operating system designed for mobile devices, televisions and in-vehicle systems. The future needs of the MeeGo software build process should be studied proactively. Ignoring the risks in provisioning may lead to longer development time and thus longer time-to-market. Buying redundant server hardware is expensive and increases maintenance need, hence other options must be studied.

This thesis examines how cloud services could be leveraged in conjunction with in-house resources to ensure sufficient computing capacity for a software build system. The chosen system is Open Build Service, a centrally managed distributed build system capable of building packages for MeeGo among other distributions. Packages could be built on virtual machines acquired temporarily from the cloud to accommodate the fluctuating demand. Main concerns are whether it is technically feasible, whether cloud could be utilized safely enough and whether it is time-efficient to transfer data to an outside service.

The concrete objective is to deploy an auto-scaling, cloud-utilizing Open Build Service for building MeeGo packages as a proof-of-concept. Emphasis is on practical setup and research of the challenges that lie in cloudbursting in the build system context. As a solid foundation for the work, several topics must be studied: theory of cloud services, managing groups of virtual machines, software building and the inner structure of Open Build Service.

The structure of this thesis is as follows. In Chapter 2 we will familiarize ourselves with the idea and basic terminology of cloud computing, its benefits and challenges.

This includes different types of cloud services and management software needed to use them. Chapter 3 explains software building and introduces the Open Build Service that will be extended on top of a cloud. The core part of this work lies in Chapter 4, where we will take a look at the design and implementation of the auto-scaling build system, representing the practical part of this thesis. In Chapter 5, we will evaluate the implemented system and the services and software used. We will also identify some challenges with suggestions on how to overcome them as well as present some related work. Finally conclusions are presented in Chapter 6.

2 CLOUD SERVICES

In this chapter we will take a look at cloud services and what kind of resources they provide. We will study the benefits and challenges with the services and introduce a provider relevant to this thesis. We will take a look at extending a private enterprise cluster to public cloud services, especially in a rapid on-demand manner. Finally the management side of the services is taken into account, as some of the management software is implemented later in this thesis.

2.1 About the cloud in general

The idea of cloud computing has been around for decades, but it has emerged in full-scale only recently. In cloud computing, information technology is provided as a utility-like service by the cloud – a set of Internet-accessible, shared and virtualized resources such as hardware, software and networking. Not only meaning technical changes, cloud computing refers to a business model where computing is outsourced on demand. A utility-like service means that the customer can consume it dynamically and it is invoiced with a pay-as-you-go model, comparable to traditional utilities like heating, water and electricity. [1, p. 2-7]

2.1.1 Essence of cloud computing

What is new in cloud computing compared to traditional computation is that to the user, the service seems to have infinite capacity available on-demand. The user spends as much as he wants, pays for what he uses and there is no up-front commitment, even on short-term usage. This is ideal for startups or risky projects as it is easy to start small and increase as needed, without making long-term plans or major investments [1, p. 4-5]. With a large number of customers and a varying amount of usage, the utility is mostly self-service with customer support only for premium customers, with basic customers asking help from a discussion forum. Customers are usually provided with a web-based user interface as well as an API for self-managing the service.

Key enablers for the advent of scalable cloud services are extremely large-scale datacenters and hardware virtualization. Large companies that had slowly accumulated massive datacenters could now leverage their existing investment and make money with economies of scale [1, p. 5-6]. This accumulation of servers enabled the companies to lower the per-server costs of datacenter infrastructure. On the other hand, virtualization allows creating huge pools of resources, where services can be migrated on-the-fly from

one hardware set to another. This hardens software services against simple hardware failures as virtualized services can be moved elsewhere during maintenance.

Cloud services are categorized based on the level of service they provide, commonly into three layers: Infrastructure-as-a-Service (IaaS), Platform-as-a-service (PaaS) and Software-as-a-Service (SaaS). These layers can be stacked as shown in Figure 2.1, so that SaaS can be deployed on top of IaaS or PaaS, or even on top of multiple other SaaS in case of a mashup service [1, p. 4-5]. Each provider is able to make profit through economies of scale.

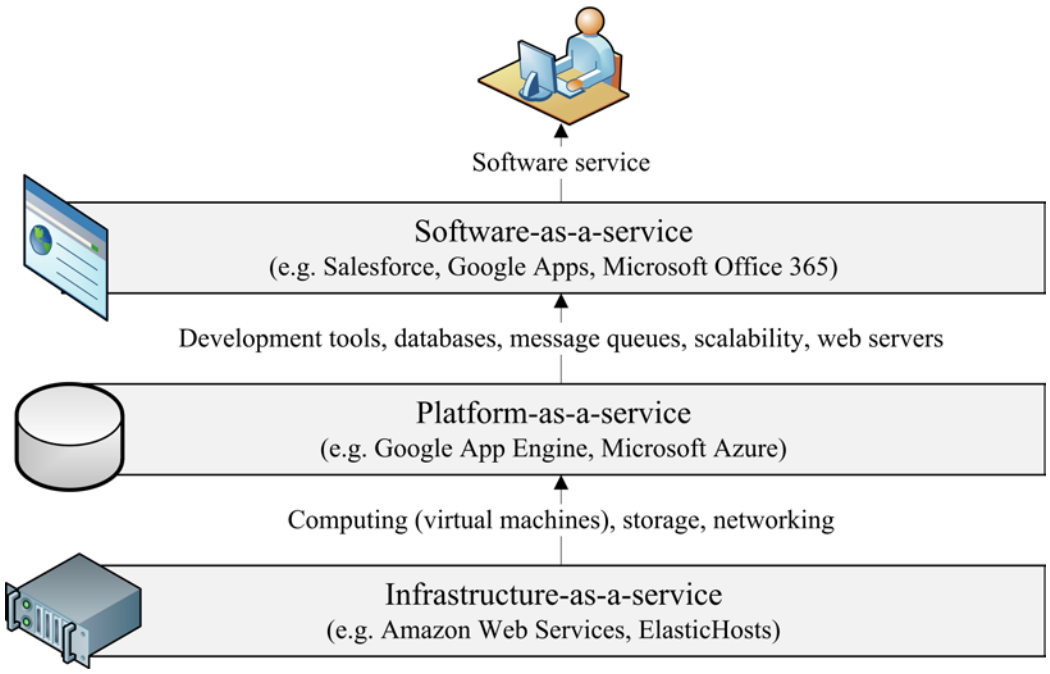

Figure 2.1. *Cloud service stack*

With each service, the customer does not need to take care of or even know about the underlying layers. End users of SaaS can simply access the service practically anywhere, anytime [1, p. 4]. These services usually have detailed Service Level Agreements (SLA), which state Quality-of-Service (QoS) requirements the service should fulfill. Failure to fulfill certain availability and response time threshold as a provider are usually financially compensated to the customers.

2.1.2 Infrastructure as a service

In IaaS, customers can buy access to virtual machines hosted by the service provider. One of the most well-known IaaS offerings is Amazon Web Services (AWS) suite [2], accompanied by services from ElasticHosts, FlexiScale, GoGrid and Joyent. The centerpiece in AWS is the Elastic Compute Cloud (EC2) service [3], in which virtual machines (called "*instances*" in EC2) can be used on-demand. Two separate storage services are offered: Elastic Block Store (EBS) [3] which instances use for storing their

block device volumes (can be thought of as virtual hard drives), and Simple Storage Service (S3) for storing free-form data [4].

Management of resources is done programmatically via APIs or manually using web-based user interfaces. In AWS, user sends either RESTful Queries or detailed SOAP XML requests [5] transmitted over HTTP or HTTPS. AWS does not require users to encrypt messages using HTTPS, but each request must be signed with an AWS access key. In this thesis, AWS is used with the Query API over HTTPS.

In IaaS, customers only pay for what they use; there is no entry or minimum fee. A selection of machine types with varying CPU and memory capacity is provided. Users are free to choose an operating system of their choice and have full administrator privileges to configure the system. In addition to Linux distributions, AWS provides instances with commercial software such as Microsoft Windows Server editions. The customer does not need a license but will pay slightly higher fees.

Several reservation models also exist. On-demand reservations cost more than reservations made in advance, and AWS also allows users to bid for unused capacity. These features give customers the option to trade some agility to lower overall costs. In addition to uptime, traffic in and out of EC2 is also billed, though AWS instances are allowed to transfer data for free amongst each other using a private network. Amazon provides a cost calculator to project expenses.

IaaS services provide a diverse set of additional services such as automatic scaling and loadbalancing, but they are unsuitable for this work as Open Build Service manages workload scheduling by itself. Other features that cost extra include remappable IP addresses, fine-grained monitoring and virtual private clouds.

2.1.3 Elasticity – shifting the risks of provisioning

The key motivation for cloud customers are the economic benefits achieved from elasticity and transference of risk. As there are no up-front commitments and the customer avoids large, long-term investments in hardware as well as software licenses, cloud computing can be seen as transforming capital expenditures into operational expenditures, offering flexibility to the customer. Even though the expenditures may rise higher in the end, the customer is free especially from the risks of under- and overprovisioning hardware. [1, p. 10]

Figure 2.2 presents two ways of provisioning capacity. Provisioning statically (a) for the peak load incurs waste if the load fluctuates [1, pp. 10-11], while underprovisioning causes the service to slow down or even become unavailable. In services where load is caused by users, e.g. a web service, user discontent eventually causes the demand to decrease until capacity is sufficient again. Elastic cloud services make it possible to match the capacity with the demand (b). This transfers the risk of successfully provisioning needed hardware from the customer to the cloud provider.

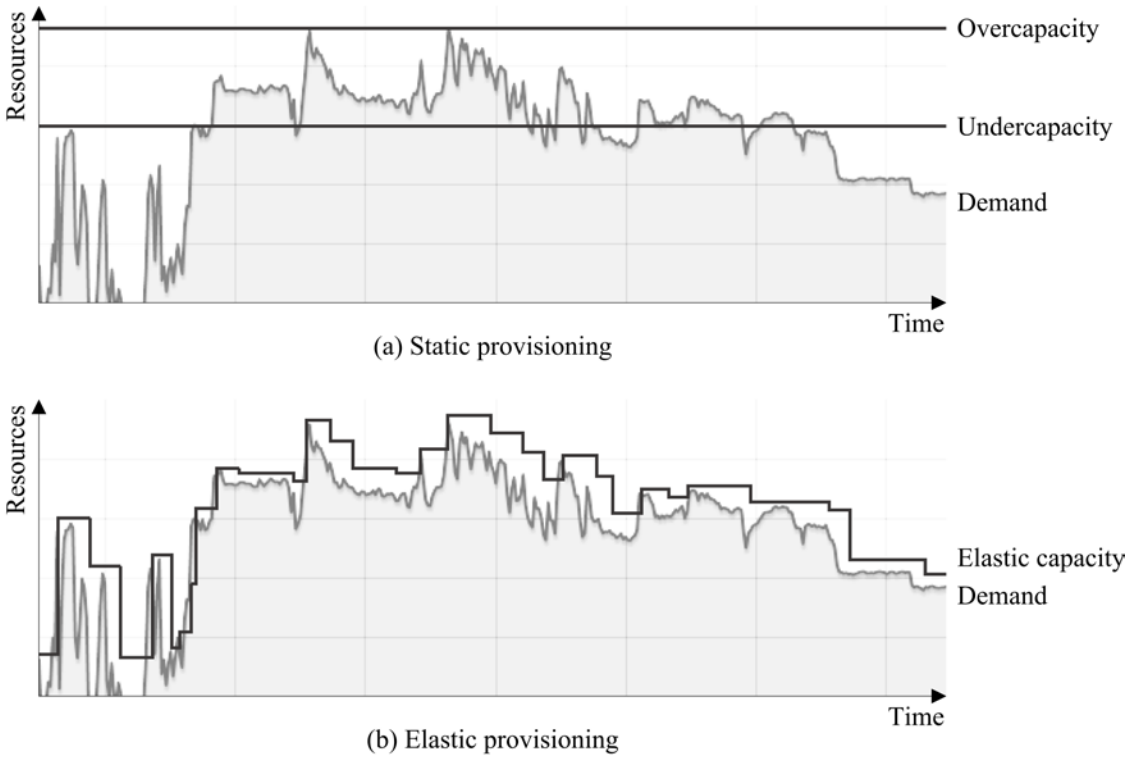

Figure 2.2. *Elastic capacity accommodates to demand*

With this elasticity it is possible to request huge pools of resources for batch processing. Due to the cost associativity, there is nearly no penalty for using 1000 servers for one hour compared to using one server for 1000 hours. This does however require that the jobs of the batch can be processed in parallel. [1, p. 7, 17]

The final decision on whether to move to the cloud or not depends on many factors. Different resources (computing, storage and networking) are often billed separately as different applications have very different usage characteristics. Running a datacenter incurs many additional costs like power, cooling, staff and the plant itself. [1, pp. 12-13] Utilizing cloud services, especially IaaS, requires a different set of knowledge than doing things completely yourself, but not necessarily less. Successful usage of cloud services requires more than just technological changes, but also business and management processes that support the transition. Many companies tend to take the cautious road and go for a hybrid solution using a combination of local and remote resources.

2.1.4 Challenges in the cloud

The main obstacles from the customer point of view are availability, security issues and vendor lock-in. Cloud computing, just like any other hosted Internet service, arouses critics' suspicions when discussing reliability.

Users expect high availability even from new providers. Several providers have had multiple outages of varying sizes, but few in-house infrastructures are as reliable as best cloud services. Average uptime is one of the main factors discussed in SLAs. Amazon guarantees 99.95% availability and most providers tend to have similar figures [6]. Just

as ISPs use multiple network providers to backup their service, very high availability can be achieved only by using multiple cloud providers. [1, p.14]

Unlike in PaaS and SaaS, when providing IaaS to customers, it is difficult to offer automatic scalability and failover, as the implementations are somewhat application-specific [1, p. 8]. These become customer responsibilities which some people easily forget. Amazon has several datacenters across multiple geographical *regions*, with each region divided into multiple *availability zones*. Regions are completely isolated from each other, meaning that instances from several regions cannot be managed as a whole and instances cannot be moved from one region to another. Inter-region communications is traditional Internet traffic between public IP addresses. Availability zones of a region are physically separated from each other, so that they would not be affected by a single physical event such as a fire. Instances in several zones in a single region can be managed as a group. In April 2011, Amazon suffered a severe outage in the U.S. Virginia region, caused ultimately by a router misconfiguration, rendering EBS inaccessible and causing availability issues for several days [7]. The failure of a whole zone caused customers to pour into other zones, causing another zone to run out of capacity. Even though the incident affected several zones, some customers were able to keep their services available as they had implemented cross-zone or cross-region failover.

Security of the customer data is also one of the key concerns and raises discussion, as the service provider has the ultimate power and the customer is in a lesser position to protect their own data. It must be noted though that storing encrypted data in the cloud can actually be more secure than storing unencrypted data in private datacenters [1, p. 15-16], and tools have emerged for encrypting and decrypting data in-house so that it is stored in the cloud in encrypted form [8]. Laws like USA's PATRIOT Act, Health Insurance Portability and Accountability Act (HIPAA) and Sarbanes-Oxley also play a big part in confidentiality issues as it may be practically impossible or too cumbersome for a company to put its data into the cloud [1, p. 15-16]. On the other hand, cloud providers do not like to be liable for their customers' misacts [1, p. 18]. When whistleblowing website Wikileaks was kicked out of EC2, it led to questions about cloud provider neutrality [9]. In addition, wrongdoings of other customers can hurt legitimate customers through reputation sharing, when public IP addresses are reused [1, p. 18].

From the agility perspective, cloud customers should be able to easily opt in and out of a service, without a major risk of vendor lock-in, which could otherwise lead to increasing prices and reliability problems [1, p. 15]. Failure to manage this causes undesirably high exit costs when ending the use of a service. The abundance of emerging cloud services has produced a number of rivaling technologies that are not interoperable. Several organizations have been formed to produce standards related to cloud computing from API and virtualization specifications to policies [10]. A meta cloud concept, a cloud of clouds, has been proposed in the pursuit of separating the details of cloud infrastructure from the services that are put on top of them. Common policies are needed to make the transition in and out of a service as smooth as possible. One concrete example is the Open Virtualization Format (OVF). Virtual machines can be easily ex-

ported and imported between different virtualization platforms that support OVF. For IaaS, there are several open-source abstraction layers like Libcloud [11] and Deltacloud [12] that hide the differences between cloud provider APIs, allowing users to manage instances in different clouds in a similar manner. The developers of these layers constantly update the software with support for new cloud providers and updates for modified APIs.

2.2 Hybrid cloud and cloudbursting

Clouds that publicly provide services to customers are referred to as public clouds, while private clouds are services that are provided by organizations themselves or by a private partner. The requirements for what can be called a private cloud, or even simply a cloud is fuzzy. A definite example of a private cloud is an IT department offering virtual machines to research and development projects of the same organization.

A hybrid cloud is a combination of multiple clouds or a combination of organization's local resources and a public cloud. As such, the hybrid model is often used as a first step in moving traditional in-house computing to external clouds. The underlying use of several clouds is usually hidden from the end-user, but provisioning should take into account the heterogeneity of the providers: their SLAs, billing and current status.

While computational resources are static in traditional computing, the demand is often dynamic. Cloudbursting (also in some cases called fog computing or surge computing as in [1]) is a concept of expanding a pool of local resources into a public cloud when local capacity reaches its limit. It essentially creates a scalable and elastic hybrid cloud infrastructure, on-demand without manual intervention [13]. Cloudbursting is especially related to handling sudden load peaks; allowing extra jobs to overflow to the cloud and making sure applications remain available when local resources become saturated. Cloudbursting differs from load balancing, in that not all of the resources are ready and waiting to serve requests [14]. Machines need to be deployed on the external site, and configured on boot-time. It may take several minutes for the resources to become available.

Cloudbursting and hybrid clouds have been seen as *"a compromise between enthusiasts and conservatives"* balancing between cost and trust [15]. Basic services and extra computing power can be provided from the cloud and critical services for customers provided from the private data center of the organization [13]. The economic driver in cloudbursting is that maintaining an infrastructure that can sustain even the highest peaks would be costly, especially if the peaks are rare and far larger than average load [14]. The overall cost depends greatly on the cloud provider and on the type of application – its traffic, computational, and storage requirements. Dias de Assunção et al. [16] studied the cost-benefit of using cloud computing to extend a local cluster. Usage of different work scheduling policies balance between performance and usage cost, which varied depending on the load of the system.

Marshall et al. [17] also studied the effects of different scheduling policies. In their work, they have also developed a model of an elastic site for describing cloudbursting, shown in Figure 2.3. In this model, the resource usage of a static, local site is monitored and additional resources are requested from the cloud based on the demand.

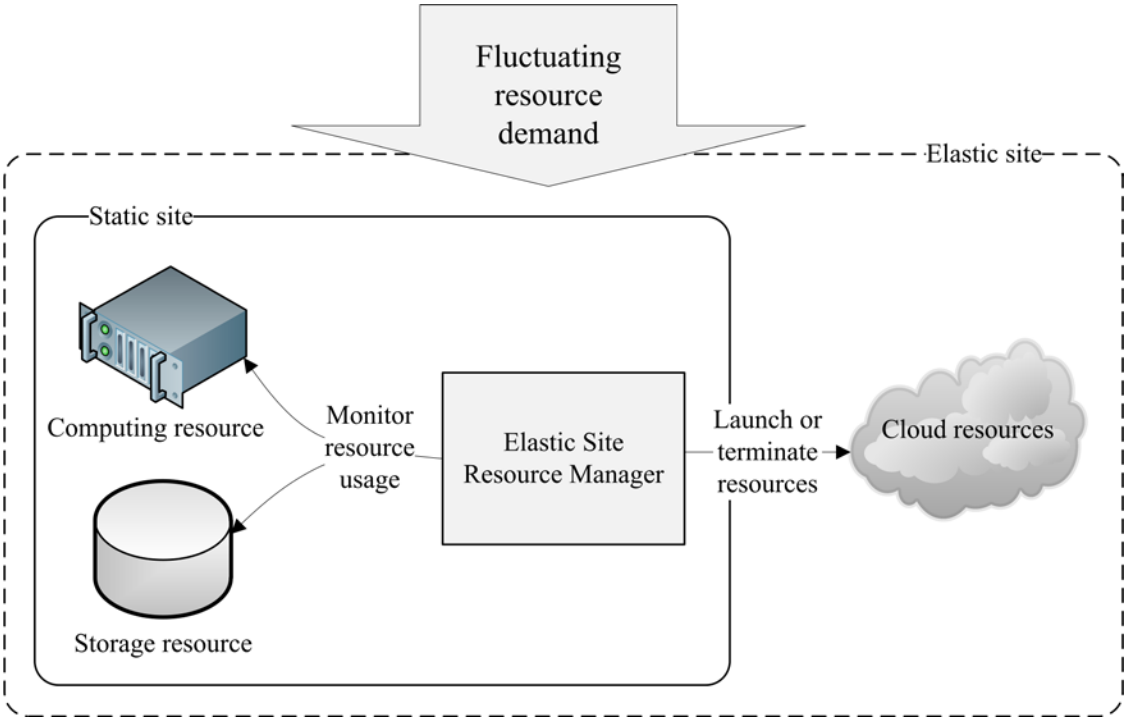

Figure 2.3. *Elastic site model [based on 17]*

For a service that has very granular workload, such as a web server, directly monitoring the utilization of resources such as CPUs and disks may be enough. With larger, more time-consuming tasks however, basing decisions on cloud resource usage can be inaccurate. In their implementation, a job manager dispatches jobs to worker nodes. The job queue of the job manager is monitored and new worked nodes are requested from the cloud when needed.

From different types of cloud services, IaaS is the most attractive one when setting up an elastic infrastructure for a service that already distributes jobs to worker nodes. IaaS offers on-demand resources with complete control over the software stack [17, p. 1]. This allows easily deploying worker nodes on virtual machines in the cloud, without making major changes to the service itself.

2.3 Managing cloud services

To effectively manage large, distributed and virtual infrastructures, several pieces of software are needed. RESERVOIR (“Resources and services virtualization without barriers”) is a European project to develop open-source cloud technology that has produced a framework for describing a complete software stack for managing cloud services. This

framework is presented in Figure 2.4, showing its three layers: virtualization, virtual infrastructure management and service management.

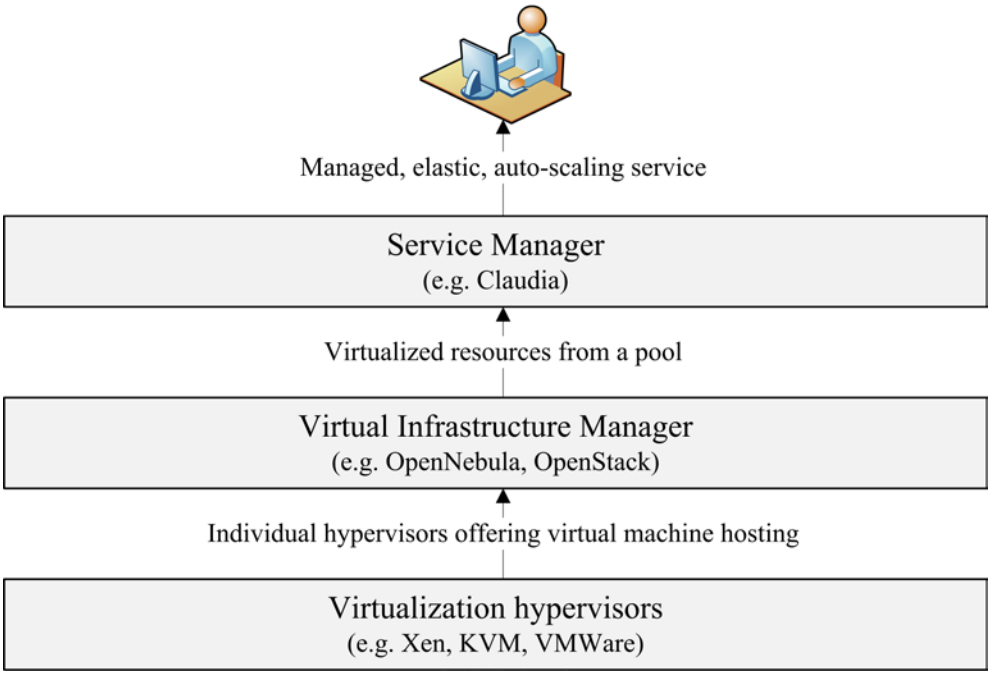

Figure 2.4. *Three-layer cloud management architecture [based on 18]*

This architecture separates the management of infrastructure from the management of services. This decoupling makes it possible to easily provision resources based on demand [19, p. 19].

Virtualization is one of the key enablers for cloud computing, allowing a large pool of physical resources to be split into finer-grained logical parts. This increases efficiency as hardware utilization can be pushed higher while keeping data and processes of different customers separated. Separation of the software and hardware allows migrating software from failing hardware, even on-the-fly.

Virtualization can be split into three categories: Full virtualization, paravirtualization and OS-level virtualization. In full virtualization, hypervisor software such as VMware or KVM separates the virtual machines from the real hardware, catching hardware calls arriving from the guest operating systems. The hypervisor can either be a program running on the host operating system, or it can itself run as a host operating system on bare metal. In paravirtualization, the guest works in conjunction with the host. Paravirtualization requires specific support from the hardware (Intel VT or AMD-V) or the guest operating system. Xen is an open-source paravirtualization technique, also used in AWS. OS-level virtualization differs from the other two in that the guest machines are not separate operating systems, but merely compartments in the host. The guests use the kernel of the host and only have separated userspaces. OS-level virtualization software includes Linux Containers (LXC), Solaris Containers and OpenVZ. [20]

While hypervisors manage virtual machines of a single physical machine, Virtual Infrastructure Managers (VIM) control, monitor and deploy large groups of virtual machines via the hypervisors. Multiple virtual machines can be launched from a single template disk image, which is copied into runtime disk images for each VM [21, p. 3]. Each VM will then use its runtime disk image, allowing them to edit their disk contents independently. Policies can be set to govern high availability and smart VM placement, so the user does not need to specify where a new VM should be provisioned [19, p. 20-21]. This serves as a foundation for providing IaaS, bringing the benefits of virtualization to distributed infrastructure. IaaS providers have their own, often proprietary VIMs which their customers use through some restricting API.

Several free and open-source VIMs exist, most notably OpenStack, OpenNebula, Eucalyptus, CloudStack and Nimbus. Many of them can be largely extended with additional software (e.g. OpenNebula's scheduler can be replaced with Haizea providing advanced resource scheduling and leasing) [22]. Most VIMs support all common Hypervisors and also offer the possibility to manage VMs from several IaaS providers through cloud abstraction layers, transparently mixing platforms together. The structure of OpenNebula which supports this is shown in Figure 2.5.

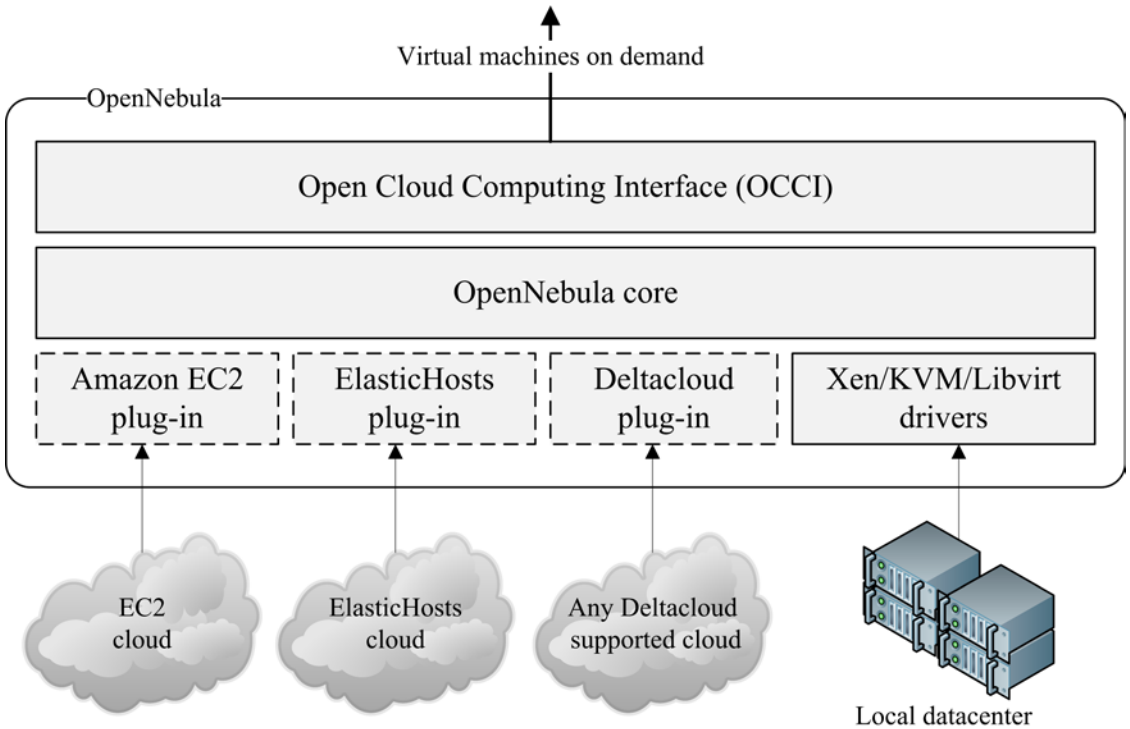

Figure 2.5. *Virtual Infrastructure Manager OpenNebula [based on 23]*

Administrators of a service do not want to constantly manage its infrastructure; they want automated management of the service as a whole. The problem with VIMs is that they are not service-aware and provide resources on-demand. If they do provide autoscaling decision-making, it is usually based on resource usage (e.g. CPU utilization levels). Depending on the service, this may not allow resources to be provisioned early

enough. A service manager provides smart and automatic scalability; it is aware of the events of the software service that is being executed on top of it, and it adjusts the infrastructure accordingly. The service manager issues requests to the virtual infrastructure manager for more or less resources. [23, p. 1226-1227]

The resource manager presented in the elastic site model is a service manager. In cloudbursting, the service manager is the main decision-making component, controlling the acquisition of resources in order to keep the service running fluently.

Claudia is an open-source service manager where services are specified in Service Description Files (SDF), which inherits its syntax from OVF. With the services described to it, Claudia is aware of the service components, dependencies between them and their elasticity and business rules. With the service description and regular status updates from the components, Claudia is able to start service components in the correct order and customize them during their deployment. [23, p. 1228-1229]

There is also a simple service manager plug-in for OpenNebula which allows the user to easily start, stop and suspend a multi-component service. However, it does not provide elastic scaling and is mostly intended for easy startup of a service. The components of a service and dependencies between them are defined and the manager starts them in the correct order.

3 BUILDING SOFTWARE PACKAGES

In this chapter we will go through the basics of software building; how a software package is created. After that, the core software of this thesis, the Open Build Service is explained. We will study its inner structure, how it works as well as how it is used for building MeeGo.

3.1 Software building

Software building is process where software is compiled from source code and bundled with configuration data into packages. A software package is a piece of software that can be installed onto a target operating system using a package management system. Packages typically contain compiled code, so a package is built for a specific build target, certain hardware architecture (e.g. x86 or ARMv7) and a certain build distribution (e.g. MeeGo, Maemo or openSUSE).

Packages are very prominent in Linux-based operating systems, where applications and the operating system itself are composed of packages. There are several alternative package formats and their managements systems, most notably .deb (used in e.g. Debian, Ubuntu and Maemo) and .rpm (used in e.g. MeeGo, Red Hat, Fedora and SUSE). Package management systems like Advanced Packaging Tool (APT) and RPM Package Manager (RPM) make it possible to easily install, update and uninstall packages, which are fetched from hosted package repositories.

In addition to the actual data, a package holds metadata such as package name, version, how to install and uninstall it. This metadata originates from the recipe of the package; a file that not only holds the metadata, but also describes how to build the package from the sources. In RPM, this file is called a *spec file*. The operating system environment where a package is built is called a *build environment*. The spec file describes how the build environment needs to be prepared and how to complete the actual build process, what source code is needed and which patches need to be applied and so on. The build environment not only has all the tools and libraries but also the prerequisite packages accessible that are needed for building. In the beginning of a build job, these prerequisites are installed if they are missing.

To lessen the amount of redundant code, packages can depend on other packages; this eventually forms a dependency tree, shown in Figure 3.1. Packages can have separate build-time and run-time dependencies, in this work we will concentrate on build-time dependencies. The dependencies are listed in the spec file. Build systems use the dependency information to figure out which packages need rebuilding when a package

changes and which dependency packages need to be installed in a build environment. Package management software needs dependency information for installing a package and for removing unnecessary packages after uninstalling a package.

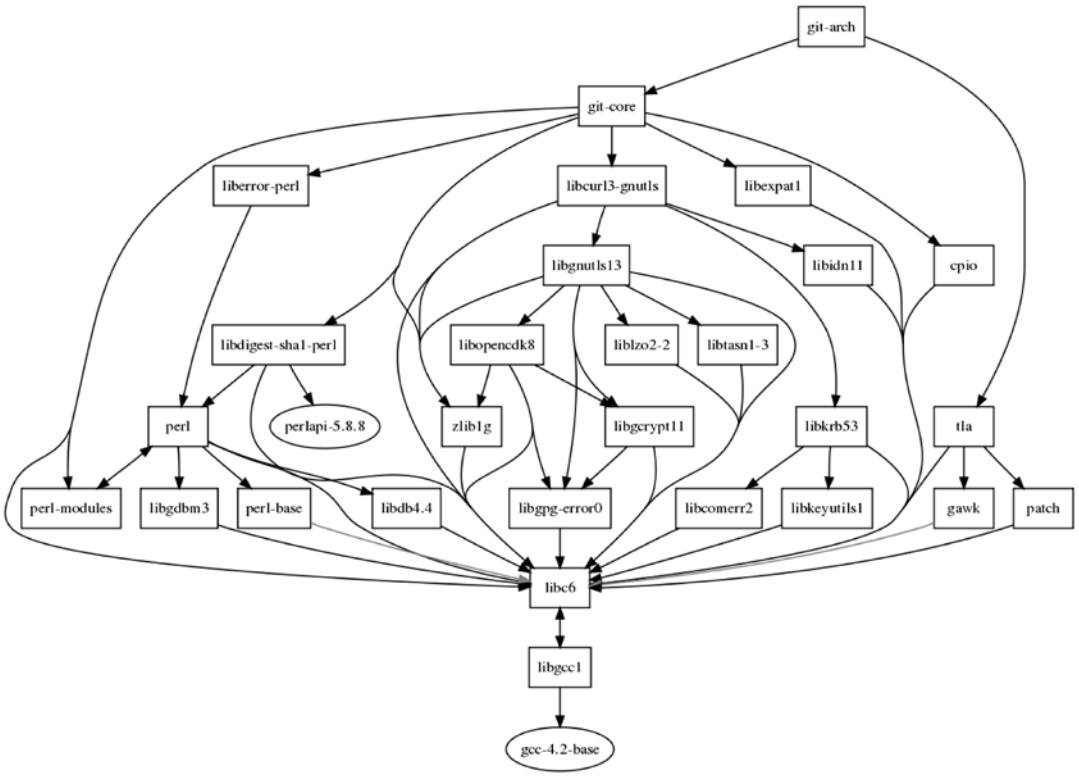

Figure 3.1. *An example of a package dependency tree for git-arch package [24]*

To decrease build times, packages can be built in parallel. If one of the prerequisite packages is being built, the depending build job is blocked until its build environment can fulfill all dependencies. The dependency tree varies for each package, but there are often bottlenecks, large packages that many other packages depend on, which can block a great number of jobs until it is finished. The number of jobs that can be built at a time varies greatly, therefore justifying the need for elastic capacity. Depending on the hardware, build environment and what is actually being built, a build job can take from a minute for a single small package to hours or days for building a complete operating system.

3.2 Open Build Service

Open Build Service (previously known as openSUSE Build Service), or OBS, is an open-source, cross-distribution development platform published under the GPL [25]. It is used to build software packages for several Linux distributions for different architectures such as ARM and x86. Users of the service are software developers who write source code and package descriptions. As changes are made, OBS automatically goes through the dependency chains and rebuilds necessary packages. It also provides tools

for collaboration, package repositories and is able to fetch sources from revision control systems (e.g. Git, Subversion) and even from other OBS instances. There are several more or less publicly accessible OBS instances running, including openSUSE OBS for developing openSUSE distribution [26], MeeGo Core OBS for developing MeeGo itself [27] and MeeGo Community OBS for developing software for MeeGo [28].

In addition to the public instances, several universities and corporations are running an OBS of their own [29]. OBS can be installed via package management but there are also appliance images available for easier setup [30]. The appliance images can be written to disk or imported as a virtual machine into a hypervisor and they include an installation of openSUSE Linux distribution and relevant OBS packages. For a useful build system setup, several servers with different roles are needed.

OBS is a centrally managed distributed system, where a head node receives requests, manages sources and packages and dispatches build jobs to multiple worker nodes. The head node is composed of several software services, which can be spread to separate hardware if necessary [31]. All of the services can be run on a single machine, including workers, even though it is not recommended for actual usage. A moderately scaled-out example is shown in Figure 3.2.

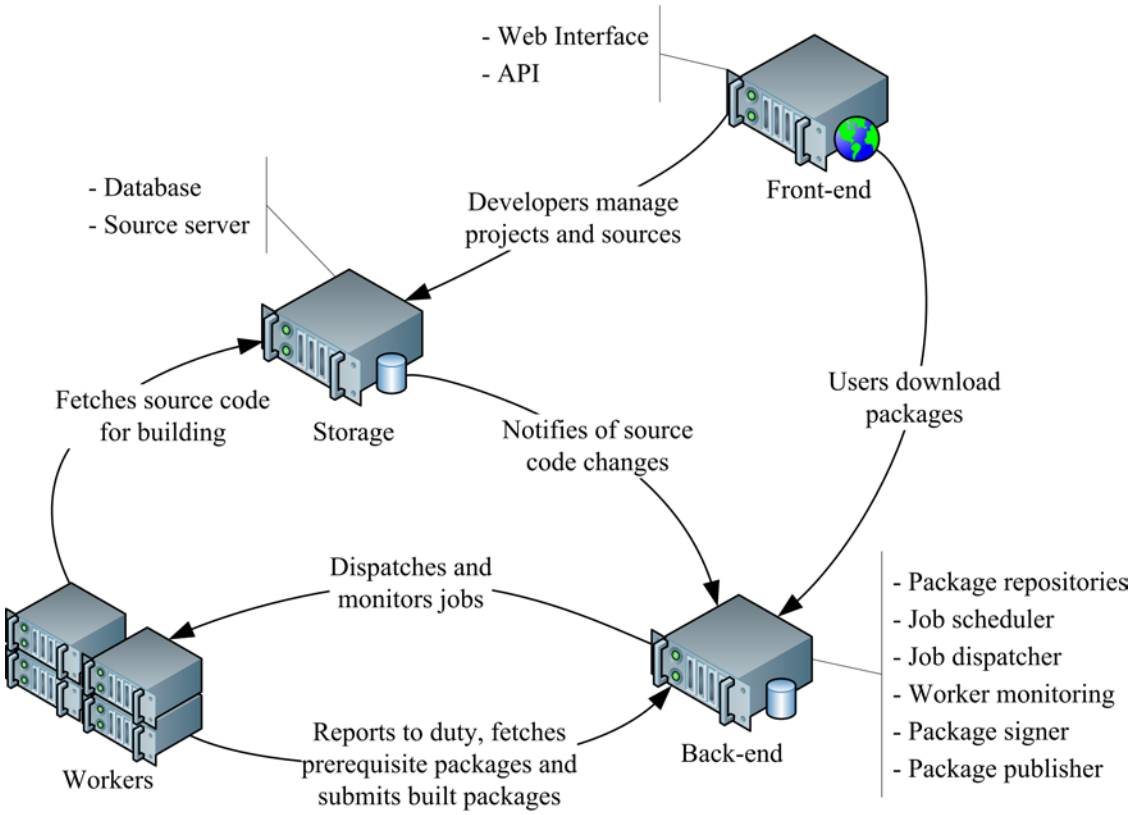

Figure 3.2. *Structure of OBS services*

The front-end consists of an API service and a web server. Users use OBS mainly by interacting with the API in the front-end: either by using the graphical web interface, the OSC command line client or by sending XML-formatted messages to the API [32]. The front-end takes care of access control.

The storage node has source code repositories and a MySQL database for persistent data. When software developers make changes to the source code, the source code service notifies the back-end.

The back-end hosts most of the decision-making components of OBS. When the job scheduler gets notified of source code changes, it calculates build dependencies and generates jobs into a queue for the dispatcher. The dispatcher assigns available jobs to idle workers. An optional monitoring warden keeps track of workers and whether they become unresponsive. When a job is completed, the resulting package is optionally signed as authentic and then published. End-users can download packages from the OBS package repository or they can be mirrored on a separate server.

The *workers* build the packages and due to this being the most time-consuming operation, there are usually several workers in a setup. Each worker can build a single package at a time, but several workers can run on a single machine, a *build host*. By default, OBS sets the number of workers to the number of CPU cores on the build host. The web interface has a monitoring view – shown in Figure 3.3 - which lists each build host, their workers and what the workers are doing at that time. This screenshot is taken during a test run from the system set up in this thesis.

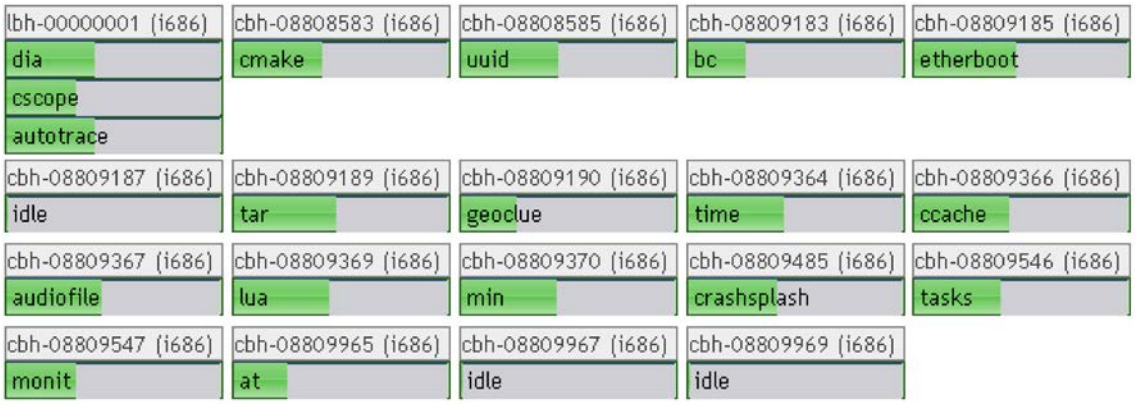

Figure 3.3. *Several build hosts building packages, observed from OBS web interface*

On startup, the worker process tries to connect to the predefined back-end server. Service Location Protocol (SLP) can also be used to connect automatically to servers in the same subnet. Workers are not authenticated by the server, so the build network must be kept isolated from other, more public networks to avoid malicious worker nodes. A worker may also build for multiple OBS server instances. Test and production environments can be separated into two servers, each having different configurations and source code and package repositories, but both utilizing the same pool of workers.

When a worker is assigned a build job it downloads prerequisite packages from the repository if they are not found in the package cache of the build host. Because builds have to be reproducible, the build environment is cleaned and recreated for each build. The worker then downloads source files, compiles the code and packages it. The resulting package is then sent to the package repository.

To battle the problem of bottleneck packages, build hosts in OBS can be manually split into two categories based on their hardware performance: regular ones and *power hosts*. These hosts will be prioritized for building critical packages, *power packages*. Both the hosts and the packages can be listed in the build service configuration file BSConfig.pm:

```
# List of power hosts that should build critical packages fast.
#our $powerhosts  = ["build20"];
# List of power packages that can be built on power hosts
#our $powerpkgs = [ "glibc", "qt" ];
```

As such, the lists are statically configured and editing the list will require a service restart or reloading configuration settings. Advanced dispatching policies could be extended into OBS to dynamically allocate the most powerful build hosts available to largest jobs.

For manageability, sources and resulting packages in OBS are organized into a hierarchy of *projects* and their subprojects. These projects output packages into *build repositories*. When a package is being built for a certain target distribution, its build-time dependencies are mainly searched from the target's build repository. For reusability, links can be created to other repositories and packages can be branched into new projects. [33]

3.3 MeeGo and its build services

MeeGo is a Linux-based operating system aimed to run on various mobile devices and information appliances. Born by combining Intel's Moblin and Nokia's Maemo mobile operating systems, it is open-source and the project is hosted by the Linux Foundation, a consortium for promoting and standardizing Linux. It was first announced in February 2010, with the first release following in May 2010. [34]

MeeGo inherited the use of RPM package management and OBS from Moblin (while Maemo was Debian package -based). MeeGo has two major public OBS instances: one for the development of MeeGo core and another one for the community to develop software for MeeGo [35]. As of September 2011, the core OBS consists of 20 build hosts each running 8 workers for building 21 000 packages in 2 000 projects. In the MeeGo project, OBS is part of the release infrastructure, BOSS "Build Orchestration Supervision System", a much wider vision of release management [36].

MeeGo is targeted at device vendors as a platform to build products, consisting of MeeGo Core and a set of frameworks for different vertical markets (e.g. In-vehicle infotainment, smart television). On top of this stack vendors develop their own User Experiences (UX), the user interface, the suite of applications and widgets and the look and feel of the operating system. The MeeGo project itself only provides reference User Experiences. This architecture is shown in Figure 3.4.

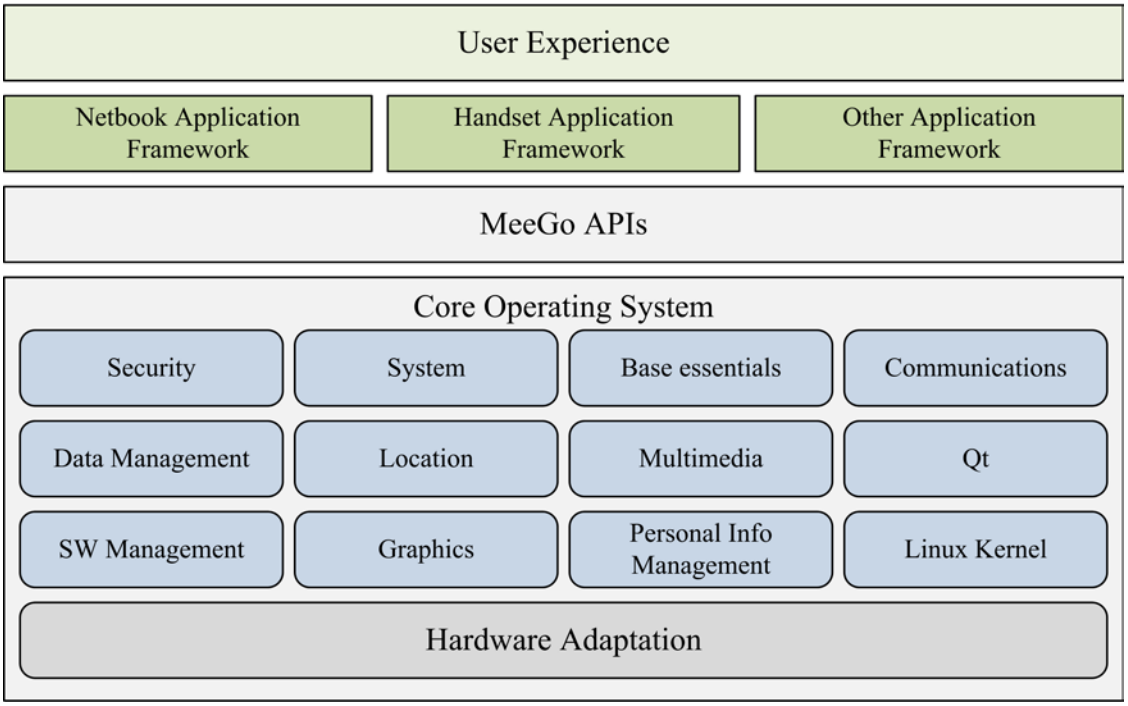

Figure 3.4. *MeeGo architecture layers [37]*

Vendors developing MeeGo-based products can benefit from Inter-OBS links, shown in Figure 3.5. Just as packages can be branched from one project to another, projects from other OBS instances can be linked another OBS instance. This allows the vendors to develop proprietary code in their own OBS while public development of the MeeGo core is done on the MeeGo build services.

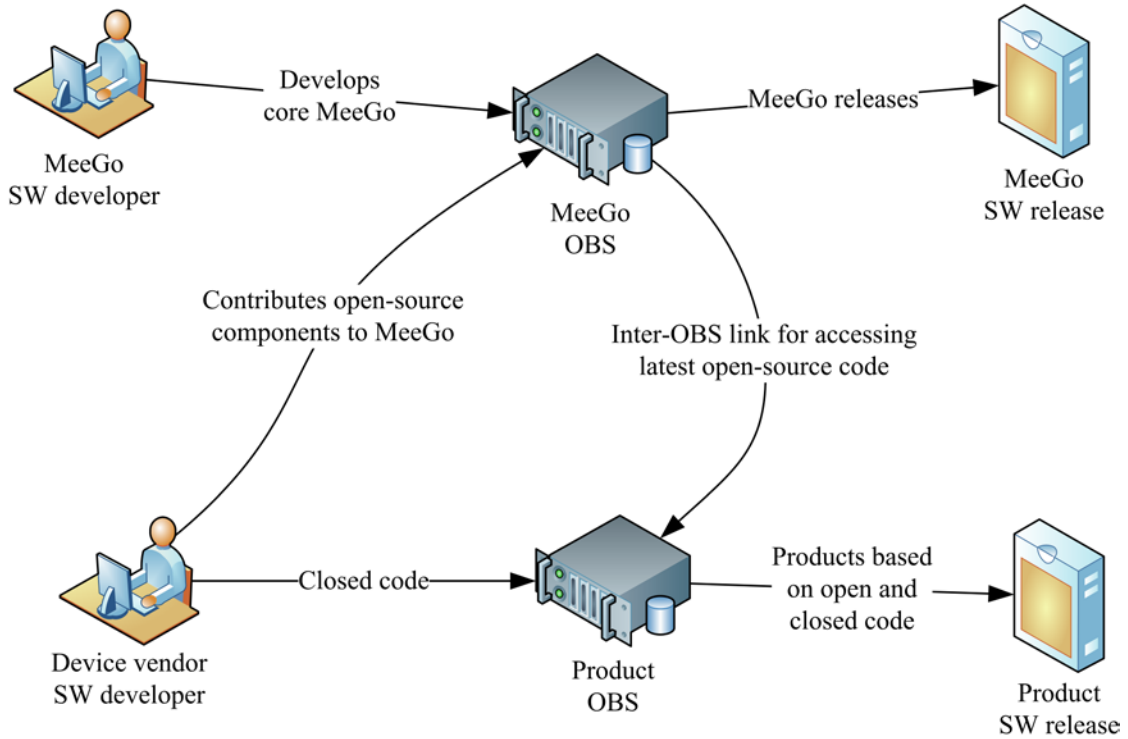

Figure 3.5. *Release collaboration through OBS linking [based on 38]*

This is done by creating a meta project which behaves like a symbolic link in Linux: content beyond it seem to be locally present and projects and packages can be used through it, even though they are in a remote location. Vendors will have the latest core packages available. Inter-OBS links can also be used inside a single organization to separate different tiers of development such as test environments into their own systems.

4 EXTENDING MEEGO BUILDING TO THE CLOUD

In this chapter, the practical work of the thesis is documented. Here the question on how to set up a cloudbursting build service is answered. First we will take a look at the overall architecture of the system. We will then go through how the infrastructure and the build service itself were set up. Finally we will go through the design and implementation of the service manager.

4.1 Planning the overall architecture

The overall requirement for the build system was that it would be locally hosted and would smartly use cloud resources for extra capacity when needed. The cloudbursting should be completely automated and done in a way that both costs and build times would be minimized [1, p. 18].

As workers also exist in-house, auto-scaling features of IaaS providers cannot be used. Furthermore, a single worker can only build a single package at a time, no matter its instantaneous resource usage. A worker that is building a package is completely occupied, even though its resource utilization would be temporarily low. Demand for new resources cannot be therefore determined from the usage of resources at that time. This requires monitoring of both the work queue and the build jobs, and allocating resources accordingly.

Traffic between worker nodes and OBS server as well as traffic between the service manager and AWS API servers should be encrypted. Similarly to the work by Moreno-Vozmediano et al. [19], open-source VPN implementation OpenVPN is used to securely connect worker nodes from the cloud to the rest of the system. These nodes authenticate themselves with a certificate to the VPN server.

Based on the RESERVOIR's management architecture, several components were picked and an initial architecture was formed, presented in Figure 4.1 with example software which could be used shown in parentheses. The architecture was designed to present a locally managed hybrid cloud, where new build hosts could be dynamically spawned to several IaaS providers as well as to a local private cloud.

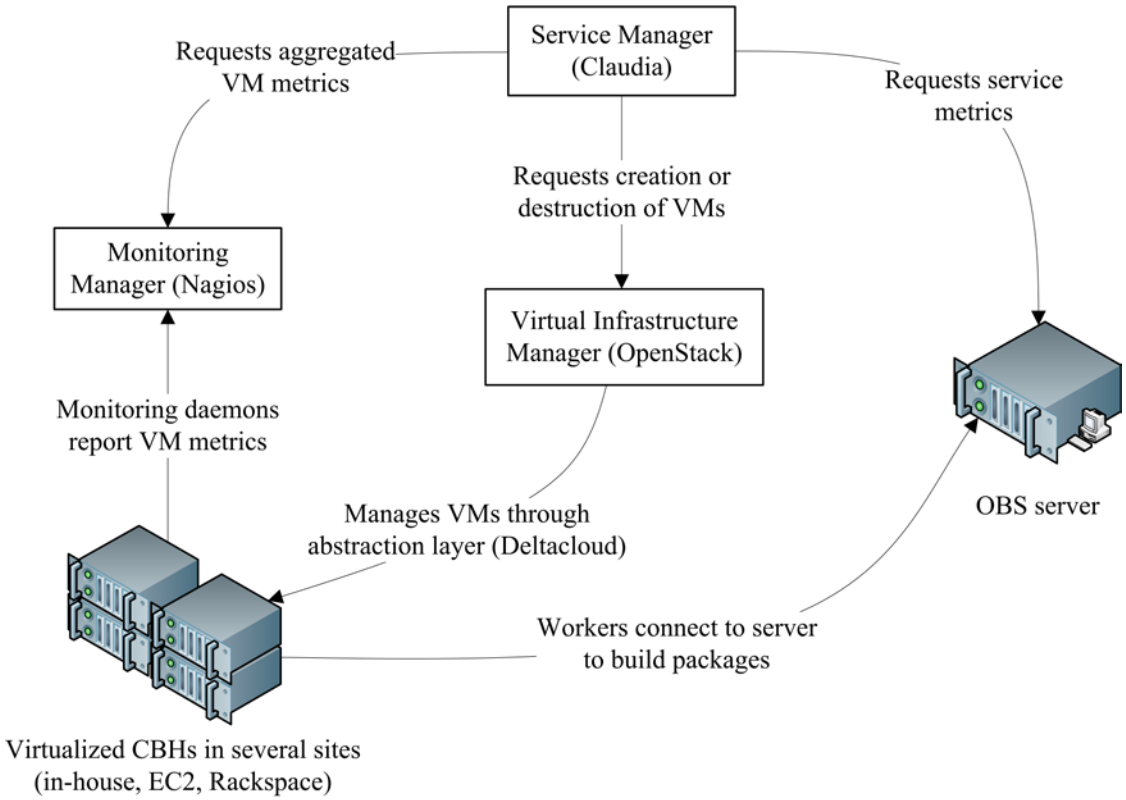

Figure 4.1. *Suggested architecture for production use*

Due to time constraints, the implemented architecture had to be simplified. Many features that were nice-to-have in this work but more crucial in production setup had to be dropped. Support for multiple parallel IaaS providers through Libcloud or Deltacloud was dropped, affecting fault tolerance. Build hosts were also divided into Local Build Hosts (LBH) and Cloud Build Hosts (CBH). LBHs would be seen as static hardware investments and would be manually started up and left on. CBHs would be EC2 instances, managed and monitored through AWS APIs by the service manager. With these simplifications, a separate VIM is not needed. In addition, all machines were initially configured manually instead of using configuration management systems, traditionally used in complex production-level systems.

The key problem area in this thesis proved to be the service manager. OBS itself handles scheduling and dispatching centrally, but is not capable of spawning new workers, let alone new virtual machines. AWS autoscaling feature does not support the hybrid cloud architecture necessary in this work. Traditional load balancers like the AWS load balancing service are designed to handle web traffic with lots of small requests and quick replies. Requests in OBS take minutes, easily even hours to serve which leads to starting up new instances for each request. Thus the autoscaling feature would need to communicate tightly with the OBS server. Only two viable service manager implementations were found (Claudia and Service manager plug-in for OpenNebula), but their development had seemed to slow to a crawl. The properties and needs of software services vary greatly and even with a generic service manager, the service would still be needed to be described to the manager through code. As OBS is the only service to be

managed, it has simple component layout (one server and multiple identical workers) and users only interact with the main OBS server, a full-blown service manager might be overkill. It was decided that a simple service manager script will be implemented instead. The resulting simplified architecture is presented below in Figure 4.2.

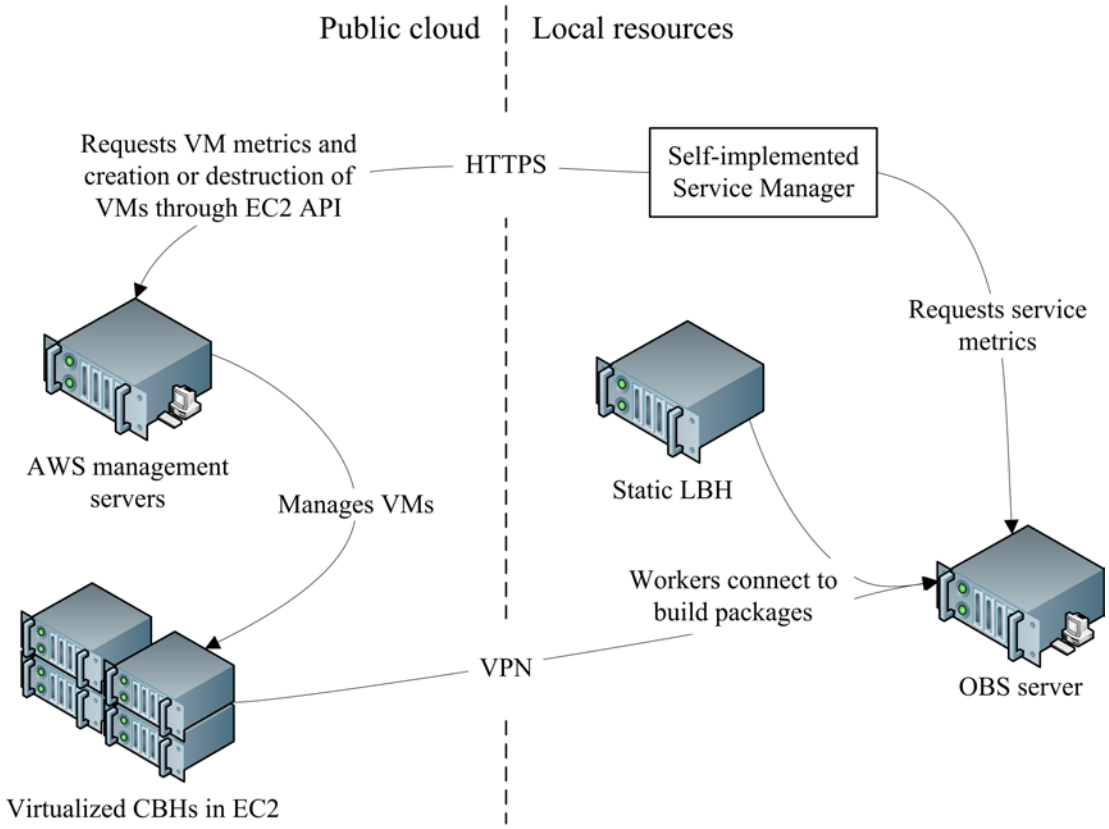

Figure 4.2. *Simplified architecture implemented in this work*

The system uses an aggressive elasticity policy, i.e. it gives explicitly a new worker for each job until the max number of workers is reached. This is simple and minimizes build time but costs extra. Smarter policies (e.g. do not launch a new instance if some job is about to finish) would require extra data from OBS, including build time estimates.

4.2 Infrastructure and build service setup

The build service will be established in two separate locations. A local, in-house network will have all OBS management services on one server and a local build host as another server. The remote cloud will only have build hosts. The build service was first set up with local resources only and then extended later when it was confirmed working.

4.2.1 Local build host and build server

The local cluster was set up on two VirtualBox virtual machines on separate desktop computers, one working as the OBS server running all the management services and the

other as a local build host with three workers. The server and build host setup were loosely based on the OBS appliance installation instructions and the MeeGo-community instructions on how to set up a cross-building, MeeGo-enabled OBS [39][40]. The latest stable appliance images were used as a base for the setup. The image is a disk image that includes openSUSE operating system with OBS components preinstalled, which requires some configuration before running successfully. Several configuration files were edited manually (mainly /usr/lib/obs/server/BSConfig.pm and /etc/sysconfig/obs-server), static hostnames were added to hosts file and OpenVPN connection from the OBS server to the gateway was set up. Needed disk partitions and worker settings were configured.

By default, the server appliance also runs workers. They were disabled to dedicate the machine for the OBS server. Even though they built successfully, the server front-end slowed down severely when the workers were building. Local build host was assigned to have one worker per core to saturate the resource usage.

4.2.2 Cloud build hosts

An AWS account was created and security keys and certificates were generated and downloaded. Next, an Amazon Machine Image (AMI) had to be prepared for the CBH. In EC2, each VM instance is always launched from a certain AMI, comparable to a VM appliance bundle in VirtualBox. This kind of template contains all the necessary software that is needed for a single CBH, and through boot-time customization (via User Data), all CBHs can be created from the same static AMI. Relations between instances and different images are represented in Figure 4.3.

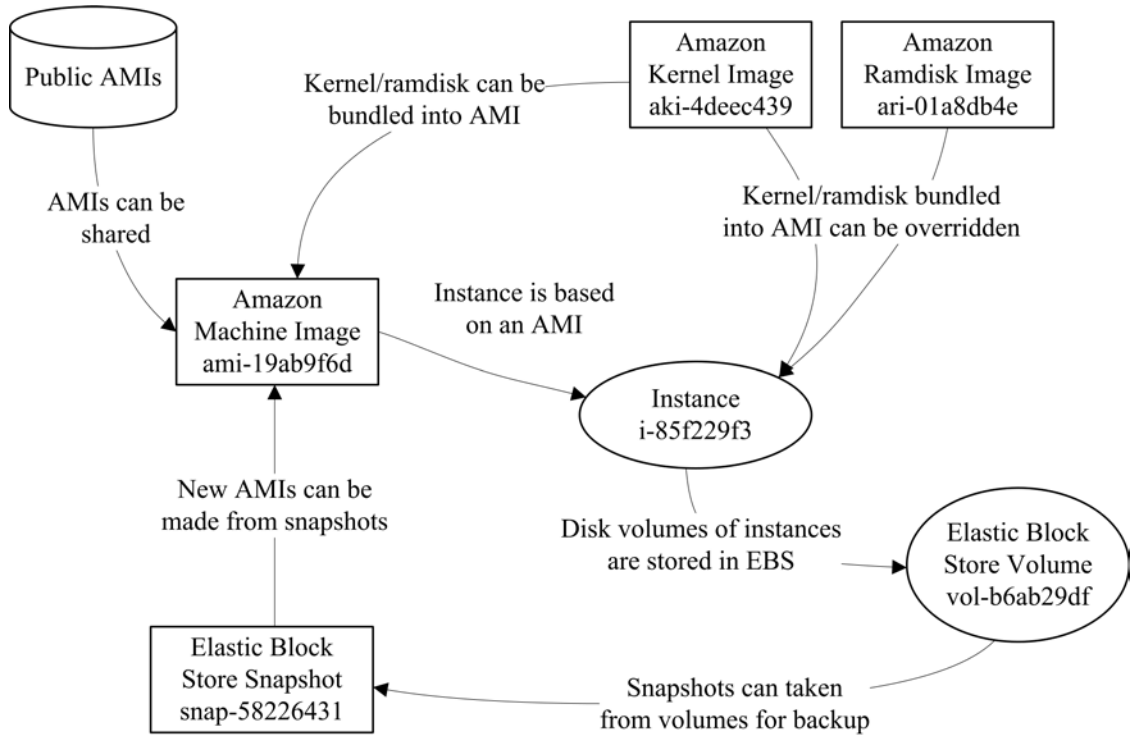

Figure 4.3. *EC2 instances are created from Amazon Machine Images*

Machine, Kernel and Ramdisk images as well as snapshots are read-only, so a new one needs to be created every time its contents need to be changed.

A variety of Amazon and community-made AMIs are publicly available. Custom AMIs can also be created either from an existing EC2 instance (that was created from another AMI earlier) or imported to EC2 using official or 3rd party tools. Currently the official tool only supports VMware's Virtual Machine Disk (VMDK) format with Windows Server 2008 Service Pack 2 [41] with support for other operating systems and image formats is planned to be released later. While the image format does not pose a restriction (as almost any VM image can be converted to VMDK format), the operating system restriction renders the tool unusable for this case. OBS is heavily tied to OpenSUSE, and EC2 offers this as a community AMI.

The AMI for CBHs in this thesis was configured manually by launching an instance from a community-based openSUSE AMI (ami-19ab9f6d, 32-bit openSUSE 11.4), editing it to run needed software and then creating a new AMI from the instance. This new AMI would then be used to spawn all the OBS build hosts, while the stem instance originally edited was spared for later updates. For publishing the image without leaking private details such as certificates or keys, a generic CBH AMI needs to be created with 3rd party tools. This AMI would then not have OBS server address and needed VPN details, so these will have to be configured to each instance when they are initiated.

The first problem was that the AMI defines the size of the disk volume, which was only 2 GB for the openSUSE AMI. This is inadequate for a build host, so the disk had to be extended. A snapshot was created from the stem instance, a new larger disk volume (8 GB) was created from the snapshot and the partition and file system were enlarged with basic shell commands to use the new larger space.

The second issue was that EC2 reset the hostname to default with every boot. After forcing DHCP not to update the hostname (/etc/sysconfig/network/dhcp), the issue was resolved. However, the hostname could still not be configured using the User Data option of EC2. User Data allows the user to pass arbitrary data to the instance when it is created. This base64-encoded data can be interpreted as environment variables or shell commands to be run after the instance has booted up. In the openSUSE AMI, the processing of User Data was disabled. After enabling it (from /etc/sysconfig/amazon), an arbitrary hostname could be passed to a new instance in the spawning request and the name would stick.

With the new disk image in place, all packages were updated using zypper. In addition, the openSUSE tool repository was added to zypper for installing OBS worker (package obs-worker). The worker software was configured (/etc/sysconfig/obs-worker) and needed partitions were set.

After the worker, OpenVPN was installed and configured with relevant settings and keys and was set to start on boot (using insserv command). The OBS worker fails to start if it cannot find the server at first attempt. The daemon init scripts for both software were edited (Start-Requires variable) to make sure that the worker starts last and

the VPN connection starts right before the worker. With these changes, the VPN connection came up on boot, but the worker failed to start. Investigating the logs revealed that the VPN connections comes up 5 seconds too late and by that time, the worker has already started and failed to find the server. As a workaround, a 10 second sleep time was added to the beginning of the start function of the worker's init script.

With only a few days use, the logs started filling with hundreds of failed SSH login attempts per day, a common sight with servers having an open SSH port towards the Internet. The EC2 security group was edited to only pass packets to SSH port from a certain address range. SSH was only used for debugging the workers and is not normally needed.

In EC2, disk volumes can be set to be destroyed when the instance that they are attached to is terminated. Unfortunately the current version of the web interface did not allow changing the setting. Creating an AMI from the persistent stem instance inherited the persistence of disk volumes. Build hosts that were created from the new AMI and then terminated, left behind their volumes. To auto-destroy the volumes upon instance termination, the AMI image had to be created using EC2 CLI tools instead of the web interface. A snapshot has to be created first from the stem instance, and then the new AMI can be created from the snapshot using ec2-register command:

```
ec2-register --name tebs-ec2-worker-v0
             --description "tebs - EC2 cloud worker"
             --architecture i386
             --kernel aki-4deec439
             --block-device-mapping "/dev/sda1=snap-defdb9b7::true"
             --region eu-west-1
             --verbose --headers --debug
```

The "true" in the block-device-mapping option designates that volumes of an instance should be destroyed when the instance itself is terminated.

4.3 Implementing a service manager

To automate the cloudbursting, a service-aware service manager was needed. Implementation of the service manager depended tightly on the components it was to communicate with, the AWS and OBS. AWS management API is used to manage and monitor CBHs, while OBS API is used to monitor job queue and build status.

Initial prototyping of the AWS API was done in Perl script language using the Net::Amazon::EC2 Perl module, which proved to lack several features. First, the module was fairly old and used an older version of the AWS API (AWS supports older APIs by including API version in each request), lacking support for newer features such as assigning tags to EC2 instances. In addition, the module did not use HTTPS by default (API messages are required to be signed, but not necessarily encrypted and thus can be carried over HTTP), though this was possible to implement by adding support for

IO:Socket:SSL Perl module. The Java-based command line tools that Amazon provides were also evaluated, but relying on them would have required parsing their output instead of reading variables in code.

Python script language was later chosen for the script as OSC, the CLI tool of OBS offers Python modules for OBS interaction and there is a feature-rich Python module 'boto' [42] for accessing AWS API. The service manager will communicate to the servers using these modules, as shown in Figure 4.4.

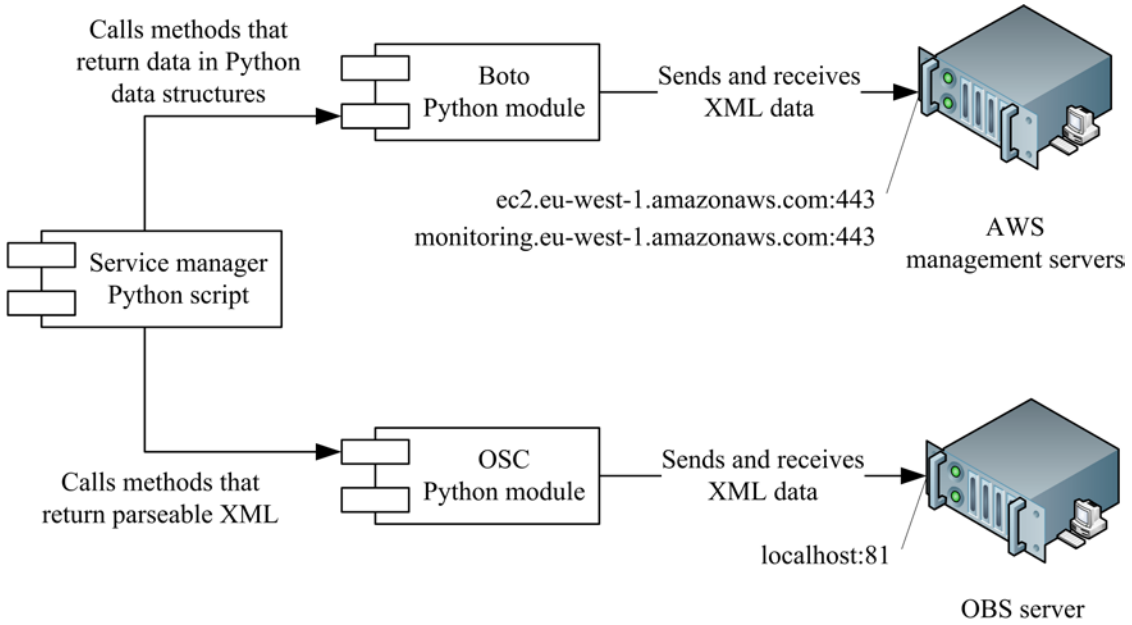

Figure 4.4. *Script components and their connections*

Boto Python module was installed to the OBS server. AWS authentication keys were added to its .boto configuration file and the module was ready to be imported in the service manager script. Boto uses HTTPS by default and also supports the use of certificate validation through a configuration setting (https_validate_certificates = True). Boto sends a HTTPS request with Query parameters and the server replies by sending the data in XML. An example of API messages can be seen in Appendix 1.

The OSC tool can be used remotely in any computer but it also comes with the OBS server software. OBS credentials and the API URL were added to the .oscrc configuration file. After importing the OSC Python modules, its functionalities could be used in native Python instead of calling the CLI tool and parsing its output. Even though OSC is only used for authentication and still hands over XML which has to be parsed in the service manager, using OSC allows easier extensibility for the service manager. An example of the parsed XML can be seen in Appendix 2.

The service manager was designed so that it could be ran manually or scheduled with cron job scheduler. Every time it runs, it gathers all the necessary data it needs to understand the current situation and makes decisions on whether it should launch or terminate instances or do nothing. Due to simplification reasons, the service manager was chosen to only manage CBHs and to ignore LBHs which would be always on.

The Python script has been made publicly available under the GPL [43]. The service manager script was added to a user's home directory on the OBS server. Running the OSC tool creates a basic template of the .oscrc settings file, which was further edited manually to grant the script access to the OBS API. The script was set to run once every minute using cron job scheduling. As OBS also has a cron job running every minute to update worker statuses, the script was delayed using sleep function to make sure it runs after the status updates. The main logic of the script is presented in Figure 4.5.

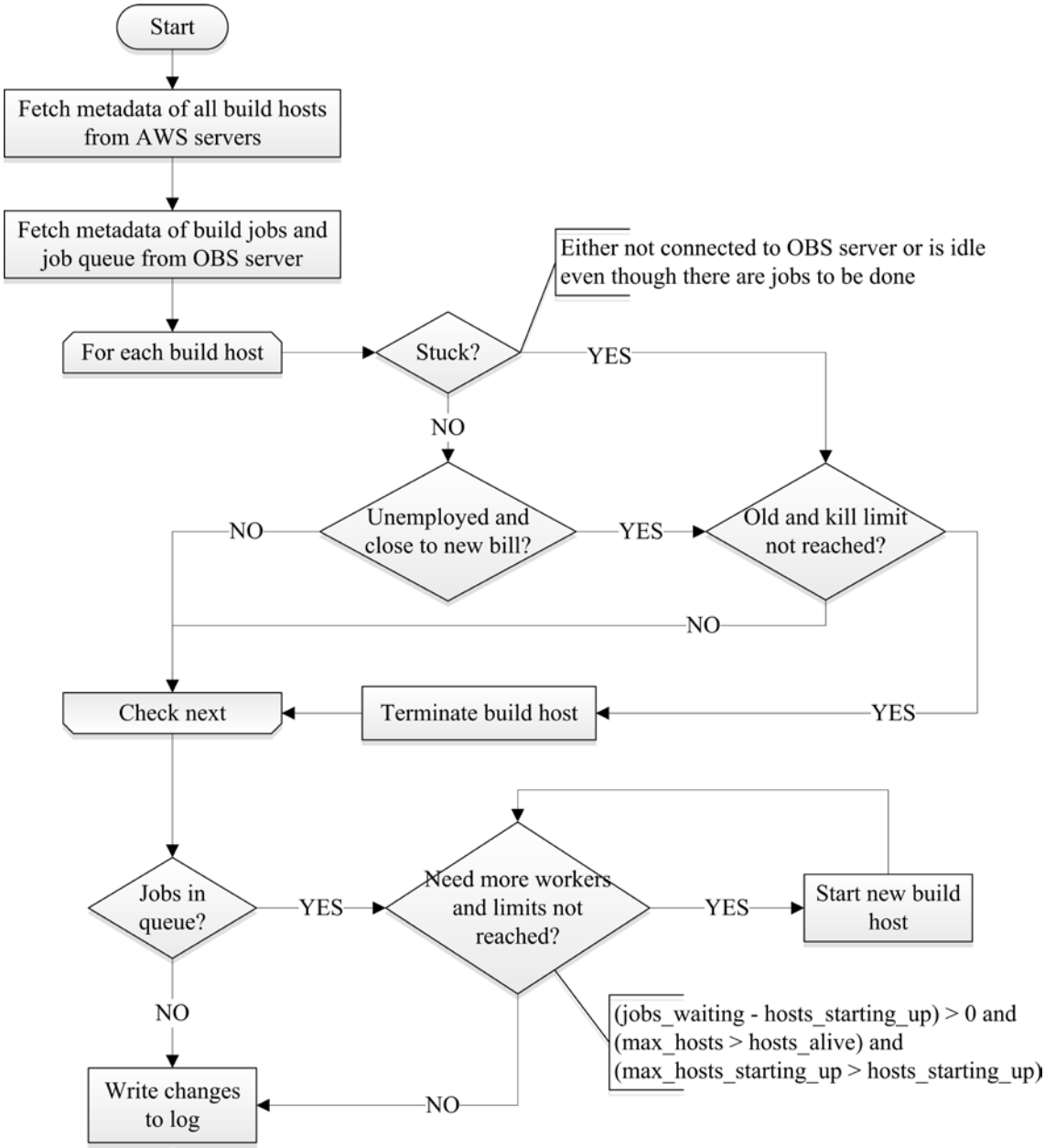

Figure 4.5. *Simplified flowchart of the implemented service manager script*

First, the service manager connects to AWS management and monitoring servers and requests metadata of instances owned by the specified user account. This data includes instance ID, name tag, state, launch timestamp, IP address and the latest measured CPU utilization. Second, the service manager connects to the local OBS server and

requests metadata of both the current workers (hostname, state, packages being built if any) as well as the queue of future jobs (number of jobs waiting and number of blocked jobs). It combines the data of VM instances and worker processes into a single data structure, which can be printed when using the script manually as shown in Figure 4.6.

```
LISTING ALL CLOUD BUILD HOSTS (CBH)
EC2 ID     HOSTNAME     TYPE      CPU STATE     TTL IP              CURRENT JOB
_____________________________________________________________________________
i-781d660e cbh-08808583 m1.small  4   running   27  79.125.99.15    cmake
i-661d6610 cbh-08808585 m1.small  3   running   27  46.137.1.214    uuid
i-24126952 cbh-08809183 m1.small  3   running   37  46.137.4.130    bc
i-20126956 cbh-08809185 m1.small  35  running   37  46.137.63.90    etherboot
i-2e126958 cbh-08809187 m1.small  23  running   37  46.137.47.67    IDLE
i-2c12695a cbh-08809189 m1.small  49  running   37  79.125.62.254   tar
i-2812695e cbh-08809190 m1.small  32  running   37  46.137.56.201   geoclue
i-96116ae0 cbh-08809364 m1.small  3   running   40  46.137.3.189    time
i-94116ae2 cbh-08809366 m1.small  5   running   40  46.137.24.139   ccache
i-92116ae4 cbh-08809367 m1.small  3   running   40  46.137.62.106   audiofile
i-90116ae6 cbh-08809369 m1.small  6   running   40  46.137.63.79    lua
i-9e116ae8 cbh-08809370 m1.small  3   running   40  46.137.62.78    min
i-32106b44 cbh-08809485 m1.small  3   running   42  46.137.60.157   crashsplash
i-18106b6e cbh-08809546 m1.small  4   running   43  46.137.58.111   tasks
i-06106b70 cbh-08809547 m1.small  3   running   43  46.137.60.210   monit
i-9c176cea cbh-08809965 m1.small  3   running   50  46.137.47.22    at
i-9a176cec cbh-08809967 m1.small  1   running   50  46.137.62.248   IDLE
i-98176cee cbh-08809969 m1.small  0   running   50  46.137.39.182   IDLE
```

Figure 4.6. *Status printout from the service manager, listing each CBH*

Build hosts have the following naming convention: “[Prefix]-[ID number]”. The script fetches details of all EC2 instances found in the region for the account, but only manages (i.e. terminates) the ones with the correct prefix in their name tag. The ID number is a substring of the UNIX timestamp when the instance was created, with the first two digits removed to conserve space. For example, a CBH launched at June 23rd 2011 05:56:23 GMT (1308808583 as UNIX time) is named “cbh-08808583”. This allows unique build host names for over three years (the cropping of the timestamp can be easily removed to hundredfold the time) and helps troubleshooting.

For the service manager to understand what is going on, build jobs and build hosts need to be mapped into instances. If a build host is idle according to OBS server, the instance needs to be terminated, but for that the instance ID is needed.

EC2 allows the user to add arbitrary metadata tags to instances to ease management. These key-value pairs can hold any variables the user wants and they can be fetched and changed over the API. When requesting a new instance, the script generates a unique name for the build host and requests the EC2 Name tag and the hostname to be the same via User Data. This is how build host data from OBS (“package cmake is being built at hostname cbh-08808583”) is linked to instance data from EC2 (“instance ID i-781d660e has a name tag of cbh-08808583”). When using a tag with a key “Name”,

the value of that tag is displayed in the EC2 web interface in the Name column, as shown in Figure 4.7.

Name	Instance	AMI ID	Zone	Type	Status	Public DNS
tebs-stem	i-85f229f3	ami-19ab9f6d	eu-west-1c	t1.micro	stopped	
cbh-08809969	i-98176cee	ami-53d1e627	eu-west-1c	m1.small	running	ec2-46-137-39-182.
cbh-08809967	i-9a176cec	ami-53d1e627	eu-west-1c	m1.small	running	ec2-46-137-62-248.
cbh-08809965	i-9c176cea	ami-53d1e627	eu-west-1c	m1.small	running	ec2-46-137-47-22.e
cbh-08809547	i-06106b70	ami-53d1e627	eu-west-1c	m1.small	running	ec2-46-137-60-210.
cbh-08809546	i-18106b6e	ami-53d1e627	eu-west-1c	m1.small	running	ec2-46-137-58-111.
cbh-08809485	i-32106b44	ami-53d1e627	eu-west-1c	m1.small	running	ec2-46-137-60-157.
cbh-08809370	i-9e116ae8	ami-53d1e627	eu-west-1c	m1.small	running	ec2-46-137-62-78.e
cbh-08809369	i-90116ae6	ami-53d1e627	eu-west-1c	m1.small	running	ec2-46-137-63-79.e
cbh-08809367	i-92116ae4	ami-53d1e627	eu-west-1c	m1.small	running	ec2-46-137-62-106.
cbh-08809366	i-94116ae2	ami-53d1e627	eu-west-1c	m1.small	running	ec2-46-137-24-139.
cbh-08809364	i-96116ae0	ami-53d1e627	eu-west-1c	m1.small	running	ec2-46-137-3-189.e
cbh-08809190	i-2812695e	ami-53d1e627	eu-west-1c	m1.small	running	ec2-46-137-56-201.
cbh-08809189	i-2c12695a	ami-53d1e627	eu-west-1c	m1.small	running	ec2-79-125-62-254.
cbh-08809187	i-2e126958	ami-53d1e627	eu-west-1c	m1.small	running	ec2-46-137-47-67.e
cbh-08809185	i-20126956	ami-53d1e627	eu-west-1c	m1.small	running	ec2-46-137-63-90.e
cbh-08809183	i-24126952	ami-53d1e627	eu-west-1c	m1.small	running	ec2-46-137-4-130.e
cbh-08808585	i-661d6610	ami-53d1e627	eu-west-1c	m1.small	running	ec2-46-137-1-214.e
cbh-08808583	i-781d660e	ami-53d1e627	eu-west-1c	m1.small	running	ec2-79-125-99-15.e

Figure 4.7. *Several CBHs created from the same AMI, listed in EC2 web interface*

After collecting data of the current situation, the service manager goes through all the found instances to be nominated for termination. Build hosts that are not connected to the OBS server or that are idle even though there is work to be done could be stuck and will be nominated. However, as it takes a varying amount of time for the instance to start up, create a VPN connection and connect to the OBS server (usually 3 minutes), newborn build hosts will be ignored until they reach a boot time threshold (set to 5 minutes).

Idle hosts that have no more work and that are close to a new recurring payment are also terminated. VM instances in EC2 are each billed hourly, for every starting hour counting from the time the instance was started. Starting a new instance and letting it run for ten minutes and repeating it five more times in one hour, will cost six instance-hours instead of one. Starting and stopping instances should therefore be minimized. The TTL value seen in the Figure 4.6 is the amount of time left in minutes until the instance is billed again for the next full hour. Idle build hosts will be kept waiting for new jobs until their lifetime approaches another full hour.

When bad and unnecessary build hosts have been terminated, the service manager determines whether it should request new build hosts. Each instance that is considered still booting up is reserved to have a job from the queue. Subtracting the number of booting workers from the number of jobs in the queue equals to the number of how many more workers are needed. There are also limits for the total number of build hosts and for the total number of hosts that are starting up. Many of the limit and threshold variables in the script are designed to be easily fine-tunable, allowing balancing between performance and cost.

Finally once the VM pool has been updated, the script writes any changes made to a log file on a single line as shown in Figure 4.8. By default, the script does not write anything if it did not terminate or spawn an instance. This is to minimize clutter in the logs as most of the time the script does not make changes (as it runs often).

```
2011-06-23 05:55:26 newwork: 30 new jobs created MAN
2011-06-23 05:56:25 servman: jobs=2+35    [ooo++]
2011-06-23 06:06:31 servman: jobs=11+31   [oooww+++++]
2011-06-23 06:09:31 servman: jobs=5+31    [ooowwwWWww+++++]
2011-06-23 06:11:25 servman: jobs=6+30    [ooowwwWWwwWLCCC+]
2011-06-23 06:12:28 servman: jobs=3+30    [ooowwwWWwwwWWWWC++]
2011-06-23 06:19:29 servman: jobs=3+22    [oooWwWWWWWwWWWwW+++]
2011-06-23 06:53:25 servman: jobs=0+2     [...wu-iiiiiiiiiiiiiii]
2011-06-23 07:03:26 servman: jobs=0+0     [...iu-u-u-u-iiiiiiiiii]
```

Figure 4.8. *An example of the service manager log file*

In the log syntax, each character inside the brackets represents a worker (with each CBH having only one worker in this case) with the exception of minus, which represents the termination of the build host in front of it. The first line indicates that 30 build jobs have been manually triggered. For testing purposes, a separate helper script was implemented to easily trigger arbitrary amount of rebuilds. The next time the script runs all three local workers are building (“o”). There are 2 jobs in the queue (and a total of 35 packages in the repository are blocked) so the script starts 2 new workers (“+”). 10 minutes later, large jobs that were assigned to the local workers in the beginning have finished, opening up new jobs for the local workers as well as 11 additional jobs. The 2 cloud workers that were started earlier are now working (“w” and “W” for working with low CPU usage). The script requests more workers, reaching the limit of five booting workers. Few minutes pass as the script is unable to launch more as the new workers are still booting. At 6:09, the five new workers have started building, so the script can once again launch new workers. At 6:11, one of the latest workers is identified as lazy (“L”) while three workers are not connected to the OBS server (“C”). All four non-building workers are redeemed, as they are considered to be young and still starting up. Lazy workers are workers that are connected to the OBS server but are not building even though there are jobs to be done. In this case, the script has run at a point where the new worker has just connected to the OBS server but has not started building yet. A minute

later, at 6:12, the lazy worker has started building. At 6:53, only one cloud worker is building, with rest of the cloud and local workers are idle (“i” and “.”). At this point, the earliest idle cloud worker is reaching the limit for a new payment (it was launched at 5:56, 57 minutes ago) and it is terminated as unemployed (“u-“).

4.4 Example of the complete system

In this example each build host is running a single worker process. The OBS server and LBH are continuously running in-house. The sequence presented in Figure 4.9 starts when a developer makes source code changes. The OBS server recalculates the package dependencies and schedules 2 build jobs (job-A and job-B) into the queue.

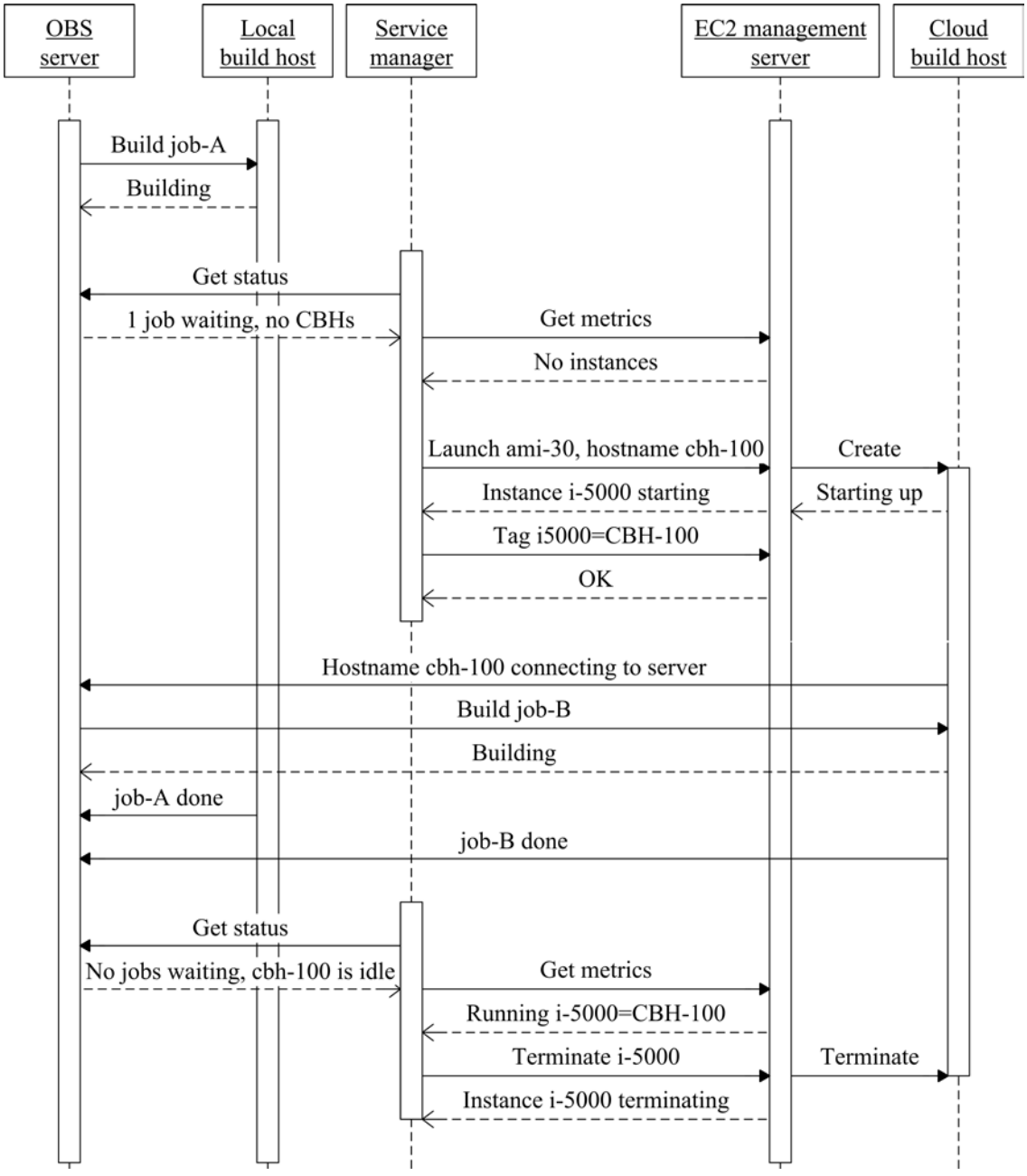

Figure 4.9. *Example sequence of the system*

OBS dispatches job-A to the available LBH worker while job-B is left in queue. The service manager script is started by cron. It requests queue status and a list of ongoing builds from the OBS server and a list of instances from the EC2 management server. Based on the data, the service manager decides that one new build host is needed. It generates a unique ID (CBH-100). It then requests the EC2 server to launch one new instance from the pre-created image (ami-30) and passes a script via User Data to set the hostname of the instance to “cbh-100”. EC2 accepts the request and passes the ID (i-5000) of the newly created instance in the response. The service manager requests that instance i-5000 should also have a Name tag with value CBH-100. Once the instance has started networking services, it executes the User Data, setting its hostname to cbh-100. The machine continues its boot cycle, opening a VPN connection to the network of the OBS server. It then starts a worker which connects to the OBS server reporting its hostname cbh-100. The server dispatches job-B to cbh-100.

Time passes on while both workers build. Once they are done, they report back to the server and send the resulting packages to the package repository. After this the service manager runs several times but does nothing until cbh-100 is nearing a new payment. The manager decides that idle cbh-100 should be terminated. Comparing hostnames and tags, it resolves that cbh-100 is running as i-5000. The manager then requests EC2 to terminate instance i-5000.

5 EVALUATION

In this chapter we will evaluate the implemented system through traffic and build time measurements, and take a look back at how the system fulfills the requirements and answers the research questions. We will also take a look at what was learned along the way, how to continue from here and what others have been doing related to the presented work.

5.1 Build measurements and analysis

One of the concerns in this assignment was that preparations for building in the cloud would take so long that it would not be feasible to outsource builds. In addition, the total cost of building in the cloud was also a concern. To address these concerns, several measurements were performed to evaluate the speed gain and the cost when cloudbursting. The tests are rough as the real environment is very dynamic: the performance of EC2 resources as well as the network latencies varies. In addition, build jobs arrive randomly and are not always scheduled to the same workers.

For testing, a set of 30 MeeGo packages of varying sizes were selected randomly and copied from the MeeGo repository to the private OBS instance, so in each test the same versions of the packages were being built. A list of the packages with build times from an example run can be seen in Appendix 3. Workload was initiated by a script that forced OBS to rebuild the packages.

As the CBHs did not have persistent cache for packages, no caching was used on the LBHs either. All workers had to download all dependency packages from the server before building. For simpler scheduling, only single-core machines from EC2 were used as build hosts, therefore each build host had a single worker which was able to build a single package at a time.

Four different scenarios were tested, each repeated three times for averaging. In each scenario, the LBH was enabled while the use of CBHs was varied. There was no significant variation in the build times and traffic amounts between different runs in each scenario, so the summary in Table 5.1 presents the averaged values for each scenario. Traffic values are the combined amount of traffic between the OBS server and the build hosts in an average run.

Table 5.1. *Benchmarking results*

cloud zones in use	CBH instance type	total time (h:mm)	traffic from LBH (MB)	traffic to LBH (MB)	traffic from CBHs (MB)	traffic to CBHs (MB)	packages built by LBH (MB)	packages built by CBHs (MB)
none	none	2:33	109	3859	0	0	30	0
all three	m1.small	1:05	139	1506	129	2457	11	18
one	m1.small	1:01	151	1448	145	2650	10	20
one	c1.medium	0:55	156	1221	113	2889	8	22

In the first scenario (the first line in the table), the service manager was disabled and only the LBH with its 3 workers was used to get a reference time. With packages waiting in the queue for almost the whole time, the total time was a bit over two and a half hours. Even though the 30 packages could not be built in parallel, it is clear that 3 workers are not nearly enough.

In the second scenario the service manager was enabled, allowing CBHs to be spawned when needed. The service manager was allowed to launch CBHs to any availability zone in the EU region of EC2. Even though the total build time was cut down to 40%, the system did not run flawlessly. Many of the new instances would not connect to the server within 5 minutes and thus were terminated by the service manager, eventually also causing one job to be skipped. The log shown in Figure 5.1 shows that even though jobs piled up in the queue, fresh build hosts were continuously terminated and new ones started (from 12:21 to 12:53).

```
2011-06-20 12:06:43 newwork: 30 new jobs created MAN
2011-06-20 12:07:26 servman: jobs=1+34     [..o+]
2011-06-20 12:12:37 servman: jobs=0+34     [oooi] MAN
2011-06-20 12:18:32 servman: jobs=10+31    [oooW+++++]
2011-06-20 12:21:28 servman: jobs=8+31     [ooowCWCWC++]
2011-06-20 12:24:31 servman: jobs=10+29    [ooowC-WC-wC-CC+++]
2011-06-20 12:27:29 servman: jobs=10+23    [ooowwwC-C-wCC+++]
2011-06-20 12:30:31 servman: jobs=4+20     [ooowwwwC-C-wWW++++]
2011-06-20 12:36:24 servman: jobs=0+17     [ooowwiwwwwC-wC-W]
2011-06-20 12:53:24 servman: jobs=0+3      [...iiwiwiC-ii]
2011-06-20 13:04:24 servman: jobs=0+2      [...u-iwiiiii]
2011-06-20 13:15:23 servman: jobs=0+0      [...u-u-iiiii]
2011-06-20 13:21:32 servman: jobs=0+0      [...] MAN
```

Figure 5.1. *Service manager log showing instances failing to start properly*

A separate test was made to simply launch one instance in each zone. All instances launched into running state, but only one of them managed to connect to the server within the 5 minute time-frame. The time limit for establishing a connection was then extended to 15 minutes and the test was repeated. Within 3 minutes from launching, instances from 2 zones managed to connect to the server. The third failed to connect to

the build server within 15 minutes. The web interface reported the failing instances to be in running state, but the system log was empty and all network connections to the instances failed.

For further tests, the zone was hardcoded to the best performing zone (eu-west-1c, but note that zone labels are account-specific and this could be e.g. eu-west-1a on another account) at that time. The third scenario was similar to the second one, but with all CBHs in the single availability zone. Total build times were similar with the previous scenario, with only slightly less time consumed. The effect of slower scale-up due to the zone problems in the second scenario was miniscule compared to the bottleneck in the first scenario where only three workers were building the packages.

In the fourth scenario, the instance type of the CBHs was changed from Small to Hi-CPU Medium. Again, the total build time was only slightly less. Overall, build time was cut by more than half of what it was with only one LBH, but this was due to the fact that the first scenario was severely bottlenecked.

The difference between using Small and Hi-CPU Medium instances was surprisingly minimal compared to the difference of their stated computing powers. According to AWS, a Small instance has 1 Elastic Compute Unit (ECU), a unit comparable to the CPU capacity of a 1.0-1.2 GHz 2007 Opteron or 2007 Xeon processor, while Hi-CPU Medium has two cores each with 2.5 ECUs [3]. Both machines have the same vague description for I/O performance: “moderate”. The effect of I/O performance on cloud build jobs is difficult to estimate as the AWS monitoring claims that disk usage is almost always 100% idle.

The explanation to this minimal difference is job blockage. In the first scenario there were not enough workers, causing a lot of jobs to wait. However, when the CBH pool provided seemingly endless number of workers in the later scenarios, the packages could not be built in parallel because of package dependencies. The significance of this effect depends heavily on the packages that are built in the system, but job bursts tend to have packages related to each other. The optimum between few fast workers and many slow workers needs to be found while still minimizing cost. As some packages have many dependent packages while other packages have no dependent packages, not all workers need to be particularly fast.

The cost breakdown of the test runs is presented in Table 5.2, fetched from Amazon’s billing history. These sums include the three test scenarios that used CBHs, each ran three times building 60 packages in the cloud in total. Free tier campaign in AWS allows some free usage per month during the first year of subscription. The free tier usage is transformed to billed figures here as if the service had been used for over a year. From the breakdown we can see that for the test runs, disk usage and snapshots costs $3.14, network traffic costs $2.66 and computing itself costs $17.10.

Table 5.2. *Cost summary for above tests*

Amazon Elastic Block Storage		
$0.11 per GB-month of provisioned storage	11.100 GB-Mo	1.22
$0.11 per 1 million I/O requests	8,185,013 IOs	0.90
$0.15 per GB-Month of snapshot data stored	5.280 GB-Mo	0.79
$0.010 per 10,000 gets (when loading a snapshot)	234,320 Requests	0.23
		$3.14
Amazon EC2 running Linux/UNIX instances		
$0.095 per Small Instance (m1.small) instance-hour	118 Instance-Hrs	11.21
$0.19 per High-CPU Medium Instance (c1.medium) instance-hour	31 Instance-Hrs	5.89
		$17.10
Amazon Data Transfer		
$0.100 per GB - data transfer in	24.930 GB	2.49
$0.000 per GB - data transfer out (first GB free)	1.000 GB	0.00
$0.150 per GB - data transfer out	1.137 GB	0.17
		$2.66

Computation forms three-fourths of the total cost. Even though some more expensive High-CPU instances were used, they performed similarly with cheaper instances ($0.295 per job on small instance vs. $0.268 per job on a High-CPU medium instance). High computational costs derive mostly from the aggressive scheduling policy used. More sophisticated policies could be implemented to increase worker utilization with minimal impact on build times. CBHs also have a static starting time overhead of about 3 minutes. As this affects only the first package for each build host, the worst case for building in the cloud is a burst of small packages, a one-hour break, followed by another burst of small packages.

Storage cost can be cut by optimizing the contents of the CBH image, removing all unnecessary content from it. Snapshots also cost a quarter of storage costs, which can be cut almost to a tenth by removing redundant snapshots.

Traffic cost is a small part of the total cost. The traffic is extremely asymmetrical, with over 20 times the amount sent to the workers compared to what is sent from the workers back to the server. On an average, 135 MB is sent to a worker prior to a job (sources and dependencies for the job). On a 100 Mbit connection this equals to less than a minute even with VPN encryption overhead. As build times range from 3 to 50 minutes (13 minutes on average) with the selected packages, effects of transfer times depend heavily on the job size. To save time, a common cloud-side cache of packages should be created for the workers. After these tests Amazon Web Services announced (following Microsoft's similar announcement concerning Azure) that inbound traffic will not be billed anymore and that outbound traffic will cost less than earlier [44]. With the new prices, the share of cost for traffic is even smaller. As cloud-side caching of packages would mostly reduce traffic cost, the only major benefit with caching is time-savings.

Overall, the implemented system works as planned, autoscaling to match the workload. Preparations for build jobs in the cloud take only a minor portion of the time and cloud workers successfully build packages with affordable cost. Cloudbursting is most useful to quickly overcome a large number of packages that do not depend heavily on each other. Sustained workloads are probably cheaper to handle in-house. There are however several shortcomings that need to be dealt for production use.

5.2 Cloud-readiness of Open Build Service

In general, build hosts can be distributed over the Internet without code changes and with very little configuration in OBS. With a virtual network, the build hosts are easily connected to the OBS server. As build hosts do not communicate between each other, the main requirement is that the network connections between the build hosts and the server have sufficient bandwidth. This is especially important if no package caching is in use due to the difficulty of implementing it in the cloud.

For cloudbursting purposes, worker status metrics from OBS are quite coarse. Main issue is that it does not provide any time estimates for the builds. It has been argued that it is impossible to estimate beforehand the total number of files that need to be created and modified in the build process. Estimates based on history data of previous builds would also be inaccurate if major changes are done in the package sources or if the package has been built on different hardware. In cloud environments, even the same types of instances do not have consistent performance. OBS is also slow to remove build hosts that have been terminated, so the actual state of the build cluster cannot be determined from the OBS metrics alone. The implemented service manager does recognize these “ghosts” as no CBHs are found from EC2 for them. The status of a worker is updated when it first joins the server, when it starts to build and when it finishes building. The warden process also looks for build jobs that are stuck and redispatches these jobs to other workers.

Main challenge with OBS was configuring it to work as error messages were generic and documentation poor. The documentation is quite outdated and distributed over many wiki pages. To battle the problem, Adrian Schröter started a collaborative project in the beginning of 2011 to write two guidebooks, one as a reference guide [45] and the other one for best practices [46]. As of October 2011, they are early drafts.

Most misconfigurations and problems in OBS simply led to service daemons not starting up. If a build host could not find the OBS server the service simply stopped without retries. Most of the configuration work is related to network and disk partition settings. When creating an OBS server from the appliance image, partition configuration is needed as the appliance provides only a 1 GB main partition which quickly fills from logs.

5.3 Suitability of Amazon Web Service for building

From the moment of requesting a new instance, it takes about 3 minutes on average until the workers are ready to accept build jobs. This is in line with other results (70-200 seconds [17]). While the starting delay may seem high compared to the build times of the smallest of packages, jobs tend to come in bursts. As packages have build-time dependencies, all the packages in these bursts cannot be built parallel at the same time. Later jobs built on the same instances will not suffer from the starting delay.

Main concern with AWS was its inconsistent behavior. Sometimes instances failed to start up properly at a specific zone, not connecting to OBS server, not responding to SSH connection attempts and not showing any activity in their system log. Often in these cases the status of the instance was stated “running” by EC2, so Amazon still bills for one instance-hour for each of these instances. No clear reason for this behavior was found. One explanation is that less used images take longer time to initialize in a zone and once an image is used often enough, it becomes cached in that zone allowing instances to start up faster. This problem was witnessed on and off during the six months of usage. The image for the CBHs was updated several times, which forced the creation of a new AMI. This may explain the reoccurrence of the problem.

During the work, the service also suffered two larger downtimes where customer data was lost due to bugs or human error. Even if a multi-vendor system is setup, the main OBS server should be kept in-house to ensure that packages can be built with at least some build hosts.

The service manager script was originally planned to monitor CPU and disk usage to detect crashed build hosts, but the AWS monitoring proved to be too coarse. The disk usage was very inconsistent, showing almost 100% idle for most of the time. The script was left to merely issue warnings of busy workers with low utilization and vice versa.

The general performance of EC2 instances is somewhat fluctuating but adequate. Due to the nature of IaaS systems, VMs share physical disks and may affect the I/O performance of other VMs [1, p. 17]. This affects build times as building is very disk I/O heavy. The smallest EC2 instances (t1.micro) are inadequate for building, as they run out of memory in bigger builds with their slightly over 600 MB of RAM. The other two types used (m1.small and c1.medium) worked flawlessly, but other types could not be tested as the image created on a 32-bit platform could not be run on 64-bit instances.

More powerful instance types tend to increase the number of CPU cores. To benefit from multiple cores, support for multi-threaded compiling may need to be added to the packages. A CBH can also run multiple workers, but this leads to the problem that single workers cannot be managed separately and hence a CBH with multiple workers cannot be shut down until all of its workers are idle.

5.4 Future work

For production use, the service management needs more advanced scheduling policies. Cost efficiency can be improved with minor effect on build times. More mature service managers such as Claudia should be considered. However, advanced scheduling requires developing features into OBS for more precise monitoring data and storing the management data locally on the management server.

5.4.1 Improving system robustness

The infrastructure for this build system should be made provider-agnostic, to ensure business continuity in vendor lock-in situations. Even with strict SLA, there are other risks, e.g. the provider may go out of business. To ensure availability, the system should use several IaaS providers with backup network connections. With a private cloud and several public clouds, VIM becomes mandatory. Provider-agnosticism can be further driven with the use of abstraction layers such as Libcloud and Deltacloud. If multi-provider setup appears to be troublesome, then at least cross-zone or even cross-region failover should be implemented, taking into account the problems with availability zones.

This brings challenges however, as caching becomes even a larger problem with several providers. Having one cache at each provider increases the traffic needed compared to using only one provider. Using one cache for all the providers introduces a single-point-of-failure, increases expensive extra-provider traffic and does not differ much from not using caching at all.

The service manager should accommodate to remote errors and fail gracefully. On an extremely rare case, an instance successfully booted up and connected to the OBS server but the AWS name tagging failed. The script did not identify the build host and would not terminate the instance automatically. This should be addressed with retries and local database to store instance status between management checks. The service manager should also monitor if instances have trouble starting up on certain zones and temporarily avoid using those zones.

5.4.2 Improving build efficiency

The utilization of CBHs can be monitored and optimized; the service manager should track how long build hosts build, how long are they paid for and thus how much they idle in total. The version demonstrated in this work uses an aggressive scheduling policy, i.e. it launches a new instance for each job that is ready to be built. If a large number of instances are started at the same time, each of them build a package for 5 minutes and idle the rest 55 minutes, the utilization is very poor. The launching could be then held back until a suitable utilization level vs. extra delay is found, thus saving money by not starting a new machine for each job. If incoming build jobs could be predicted by esti-

mating when blocked jobs become unblocked, instances could also be requested beforehand effectively mitigating the 3-minute startup delay with CBHs.

Caching the packages that workers download for dependencies would bring a performance as well as a small cost gain. Normally each build host has a cache of its own, shared by the workers running on that build host. In the cloud however, the most efficient way to cache is to have a single cache for all the build hosts. By default, OBS build hosts have a file directory where packages are put. Figure 5.2 presents different ways of sharing a package cache amongst CBHs in AWS. In AWS, the packages can be cached either on an EBS volume or in an S3 bucket.

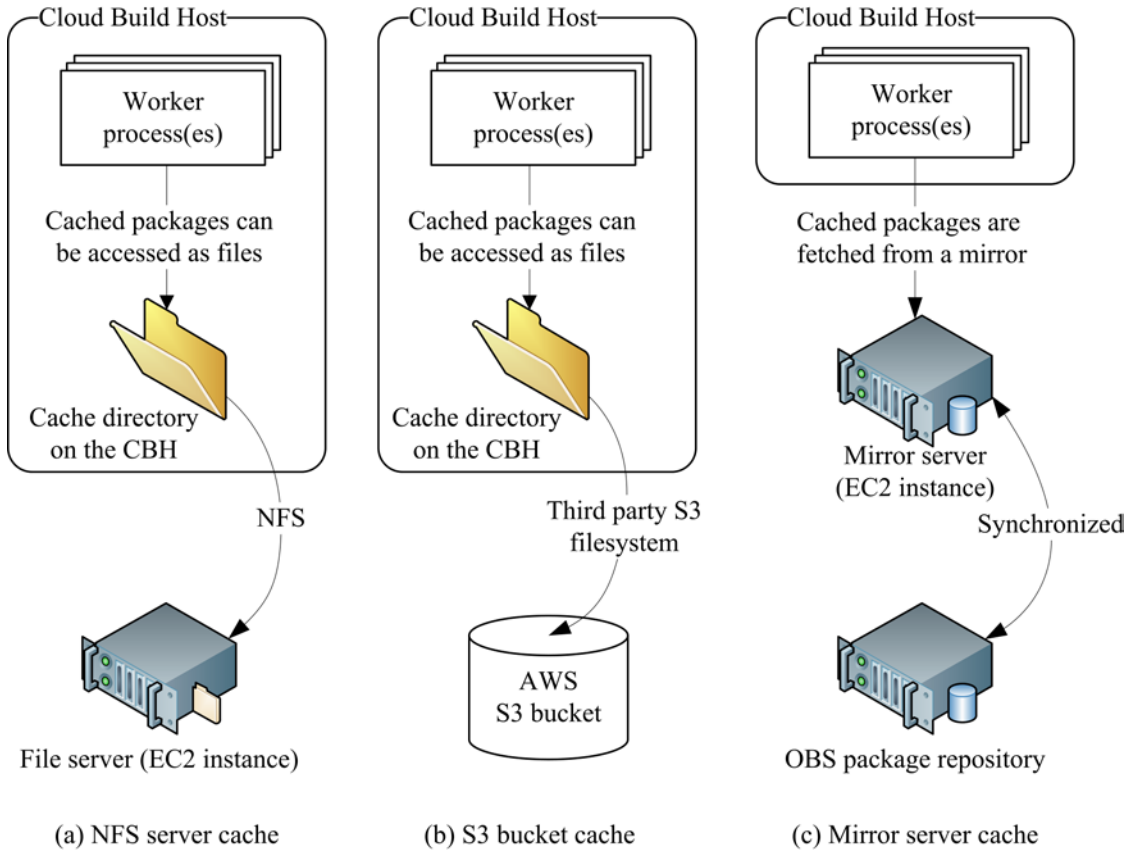

Figure 5.2. *Alternatives for a shared cloud cache*

EBS volumes can only be mounted to a single instance at a time. A shared volume that is attached temporarily to instances is possible but unfeasible to implement, as it would include lots of copying or waiting. Best alternative to create an EBS-based cache is to have a dedicated file server (Figure 5.2.a) and access its disk as a NFS partition from the CBHs. The file server would need to be running whenever a CBH is running, starting when the first CBH starts and shutting down when the last CBH is terminated.

With S3, packages could be cached without the need to run a separate server (Figure 5.2.b). Data is stored and fetched using REST and SOAP interfaces. S3 buckets cannot natively be mounted as POSIX file systems but several 3rd party S3-based file systems have been developed. These include open-source (S3QL, S3backer and several

under the name S3FS) and commercial solutions (SubCloud and ElasticDrive) of which some support mounting the file system in multiple instances at the same time [47].

The third option is to not use the local cache directories on build hosts, but instead bring a mirror of the package repository server closer to the build hosts (Figure 5.2.c). CBHs will download and upload packages to the mirror which synchronizes the package repository with the main OBS server. The mirror server would act as a proxy between the build hosts and the OBS server, similarly to Anttila's research [48]. However, keeping the mirror in sync is not trivial. If other clouds are also used or if the mirror is shut down, updating the mirror in time (before a build job starts in the cloud) with the packages that were built elsewhere is tricky.

As the amount of work needed to build a package varies drastically depending on the package, large bottleneck jobs should be built on dedicated machines using the power hosts feature of OBS. When developing a package that many other packages depend on, it should be temporarily branched to avoid constant unnecessary rebuilds of the depending packages. Building is also very disk write intensive so disk I/O performance should be pursued with RAID striping on LBHs. As the disk performance in AWS is somewhat poor, the build environment on CBHs could be put on a RAM disk: a disk volume that acts as a regular hard drive but is actually located in volatile RAM. The size of the needed RAM disk depends on the size and number of the build environments. The size is determined by the packages and their dependencies that are generally built in the OBS, while the number of build environments equals to the number of workers, which is loosely tied to the amount of processor cores on the CBH. A regular small EC2 instance provides 1.7 GB/core, large and extra-large instances have 3.75 GB/core, while the more expensive high-memory instance types all have 8.55 GB/core [3]. At least some of these instance types should have enough space for building packages completely on volatile RAM.

5.4.3 Ensuring system security

In order to protect the build hosts and the whole build service from attacks originating from the source code being built, the build environments need to be sandboxed. OBS supports the use of virtualization to sandbox the build process, separating the build environment of a worker from the build host. Supported platforms are KVM and Xen.

If no virtualization is used, OBS only uses chroot operation, which is not considered to be secure encapsulation. Chroot is entered with root privileges and packages that the upcoming build requires are installed in the build environment. When the build itself starts, root privileges are dropped. The possible attack vector is to create a package with malicious code in its installation section, and then build something that depends on that package. Chroot can be escaped with root privileges, eventually granting the attacker root privileges on the build host. If a build host is compromised, the attacker could compromise the OBS server and further parts of the network.

EC2 instances are Xen virtual machines. Nesting KVM or Xen environments inside instances is not trivial, and even if nested virtualization could be achieved, the use

of multiple hypervisors degrades performance [49]. A compromise between full-blown virtualization and chroot is operating system -level virtualization. For Linux, these include the Linux Containers (LXC), OpenVZ, FreeVPS and Linux-VServer. LXC uses the kernel of the host for the guests and cgroups kernel feature to separate the user spaces of the guests. Some support for LXC in OBS was developed after this thesis [50].

Some packages may contain proprietary code that cannot be built outside the local cluster, restricting the use of this cloudbursting OBS currently only to building open-source packages. If the cloud provider cannot be relied to maintain confidentiality, an OBS mechanism is needed to restrict building to certain build hosts based on the license of the package. The license type is already stated in spec file of packages. This attribute needs to be checked by the OBS dispatcher to find a suitable build host, similarly as the power hosts and power packages.

5.4.4 Easing disk image configuration

For successful configuration management and more optimal build host image, the image should be created from a description-based template instead of manual editing. Not only does this make updating the image easier, but it also allows to open-source the image, which would further advance the quality of cloud build hosts.

OpenSUSE provides SUSE Studio, a hosted image creation web application with a KIWI-back-end; a tool for creating Linux distribution images. It can be used to create a customized openSUSE image straight to AWS. A more manual approach is to use server configuration management tools like Chef, Cfengine or Etch.

For publishing the image safely, it needs to be cleaned from private information such as logs, credentials, certificates, IP addresses and so on. Amazon has composed several guidelines on how to clean up AMI content before publishing it, and researchers have developed a tool to scan a system for private data [51].

5.5 Related work

A similar effect to cloudbursting could be gained by sharing workers dynamically amongst multiple OBS instances. Sami Anttila has researched resource sharing in OBS and has presented Flexible Worker Pool [48, 52]. In his thesis, several OBS instances offer their workers to be managed by a worker pool master node. This system detaches idle workers from OBS instances where they are not needed, and borrows them temporarily for other instances. This is especially useful if an organization runs several OBS instances to separate workspaces, but the system could also be used in academic and open-source fields with partners that have similar resource usage.

6 CONCLUSIONS

This thesis presented how the infrastructure of a software build system can be distributed into domains that are geographically apart and owned by different organizations. With cloudbursting, this infrastructure dynamically expands as needed, lowering the costs of cloud usage by maximizing the use of acquired resources. Even though cloud services do provide cost-efficient flexibility, they have their challenges in confidentiality, integrity and availability. Infrastructure-as-a-Service customers themselves are responsible for developing their service to failover to other providers.

Whether to keep a system in-house or move it to a cloud depends on many things, and it should be assessed whether cloud usage fits the use case. A hybrid model is a convenient way to keep business-critical data in-house, while buying extra computational power from the cloud. In software building however, the problem is that clear text source code needs to be available during the build process, effectively granting the cloud provider access to the source code. Furthermore the legislation in certain countries demands governmental organizations the possibility to access customer data, even without informing the customers. This broadens the misuse risks of intellectual property. The downside of consumer-accessible infrastructure services is also the modest service level agreement and the lack of an intimate relationship with the provider.

Another aspect to consider is the cost, and it is not simple to estimate. With public infrastructure services, the share of traffic and storage is minimal compared to the cost of computation needed for the actual building. For constant needs, large enterprises that already have the means and knowledge are better off setting up a build cluster of their own. Cloud providers themselves make profit by utilizing economies of scale, amassing servers together to save up in infrastructure and management costs.

Open Build Service is a powerful and flexible tool for building packages for multiple device architectures and target distributions. The main problem in its cloud usage is the dependencies between packages and the great variance in build job times. The balance between the number and performance of workers needs to be optimized. Another issue is that Open Build Service has been designed for closed, trusted environments. It uses virtualization to inhibit attacks from inside of the build environments to the build host and the whole build system. Full-blown virtualization cannot be used in a cloud where build hosts themselves are already virtual machines, and support for lighter virtualization is needed.

The cloudbursting build system setup in this work was a simplified proof-of-concept. With the implemented service manager, the system automatically accommodates to the number of waiting jobs, spawning new build hosts when needed and termi-

nating them when not. While the machines from the cloud are not superior in performance, the fact that there is an seemingly endless pool of them makes the system good in flushing out a long queue of jobs that can be built in parallel. For large bottleneck packages, separate powerful build hosts need to be set up as packages can depend on each other, therefore blocking other build jobs temporarily.

For production-use, however, the system requires more work in terms of security, availability, manageability and performance. Infrastructure and service management software are needed in multi-tenant, multi-service and multi-provider environments. There are several competing alternatives for Virtual Infrastructure Management, but service managers seem to lack options. As services vary greatly, it is difficult to make a generic API for managed services. Currently most services seem to rely on ad hoc service manager implementations.

All in all, cloud services have a definite support role in build systems and other parallel batch job processing systems. Running software services on cloud-compatible platforms make it possible to freely choose the amount of cloud usage with little changes. This kind of decoupling of software services from the underlying operating systems of separate computers can be perceived in modern cloud technologies.

REFERENCES

[1] Armburst, M., Fox, A., Griffith, R., Joseph, A., Katz, R., Konwinski, A., Lee, G., Patterson, D., Rabkin, A., Stoica, I., Zaharia, M. Above the Clouds: A Berkeley View of Cloud Computing. 2009. UCB/EECS-2009-28. 25 p.

[2] Amazon Web Services [WWW]. [Retrieved 2011-03-22]. http://aws.amazon.com/

[3] Amazon Elastic Compute Cloud (Amazon EC2) [WWW]. [Retrieved 2011-03-22]. http://aws.amazon.com/ec2/

[4] Amazon Simple Storage Service (Amazon S3) [WWW]. [Retrieved 2011-03-22]. http://aws.amazon.com/s3/

[5] Amazon Elastic Compute Cloud User Guide: Making Query Requests [WWW]. [Retrieved 2011-08-04]. http://docs.amazonwebservices.com/AWSEC2/latest/UserGuide/available-apis.html

[6] Amazon EC2 Service Level Agreement [WWW]. October 2008 [Retrieved 2011-01-26]. http://aws.amazon.com/ec2-sla/

[7] Summary of the Amazon EC2 and Amazon RDS Service Disruption in the US East Region [WWW]. [Retrieved 2011-05-03]. http://aws.amazon.com/message/65648/

[8] Vijayan, J. Vendors tap into cloud security concerns with new encryption tools. Networkworld [WWW]. 2011-02-10 [Retrieved 2011-06-09]. http://www.networkworld.com/news/2011/021011-vendors-tap-into-cloud-security.html

[9] Thibodeau, P. With WikiLeaks, Amazon shows its power over customers. Computerworld [WWW]. 2010-12-2 [Retrieved 2011-06-09]. http://www.computerworld.com/s/article/9199258/With_WikiLeaks_Amazon_shows_its_power_over_customers

[10] Cloud Standards Wiki [WWW]. 2011-03-04 [Retrieved 2011-03-22]. http://cloud-standards.org

[11] Libcloud [WWW]. [Retrieved 2011-05-05]. http://incubator.apache.org/libcloud/

[12] Deltacloud [WWW]. [Retrieved 2011-05-05]. http://incubator.apache.org/deltacloud/

[13] MacVittie, L. Bursting the Cloud [WWW]. 2008-09-03 [Retrieved 2011-04-24]. http://devcentral.f5.com/weblogs/macvittie/archive/2008/09/03/3584.aspx

[14] MacVittie, L. Cloud Balancing, Cloud Bursting, and Intercloud [WWW]. 2009-07-09 [Retrieved 2011-01-10]. http://devcentral.f5.com/weblogs/macvittie/archive/2009/07/09/cloud-balancing-cloud-bursting-and-intercloud.aspx

[15] Barr, J. Cloudbursting - Hybrid Application Hosting. Amazon Web Services Blog [WWW]. 2008-08-28 [Retrieved 2010-12-22]. http://aws.typepad.com/aws/2008/08/cloudbursting-.html

[16] de Assuncão, M., di Costanzo, A., Buyya, R. Evaluating the Cost-Benefit of Using Cloud Computing to Extend the Capacity of Clusters. Proceedings of the International Symposium on High Performance Distributed Computing (HPDC 2009). ACM, New York, NY, USA, pp. 141-150.

[17] Marshall, P., Keahey, K., Freeman, T., Elastic Site: Using Clouds to Elastically Extend Site Resources. 10th IEEE/ACM International Conference on Cluster, Cloud and Grid Computing, Melbourne, VIC, Australia, May 17-20, 2010. pp. 43-52.

[18] RESERVOIR Reference Architecture [WWW]. 2010-06-08. [Retrieved 2011-02-17]. http://claudia.morfeo-project.org/wiki/index.php/RESERVOIR_Reference_Architecture

[19] Moreno-Vozmediano, R., Montero, R. and Llorente, I. Elastic Management of Cluster-based Services in the Cloud. In Proceedings of the 1st workshop on Automated control for datacenters and clouds (ACDC '09). New York, NY, USA, 2009, ACM. pp. 19-24.

[20] McAllister, N. Virtualization under the hood. InfoWorld Server Virtualization Deep Dive. June 2010, pp. 2-3.

[21] Sempolinski, P., Thain, D. A Comparison and Critique of Eucalyptus, OpenNebula and Nimbus. Cloud Computing Technology and Science (CloudCom), IEEE Se-

cond International Conference, Indianapolis, IN, Nov 30 – Dec 3, 2010. Notre Dame, IN, USA, 2011, Univ. of Notre Dame. pp. 417-426.

[22] Sotomayor, B., Montero, R., Llorente, I., Foster, I., Virtual Infrastructure Management in Private and Hybrid Clouds. IEEE Internet Computing 13(2009)5, pp. 14-22.

[23] Rodero-Merino, L., Vaquero, L., Gil, V., Galán, F., Fontán, J., Montero, R., Llorente, I. From infrastructure delivery to service management in clouds. Future Generation Computer Systems 26(2010)8, pp. 1226-1240.

[24] Hartwork Blog: Reverse package dependency lookup in Debian/Ubuntu [WWW]. 2007-12-24 [Retrieved 2011-09-26]. http://blog.hartwork.org/?p=108

[25] Open Build Service (OBS) [WWW]. [Retrieved 2011-07-19]. http://openbuildservice.org/

[26] openSUSE Build Service [WWW]. [Retrieved 2011-02-08] https://build.opensuse.org/

[27] MeeGo Build Service [WWW]. [Retrieved 2011-02-08]. https://build.meego.com/

[28] Public/Community MeeGo Build Service [WWW]. [Retrieved 2011-02-08]. https://build.pub.meego.com/

[29] openSUSE: Build Service installations [WWW]. 2010-11-10 [Retrieved 2011-02-08] http://en.opensuse.org/openSUSE:Build_Service_installations

[30] openSUSE: Build Service Appliance [WWW]. 2011-08-03 [Retrieved 2011-09-05]. http://en.opensuse.org/openSUSE:Build_Service_Appliance

[31] openSUSE: Build Service Deployment [WWW]. 2010-08-12 [Retrieved 2011-09-06]. http://en.opensuse.org/openSUSE:Build_Service_Deployment

[32] openSUSE: Build Service Clients [WWW]. 2010-04-28 [Retrieved 2011-02-08]. http://en.opensuse.org/openSUSE:Build_Service_Clients

[33] openSUSE: Build Service Tutorial [WWW]. 2011-08-17 [Retrieved 2011-09-07]. http://en.opensuse.org/openSUSE:Build_Service_Tutorial

[34] Haddad, I. Happy 1 Year Anniversary, MeeGo! 2011. The Linux Foundation. 16 p. http://meego.com/sites/all/files/users/u19961/meego_anniversary_article.pdf

[35] MeeGo Build Infrastructure [WWW]. 2011-11-20 [Retrieved 2011-02-17]. http://wiki.meego.com/Build_Infrastructure

[36] MeeGo Release Infrastructure: BOSS [WWW]. 2011-03-21 [Retrieved 2011-04-20]. http://wiki.meego.com/Release_Infrastructure/BOSS

[37] MeeGo Architecture Layer View [WWW]. 2010-10-22 [Retrieved 2011-04-21]. http://meego.com/developers/meego-architecture/meego-architecture-layer-view

[38] MeeGo Wiki: Release Engineering Process [WWW]. 2011-04-26 [Retrieved 2011-09-13]. http://wiki.meego.com/Release_Engineering/Process

[39] MeeGo wiki. 10 easy steps to a local OBS [WWW]. 2011 [Retrieved 2011-05-31]. http://wiki.meego.com/User:Stskeeps/10_easy_steps_to_a_local_OBS

[40] MeeGo wiki. OBS setup openSUSE 11.2 [WWW]. 2011 [Retrieved 2011-05-31]. http://wiki.meego.com/Build_Infrastructure/Sysadmin_Distro/OBS_setup_openSUSE112

[41] Amazon Elastic Compute Cloud User Guide [WWW]. 2011-02-28 [Retrieved 2011-03-29]. http://docs.amazonwebservices.com/AWSEC2/latest/UserGuide/index.html?VMImportPrerequisites.html

[42] boto – Python interface to Amazon Web Services. [WWW]. [Retrieved 2011-07-28]. http://code.google.com/p/boto/

[43] Seppänen, V. Elastic Build Service Manager [WWW]. 2011-06-20 [Retrieved 2011-09-21]. https://github.com/openSUSE/open-build-service/blob/master/dist/elastic_build_service_manager.py

[44] Varia, J. Amazon Web Services Blog: AWS Lowers its Pricing Again! [WWW]. 2011-06-29 [Retrieved 2011-07-04]. http://aws.typepad.com/aws/2011/06/aws-lowers-its-pricing-again-free-inbound-data-transfer-and-lower-outbound-data-transfer-for-all-ser.html

[45] openSUSE Build Service: Reference Guide [WWW]. 2011-04-21 [Retrieved 2011-04-21]. http://doc.opensuse.org/products/draft/OBS/obs-reference-guide_draft/

[46] openSUSE Build Service: Best Practice Guide [WWW]. 2011-04-21 [Retrieved 2011-04-21]. http://doc.opensuse.org/products/draft/OBS/obs-best-practices_draft/

[47] Comparison of S3QL and other S3 file systems [WWW]. 2011-01-03 [Retrieved 2011-08-24]. http://code.google.com/p/s3ql/wiki/other_s3_filesystems

[48] Anttila, S. Resource Sharing for Open Build Service. Master's Thesis. Tampere 2011. Tampere University of Technology.

[49] He, Q. Nested Virtualization on Xen [WWW]. Xen Summit Asia. 2009 [Retrieved 2011-10-02]. http://www.xen.org/files/xensummit_intel09/xensummit-nested-virt.pdf

[50] opensuse-buildservice mailing list: OBS Worker LXC build support [WWW]. 2011-07-01 [Retrieved 2011-07-26]. http://lists.opensuse.org/opensuse-buildservice/2011-07/msg00001.html

[51] Poeppelmann, T., Schneider, T. AMI aiD (AMID) [WWW]. 2011. [Retrieved 2011-08-22]. https://code.google.com/p/amid/

[52] Anttila, S., Geuder, U. OBS Flexible Worker Pool [WWW]. [Retrieved 2011-03-16]. http://mdeb.nomovok.info/trac/wiki/ObsFwp

APPENDIX 1: AWS API EXAMPLE

Example of AWS API where client requests a list of available EC2 regions. Client sends a signed request using boto:

```
POST / HTTP/1.1
Host: ec2.amazonaws.com
Accept-Encoding: identity
Content-Length: 217
Content-Type: application/x-www-form-urlencoded; charset=UTF-8
User-Agent: Boto/2.0b4 (linux2)

AWSAccessKeyId=AKIAIM6TUNVHQMNGWOWA&
Action=DescribeRegions&
SignatureMethod=HmacSHA256&
SignatureVersion=2&
Timestamp=2011-05-30T12%3A51%3A16Z&
Version=2010-08-31&
Signature=vKsjg%2BslGFEeOrISY7DPegAW//b0FzaY7VrW5jHGucg%3D
```

Server replies to request:

```
reply: 'HTTP/1.1 200 OK\r\n'
header: Content-Type: text/xml;charset=UTF-8
header: Transfer-Encoding: chunked
header: Date: Mon, 30 May 2011 12:51:17 GMT
header: Server: AmazonEC2

<?xml version="1.0" encoding="UTF-8"?>
<DescribeRegionsResponse xmlns="http://ec2.amazonaws.com/doc/2010-08-
31/">
 <requestId>88e49e46-ee41-43fa-949c-e57af965608a</requestId>
 <regionInfo>
  <item>
   <regionName>eu-west-1</regionName>
   <regionEndpoint>ec2.eu-west-1.amazonaws.com</regionEndpoint>
  </item>
  <item>
   <regionName>us-east-1</regionName>
   <regionEndpoint>ec2.us-east-1.amazonaws.com</regionEndpoint>
  </item>
      ... other regions removed to save space
 </regionInfo>
</DescribeRegionsResponse>
```

APPENDIX 2: OBS API EXAMPLE

Example of an OBS build status API response which is parsed by the service manager implemented in this work. One of each type of parsed element is shown underlined.

```
<workerstatus clients='23'>
  <idle workerid='pubworker01/1' hostarch='x86_64'/>
  <idle workerid='pubworker01/2' hostarch='x86_64'/>
  <idle workerid='pubworker01/3' hostarch='x86_64'/>
  <idle workerid='pubworker01/4' hostarch='x86_64'/>
  <idle workerid='pubworker01/5' hostarch='x86_64'/>
  <idle workerid='pubworker01/6' hostarch='x86_64'/>
  <idle workerid='pubworker02/1' hostarch='x86_64'/>
  <idle workerid='pubworker02/3' hostarch='x86_64'/>
  <idle workerid='pubworker02/4' hostarch='x86_64'/>
  <idle workerid='pubworker02/5' hostarch='x86_64'/>
  <idle workerid='pubworker02/6' hostarch='x86_64'/>
  <idle workerid='pubworker03/2' hostarch='x86_64'/>
  <idle workerid='pubworker03/3' hostarch='x86_64'/>
  <idle workerid='pubworker03/4' hostarch='x86_64'/>
  <idle workerid='pubworker03/6' hostarch='x86_64'/>
  <idle workerid='pubworker04/1' hostarch='x86_64'/>
  <idle workerid='pubworker04/3' hostarch='x86_64'/>
  <idle workerid='pubworker04/4' hostarch='x86_64'/>
  <building repository='DE_Trunk' arch='i586' pro-
ject='home:arfoll:xbmc-testing' package='xbmc-gles' start-
time='1312907717' workerid='pubworker02/2' hostarch='x86_64'/>
  <building repository='DE_Trunk_Testing' arch='armv8el' pro-
ject='Project:Python:PySide' package='python-qtmobility' start-
time='1312907557' workerid='pubworker03/1' hostarch='x86_64'/>
  <building repository='harmattan' arch='armv7el' pro-
ject='home:kimju:harmattan' package='mg-terminal' start-
time='1312276806' workerid='pubworker03/4' hostarch='x86_64'/>
  <building repository='DE_Trunk' arch='armv8el' pro-
ject='Project:Python:PySide' package='python-qtmobility' start-
time='1312907592' workerid='pubworker03/5' hostarch='x86_64'/>
  <building repository='DE_Trunk' arch='armv8el' pro-
ject='home:arfoll:xbmc-testing' package='xbmc-gles' start-
time='1312907718' workerid='pubworker04/2' hostarch='x86_64'/>
  <waiting jobs='0' arch='armv5el'/>
  <waiting jobs='0' arch='armv7el'/>
  <waiting jobs='0' arch='armv8el'/>
  <waiting jobs='0' arch='i586'/>
  <waiting jobs='0' arch='x86_64'/>
  <blocked jobs='0' arch='armv5el'/>
  <blocked jobs='1' arch='armv7el'/>
  <blocked jobs='3' arch='armv8el'/>
  <blocked jobs='1' arch='i586'/>
  <blocked jobs='0' arch='x86_64'/>
```

… rest of the reply removed as irrelevant.

APPENDIX 3: LIST OF BUILT PACKAGES

Below is an excerpt of OSC build history command. These are the results of a build batch ran from 2011-06-23 05:55:26 to 2011-06-23 06:56:22. Build times are from this single run only and not average times of multiple runs. It should also be noted that LBHs were slightly more powerful than CBHs.

package	build time	worker
bzip2	5m 50s	lbh-00000001/2
cvs	7m 14s	cbh-08808585/1
gawk	7m 14s	cbh-08808583/1
udev	12m 17s	lbh-00000001/1
chkconfig	5m 8s	cbh-08808585/1
curl	6m 54s	cbh-08808583/1
unzip	5m 0s	lbh-00000001/1
git	17m 25s	lbh-00000001/3
flex	12m 33s	lbh-00000001/2
eat	12m 26s	cbh-08809183/1
check	12m 31s	cbh-08809190/1
eject	14m 50s	cbh-08809187/1
cscope	7m 10s	lbh-00000001/2
autotrace	12m 45s	lbh-00000001/3
tar	17m 10s	cbh-08809189/1
ccache	15m 23s	cbh-08809366/1
etherboot	18m 14s	cbh-08809185/1
time	17m 6s	cbh-08809364/1
uuid	18m 3s	cbh-08808585/1
audiofile	17m 51s	cbh-08809367/1
lua	19m 13s	cbh-08809369/1
crashsplash	20m 13s	cbh-08809485/1
monit	15m 45s	cbh-08809547/1
at	12m 56s	cbh-08809965/1
bc	13m 33s	cbh-08809183/1
tasks	20m 56s	cbh-08809546/1
geoclue	16m 18s	cbh-08809190/1
min	29m 59s	cbh-08809370/1
dia	39m 58s	lbh-00000001/1
cmake	43m 42s	cbh-08808583/1