TAMPERE UNIVERSITY OF TECHNOLOGY

SAMI ANTTILA

# RESOURCE SHARING FOR OPEN BUILD SERVICE

Master of Science Thesis

Tarkastaja: Kai Koskimies
Tarkastaja ja aihe hyväksytty
Tieto- ja sähkötekniikan
tiedekuntaneuvoston
kokouksessa 09.03.2011

# TIIVISTELMÄ

TAMPEREEN TEKNILLINEN YLIOPISTO
Tietotekniikan koulutusohjelma
**SAMI ANTTILA: Resource Sharing for Open Build Service**
Diplomityö, 55 sivua, 11 liitesivua
Lokakuu 2011
Pääaine: Ohjelmistotuotanto
Tarkastajat: Kai Koskimies
Avainsanat: hajautetut käännösympäristöt, ohjelmien kääntäminen, paketointi, pilvi, resurssien jakaminen, aikataulutus

Ohjelmistojen automaattista kääntämistä ja paketointia varten on kehitetty käännösjärjestelmiä (esimerkiksi openSUSE:n Open Build Service), jotka jakavat raskaan käännöstyön komponenteittain tai paketeittain usealle koneelle. Tällaisten käännösympäristöjen ongelma on usein se, että vaikka yksittäinen palvelin skaalautuu hyvin käännöstyötä tekeviä koneita lisäämällä, useamman palvelimen välinen resurssien jakaminen ei ole mahdollista. Jokainen järjestelmä on rajoittunut sille staattisesti allokoituihin resursseihin ja useamman palvelimen tapauksessa kyseinen ratkaisu ei ole välttämättä kustannustehokkuuden kannalta optimaalinen. Tämä työ keskittyy kyseisen ongelman ratkaisemiseen erityisesti OBS:n suhteen. OBS-palvelimen tapauksessa käännöstyöt lähetetään erillisille koneille paketeittain, mutta kääntävien koneiden jakaminen usean palvelimen välillä on mahdotonta. Tämä rajoittaa järjestelmän skaalautuvuutta.
Tätä diplomityötä varten tehdyn tutkimuksen tuloksena syntyi joustava resurssien jako ohjelmisto, Flexible Worker Pool (FWP). Ratkaisun ajatuksena oli tarjota erillisille OBS-instansseille palvelu, josta ne pystyisivät varaamaan resursseja käyttöönsä tarvittaessa väliaikaisesti. Tämä toteutettiin suunnittelemalla ja toteuttamalla ohjelmisto, joka toimii dynaamisena välityspalvelimena OBS-palvelimen ja käännöksiä suorittavien asiakaskoneiden välillä. Jotta resurssit pystyttiin jakamaan reilusti, integroitiin järjestelmään erillinen vuorottelija (englanniksi scheduler). Tavoitteena oli kehittää järjestelmä, joka luo dynaamisen ja virtuaalisen FWP:n kontrolloiman OBS-verkkoinfrastruktuurin.
Lopullinen toteutus osoittautui staattista resurssien allokointia paremmaksi ratkaisuksi. Tapauksissa, joissa OBS-palvelimien välillä ei ollut päällekkäisiä käännöstöitä yksittäisen OBS-instanssin saamat resurssit vastasivat kaikkia järjestelmässä tarjolla olevia resursseja. Järjestelmä toimi paremmin myös tapauksissa, joissa jokaisella OBS-palvelimella oli tarvetta suuremmalle määrälle resursseja, vaikkakin etu oli pienempi. Tyypillisessä tapauksessa resurssien tarve vaihtelee suuresti, yleensä purskeittain. Suuren järjestelmän kääntämiseen tarvitaan paljon resursseja, mutta käännöksen valmistuttua resurssit ovat vapaina ja toimettomina joskus pitkiäkin

aikoja. Ratkaisu toimi juuri tällaisissa tilanteissa parhaiten, koska OBS-palvelimilla oli keskimäärin käytössä enemmän resursseja kuin staattisesti allokoiduissa tapauksissa.

# ABSTRACT

TAMPERE UNIVERSITY OF TECHNOLOGY
Master's Degree Programme in Information Technology
**SAMI ANTTILA : Resource Sharing for Open Build Service**
Master of Science Thesis, 55 pages, 11 Appendix pages
October 2011
Major: Software engineering
Examiner: Kai Koskimies
Keywords: distributed build systems, software building, packaging, cloud computing, resource sharing, scheduling

The increase in software system sizes and complexity has created a need for faster and more cost-efficient build systems to be researched. While the actual software building process is rather complex, most parts of it are usually automated by the build systems. Several modern build systems, such as HP's Domain Software Engineering Environment (DSEE) and openSUSE's Open Build Service (OBS), rely on distributing the computationally heavy build tasks to several build hosts equipped with the functionality to compile individual components and finally deliver the resulting software packages to the developers.
The problem with such build systems is that while a single build system can scale well as a singular instance, hosting multiple build systems usually requires separate sets of hardware resources in order to complete the tasks. This thesis concentrates on a specific system, Open Build Service. OBS server dispatches the build jobs to separate build hosts that compile and package the source code provided by the developer. However, these build hosts cannot be shared between OBS instances which limits the scalability of the system.
The research for this thesis led to a solution that is called the Flexible Worker Pool (FWP). The main concept of this solution was to provide separate OBS instances a service that allowed them to obtain additional build resources for the duration of their build jobs when required. This was achieved by designing and implementing a software system that functions as a dynamic proxy server between the OBS server and the dynamically allocated the build hosts. In order to allocate the resources fairly, a scheduler was also implemented within FWP. The goal was to create a virtual OBS network infrastructure that was controlled by FWP.
The final implementation of the Flexible Worker Pool turned out to be superior to the static dedication of build resources. In cases where there were no parallel build jobs on multiple OBS instances, the computational capacity was equal to the whole pool of build resources being dedicated to each of the instances individually. Even in the cases with all OBS instances being loaded with near to constant need for build resources, the FWP solution functioned more efficiently. Due to the bursty nature

of the need for build resources, the advantage of sharing the resources was often significant and the solution cost-effective as the same amount of hardware could now support more OBS instances.

# PREFACE

First off, I would like to thank the University for the proficiency it has provided me during these last 9, or so, years. Writing software has been my passion since I was a little kid and being able to design and implement these new systems with a skill-set of a professional is an unrivaled privilege.

I would also like to thank my whole family for the support I have received throughout my life. I would like to give special thanks to my parents who patiently pushed me towards graduation. I would also like to thank my supervising professor Kai Koskimies, my co-workers Uwe Geuder and Miko Nieminen; This thesis, and causally my graduation, would not have been possible without their significant personal efforts.

Furthermore, I would also like to thank Santtu Lakkala for practically pushing me into this opportunity - Thanks mate!

*"Work smarter, not harder."*
-Sami Anttila, 2011

# CONTENTS

# TERMS AND ABBREVIATIONS

| | |
|---|---|
| AUI | Administrative User Interface. The main interface for managing the flexible worker pool |
| API | Application user interface. |
| AWS | Amazon Web Service. |
| Build code | Build code is the source code the worker uses to build the the package. |
| CLI | Command line interface. |
| cpio | A binary file archiver and an archive file format. |
| FWP | Flexible worker pool is a resource sharing service application for Open Build Services |
| Handover | The process of changing the allocation of a worker from OBS to another. |
| Hard handover | Manual re-allocation of a worker through configuration modification and process restarting. |
| HTTP | Hypertext Transfer Protocol. |
| HTTPS | Secure HTTP. |
| IaaS | Infrastructure as a Service. |
| OBS | Open Build Service is an open and complete distribution platform maintained by OpenSUSE organization |
| Proc | *Proc* objects in Ruby are blocks of code bound to a set of local variables |
| PaaS | Platform as a Service. |
| Rails | Web-based application framework built on Ruby. Often referred to as Ruby on Rails. |
| REST | Representational State Transfer. |
| RPM | RedHat Packaging Manager. Also a package format. |
| Ruby | Ruby programming language. |
| SaaS | Software as a Service. |
| SOAP | Simple Object Access Protocol. |
| Soft handover | Handover performed through the worker's interface without changing the actual configuration. |
| osc | Open Build Service command line interface tool. |
| Worker | An instance of the OBS performing the actual build job. |
| Worker host | Machine hosting one or more worker instances. |

| | |
|---|---|
| Worker code | Worker code is the source code of the worker that is required to run a worker. |
| WPIM | Worker Pool Infrastructure Module. |
| WPM | Worker pool master is the main part of the back-end of the Flexible Worker Pool application. |
| XML | Extensible Markup Language. |

# 1. INTRODUCTION

This thesis describes the development process and solutions behind the *Open Build Service* (OBS) build resource sharing application called the *Flexible Worker Pool* (FWP).

Increasing software system sizes and complexity have created a need for more efficient software building methods to be researched. Compiling and building software packages is a computationally demanding task. Several build tools have been developed to optimize the process and most system building is automated in modern software development. Besides automation, modern build systems often rely on distributing system build tasks to several build hosts to complete the build jobs in a timely fashion. Such build systems scale quite well since the computational capacity can be increased by simply adding hardware resulting in faster and more cost-effective build systems. Next logical step in improving the process is sharing build resources between multiple build systems. The need for build resources is often very bursty in the sense that either a lot of resources are needed or none are required. Being able to utilize the idle times of these resources in other, parallel, build systems would result in more cost-effective systems with less hardware.

Open Build Service (OBS) is an open source based distribution development platform that can be used to build software packages against numerous Linux distribution targets and hardware platforms. A single build service instance consists of a server and the build clients, called *workers*, that commence the actual system building. Each worker is dedicated to a single instance of the OBS and cannot be shared with other instances.

While the static nature of the OBS to worker relationship is not a problem in a typical setting, it limits the scalability of multiple independent OBS instances that could benefit from a shared set of hardware resources. To address this problem, the *Flexible Worker Pool* was designed and implemented to serve as a hardware resource sharing service. The main outline of the concept was to be able to dynamically share the workers between independent OBS instances for the duration of the build jobs. The ideas presented in this research resemble the ideas from cloud computing. OBS servers do not have information of the available resources within the Flexible Worker Pool other than the possible deals with the service provider.

Chapter 2 introduces some background information related to the implemented system. This background includes information about building software, configuration management and packaging as well as brief introduction to the history, architecture and methodology of cloud computing. It also describes a related system that allows build systems to utilize cloud computing resources. Chapter 3 outlines the Open Build Service architecture and functionality. The main focus of this section is on the architecture and communication model of the OBS because those were the most important factors for the Flexible Worker Pool design process. Chapter 4 describes the concept, motivation and requirements for the Flexible Worker Pool that was implemented as a part of the project. Chapter 5 describes the design principles, the communication model and the software architecture of the implemented Flexible Worker Pool solution in detail. Chapter 6 analyzes the performance of the Flexible Worker Pool as a resource sharing medium compared to individual OBS instances running on varying static hardware configurations. Chapter 7 draws the conclusions based on the actualized functionality and efficiency of the Flexible Worker Pool system compared to the original solution.

# 2. BACKGROUND

This chapter examines the background of relevant concepts to the research. Section 2.1 examines relevant system building concepts and tools as well as potential issues. Section 2.2 introduces cloud computing concept along with advantages and challenges it creates.

## 2.1 System building

This thesis concentrates on improving an existing system building solution, it is important to understand the underlying concept behind the process. The following sub-sections outline the system building process as a whole and introduce some of the tools used in it.

### 2.1.1 System building process

An essential part of any bigger software development process is system building. In this process, the different components of a software system are collected and (usually) compiled and linked into a working executable and distributable program package. The resulting package can then be installed and executed on the target machine. Since the compiled packages are typically architecture (such as i586 or ARM) and operating system (for example Windows or Linux) dependent, the built packages can generally only be used on the system architecture it was built for. It should be noted, however, that the system building process can take place on a different architecture than the build target architecture. This is called cross-compilation. It can create problems that are difficult to approach and debug and might require a complete redo of the system build to correct.

Furthermore, system building is a computationally demanding task. Depending on the size and complexity of the system, it can take from minutes to days to compile a system completely. For this reason, additional methods of resource division have been developed for building sizable systems. These methods often revolve around the idea of distributing the computationally heavy compiling tasks to multiple host machines.

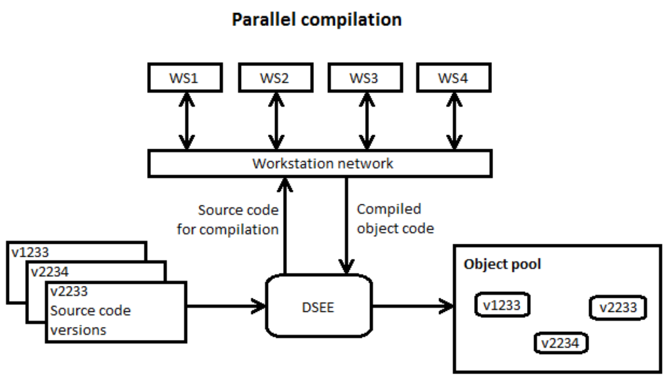


Figure 2.1: Network-oriented system building[28].

### 2.1.2 System building tools

Given the complex nature of system building, several tools have been created to easen the process of creating and maintaining builds. These systems can reduce the possibility of human errors and speed up the whole process by, for example, minimizing re-compilation if no changes have been made to the component. [28]

An example of such a tool would be the most commonly used build tool *Make* [9]. Make keeps track of compiled components and their dependencies based on a *makefile* created by the developer. While Make has its limitations when building bigger systems, it is often used in component level compiling.

### 2.1.3 Network-oriented system building tools

Network-oriented system building tools, much like the traditional ones, are used to compile and possibly package software. With the network-oriented systems, however, the focus is usually on distributing the computationally demanding building tasks to external workstations. An example of such a build system is Hewlett-Packard's *Domain software engineering environment*[15] (DSEE). DSEE's hardware architecture is illustrated in Figure 2.1.

On a grander scale of things, software and software building can both rely on other programs or libraries existing on the build system environment. While the typical build tools generally notice the absence of these components, they do not necessarily try to compile them. Such is the case with previously mentioned program Make. Network-oriented systems can provide the build environment for the programmer while the programmer provides the source code for the software and required information to compile it. Such systems are often referred to as *build services*.

One advantage of such services is that they can be integrated with version and configuration management systems. Build services can potentially be used to compile same software for multiple architectures and the different versions can be maintained in data storages within the build service. Network-oriented system building also supports distributed software development as the build systems can be accessed via networks.

## 2.2 Cloud computing

While cloud computing is not the primary subject of the thesis, the researched system implementation included several similar traits. The following sub-sections describe cloud computing briefly and introduce some advantages and related problems in cloud systems.

### 2.2.1 Overview

National Institute of Standards and Technology (NIST) defines *cloud computing* as a model for enabling ubiquitous, convenient, on-demand network access to a shared pool of configurable computing resources such as networks, servers, storage space, applications and services that can be rapidly provisioned and released with minimal management effort or service provider interaction[22].

While the underlying concept of cloud computing dates back to the 1960s, the first modern implementation was launched in 2006 by Amazon. Amazon Web Service [1] (AWS) offers remote computing services available over HTTP using REST[5] and SOAP[33] protocols.

NIST outlines five key characteristics for cloud services: on-demand self-service, broad network access, resource pooling, rapid elasticity and measured service[22]. In other words, the clients can dynamically receive a proper amount of computing capabilities through the network without human interaction and without the exact knowledge of where the resources come from other than from within the cloud. The cloud concept is illustrated in Figure 2.2.

Furthermore, for the consumer, the resources within a cloud can be virtually unlimited as the cloud systems are supposedly very scalable and thus, the limitations are mainly related to the amount of hardware in the cloud. Cloud systems also automatically control the use of resources by using some sort of metric such as bandwidth, storage space or processing power. Figure 2.2 illustrates the general concept of cloud computing by the different application layers.

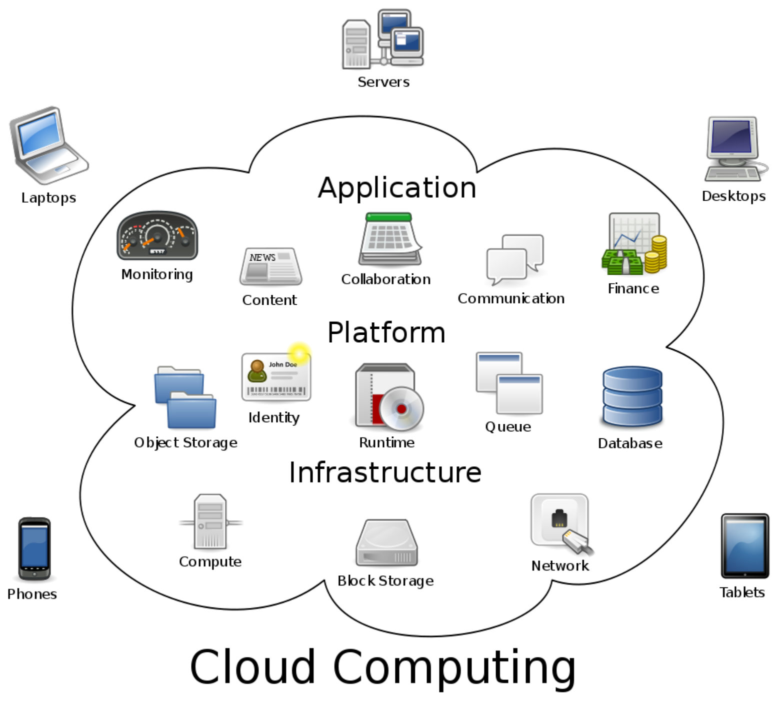


Figure 2.2: Cloud computing overview[13].

The size and infrastructure of the cloud is controlled by the deployment model used and can vary from small internal cloud services to ones open to the general public. The type of the cloud is vastly effected by the needs of the community or organization hosting it. The underlying idea is that cloud systems should be well scalable to any necessary capacity.

### 2.2.2 Cloud systems in practice

Cloud systems can be divided to three sub-categories based on the nature of their service models: Software as a Service (SaaS), Platform as a Service (PaaS) and Infrastructure as a Service (IaaS)[22].

Software as a Service
: Cloud systems offering software as a service grant the client software access to applications running within the cloud. The client cannot install additional software nor modify the underlying infrastructure of the system.

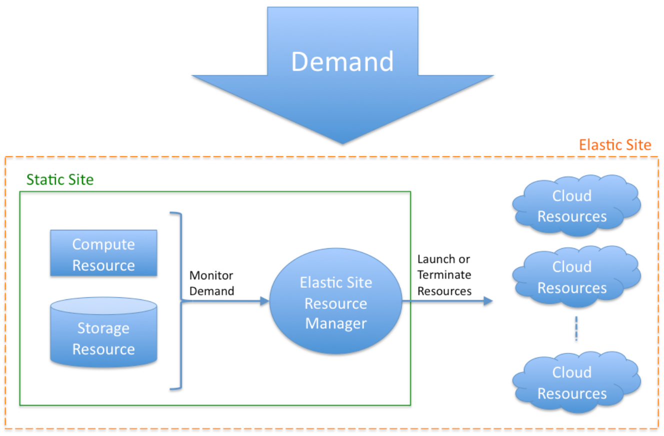


Figure 2.3: Elastic build site structure[16].

Platform as a Service

Platform as a service offers an access point to an existing software infrastructure for the client to deploy arbitrary applications. In such a case, the client does not have control over the infrastructure and is typically restricted to running applications that are created using the programming languages and tools supported by the service provider.

Infrastructure as a Service

Cloud systems can also offer infrastructure as a service; essentially giving the client control over the portion of computational resources. In practice, clients have control over the storage space and can configure their own software environment including the operating system and the applications. While the level of control over resources such as host firewalls can vary, IaaS does not grant the client any control over the underlying cloud infrastructure.

Typically, the client requests resources from the cloud, but has no knowledge of the actual source of these resources. In other words, the internal infrastructure of the cloud is not visible to the client. One prevalent goal of this approach is to simplify the problem from the client's point of view. If resources can be received on request, the client does not have to know where they are coming from. This divides the problem into two smaller problems: management of the cloud infrastructure (handled by the service provider) and management of the client application (handled by the developer). Figure 2.3 illustrates the on-demand resource allocation model.

From the service provider's point of view, cloud systems also create some issues that have to be addressed. These issues include privacy issues, performance issues and legal issues. The legal issues are not in the scope of this thesis.

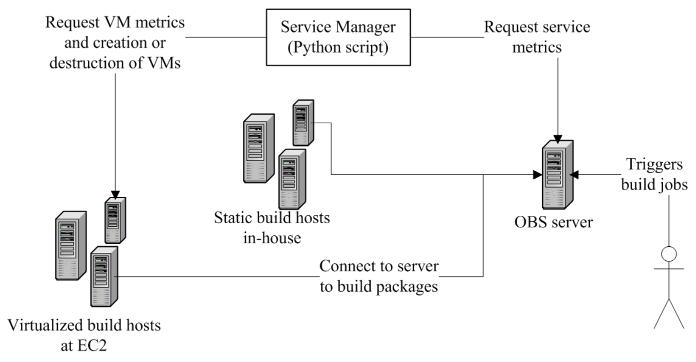

Request VM metrics and creation or destruction of VMs
Service Manager (Python script)
Request service metrics
Static build hosts in-house
OBS server
Triggers build jobs
Connect to server to build packages
Virtualized build hosts at EC2

Figure 2.4: Elastic build system for OBS[26].

Because of the nature of the resource sharing, an issue of privacy is very prevalent. The service provider has to be able to guarantee (depending on the case) a level of security and privacy over the resources used since having an external cloud means that the service provider also has access to the information within the shared resources. Furthermore, the service provider has to be able to guarantee that other users using the resources in the cloud cannot access the resources of other clients.[36]

Performance issues are also a direct result of running large distributed systems offering services to arbitrary amount of clients simultaneously. In most cases, the networking bandwidth is the limiting factor as the actual storage and processing power resources can be extended simply by adding hardware. It should be noted that many cloud systems do not offer individual host machines for use directly, but grant access to virtual machines running within a different and hidden infrastructure. In such cases the issue of managing the resources becomes more complex.

## 2.3 Cloud computing and build services

The next step in utilizing computational power more efficiently is the expansion of the network oriented build infrastructure to use cloud and cloud-like solutions. Since the build services' build efficiency is mostly limited by the amount of hardware, accessing additional resources in the cloud should be able to improve the effectiveness of the build system.

Ville Seppänen researched the use of cloud computing with Open Build Service in his Master's Thesis *Elastic Build System in a Hybrid Cloud Environment*[26]. The research takes an approach where it concentrates on utilizing Amazon's pre-existing Cloud Service, *Amazon Web Services* (AWS), in a hybrid cloud infrastructure to reserve an external infrastructure for the deployment and the usage of workers based on demand as illustrated in Figure 2.4. Reserving the infrastructure from the cloud dynamically based on a temporary need is called cloud bursting. The underline concept is that multiple OBS instances dynamically allocate the workers based on the need and can potentially reserve the same worker hosts. Such systems create new challenges related to reliability, security and cost-efficiency, but offer promising possibilities in terms of system scaling.

# 3. OPEN BUILD SERVICE

This chapter gives a brief explanation of the main functionality and implementation of the *Open Build Service* (OBS). OBS is an open and complete distribution platform maintained by OpenSUSE organization. It provides the infrastructure for creating and releasing open source software for numerous Linux distributions on different hardware architectures. [20]

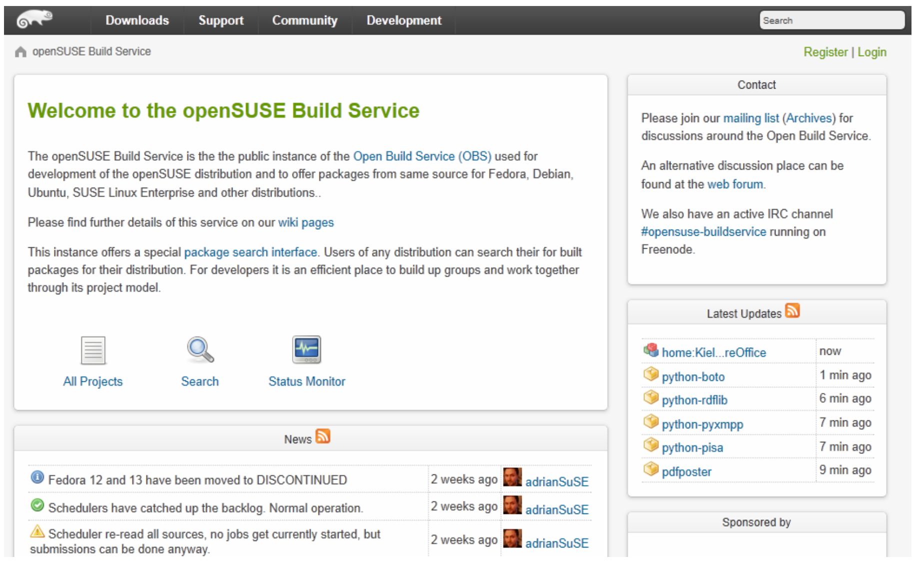
## 3.1 Functionality

OBS is an open source application and it is released under the GPL license[8]. It is being used by numerous Open Source projects such as MeeGo[31]. The biggest single instance of the OBS is currently running at Novell servers at http://build.opensuse.org/[20] with almost 30,000 developers working on more than 20,000 projects which consist of over 150,000 packages in 27,000 repositories.

The main function of OBS for software developers is configuring, building and publishing *packages*. A *project* is an aggregate of packages with additional metadata. Packages are automatically compiled by the OBS from the set of source files provided by the developer. The actual compiling can be done for different architectures and distributions without the need for external compiler farms. The developers also have the option to work in groups on the projects.

Building features include automatic dependency resolving and linking to other projects. Should a package depend on another package, those will also be compiled in the building process. Changes in these depended packages will also trigger a rebuild in the package depending on it. Such functionality makes it possible to test patches against packages from other projects. After a completed build, the resulting packages are published in separate repositories that can be accessed by the users.

OBS can also serve as a software distribution platform for normal users, such as independent developers or developer groups. Users can access the newest version of software for their distribution directly through the HTTP user interface. This availability can further be boosted by using mirror servers.



Welcome to the openSUSE Build Service

The openSUSE Build Service is the the public instance of the Open Build Service (OBS) used for development of the openSUSE distribution and to offer packages from same source for Fedora, Debian, Ubuntu, SUSE Linux Enterprise and other distributions..

Please find further details of this service on our wiki pages

This instance offers a special package search interface. Users of any distribution can search their for built packages for their distribution. For developers it is an efficient place to build up groups and work together through its project model.

News

| | | |
|---|---|---|
| Fedora 12 and 13 have been moved to DISCONTINUED | 2 weeks ago | adrianSuSE |
| Schedulers have catched up the backlog. Normal operation. | 2 weeks ago | adrianSuSE |
| Scheduler re-read all sources, no jobs get currently started, but submissions can be done anyway. | 2 weeks ago | adrianSuSE |

Contact

Please join our mailing list (Archives) for discussions around the Open Build Service.

An alternative discussion place can be found at the web forum.

We also have an active IRC channel #opensuse-buildservice running on Freenode.

Latest Updates

| | |
|---|---|
| home:Kiel...reOffice | now |
| python-boto | 1 min ago |
| python-rdflib | 6 min ago |
| python-pyxmpp | 7 min ago |
| python-pisa | 7 min ago |
| pdfposter | 9 min ago |

Sponsored by

Figure 3.1: OpenSUSE Build Service instance at http://build.opensuse.org/[20]

OBS provides integration with OpenSUSE KIWI Image System[21] for automatic product and image creation. This allows users to compile their software into appliances with just enough operating system components to run the software. The appliance images can be created to be ran from USB sticks, live CDs or external hard drives.

OBS offers an external HTTP based interface. OBS, as a default, offers a web-based client interface illustrated in Figure 3.1 and a command line interface (CLI) called *osc*. The interface is open and can also be accessed via third party applications and use its resources for their benefit.

## 3.2 Architecture overview

OBS server can be architecturally divided into a *front-end* and *back-end*. Front-end consists of the API for different user interface applications. It is typically accessed via web-based user interface application or the CLI that provide user friendly access to the resources provided to it by the back-end.

Back-end hosts repositories and sources of the projects and their packages and manages the building process and scheduling related to it. It also maintains the status information of the build clients, usually referred to as *workers*. The main components of the OBS back-end are the source server, the repository server and the pool of workers, generally referred to as the *worker pool*, that consists of one or more build clients.

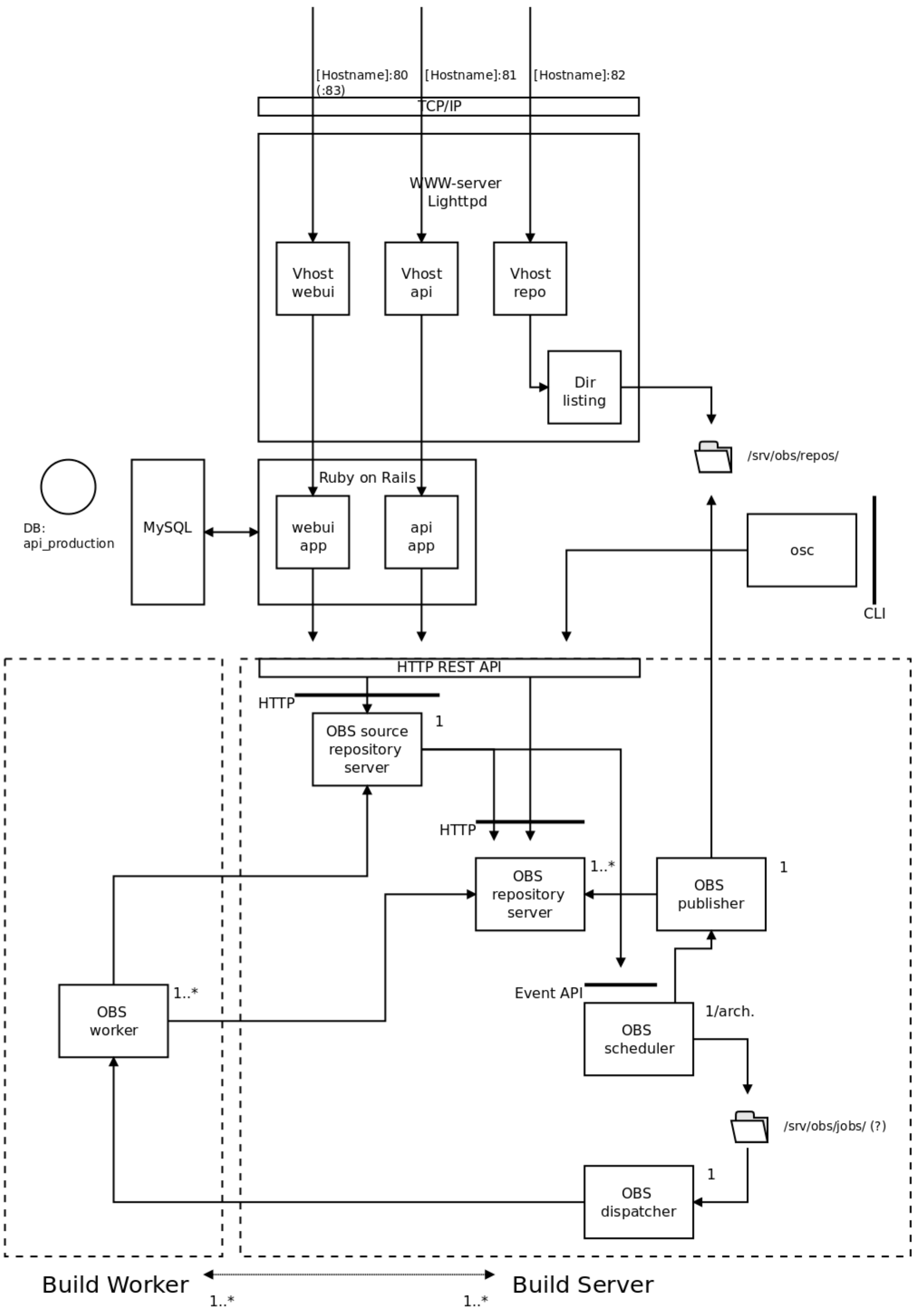


Figure 3.2: OBS Internal Architecture [30]

Another architectural line can be drawn between the OBS back-end server processes and the worker pool. While the workers commence build jobs for the repository server, they are still separate instances, often (but not necessarily always) running on separate hosts. A single OBS instance is very scalable as the amount of workers can be increased by adding hardware.

Figure 3.2 illustrates the architecture inside the OBS. The first boxed part at the top, consists of the web server that is used to forward requests to OBS. The second box, titled "Ruby on Rails", includes the web-based user interface (on the left) and the front-end API application on the right. On the bottom-left of the picture are the worker instances. Both of these are connected to the OBS back-end. The front-end connects to it through a HTTP REST API, while the workers use their own RPC calls to communicate with the back-end. The internal connections between back-end parts are discussed in better detail in the following sub-sections.

Individual parts of the OBS are implemented in different programming languages. The back-end is implemented in Perl, while the front-end and web user interface are implemented in Ruby[18] using Rails web-development framework[11]. The command line interface client, *osc*, is implemented in Python[23].

## 3.3 OBS components

As mentioned in Section 3.2, OBS architecture can be divided into two main components: the back-end and the front-end. The back-end can further be broken down into a smaller sub-set of components, each running in their individual processes. In principle, the back-end serves the front-end with the actual functionality of the build service and the workers with the necessary functionality required for acquiring build essentials.

Each OBS instance consists of one or more *repository servers*. The repository server provides access to the binary repositories of the defined projects, packages and architectures. Furthermore, it functions as a gateway for many other functions of the OBS. It also forwards some events to the source server. The repository server also maintains the list of workers registered to it (the worker pool) and informs the scheduler of any finished jobs by those workers.

It is noteworthy that in most cases, the OBS configuration has only one repository server. At the time of this research, the support for multiple repository servers was still in development and OBS instances were thus limited to a single repository server instance.

For each OBS instance, there exists a single *source server*. It manages the source codes, revisions, project/package metadata, submit requests, etc. that originate from the users of the OBS. In a typical configuration, it runs on the same host as the repository server.

*Dispatcher* handles the dispatching of the build jobs to the available workers. The queued build jobs are stored in the file system as XML[34] files by the scheduler. Similar convention is used for storing the worker information. Dispatcher sends the build jobs as XML through the worker's API.

*Schedulers* maintain the projects and packages for their corresponding repository servers. Individual scheduler instances are required for each architecture supported by the OBS instance. Schedulers are responsible for starting the build jobs in the correct order as well as collecting binary packages and creating external repositories in case the project's repository is already completely built. However, schedulers do not actually send any commands to the workers, but store the queue in the file system as XML files for the dispatcher that will send the whole build job objects to the worker API. Scheduler also uses all the information about finished build jobs for dependency recalculation and creates publish events (in the filesystem) for the publisher.

*Signer* process runs in its individual process and signs the built packages. It is not a necessity but is often recommended.

*Publisher* handles the publish events generated by the scheduler. It moves the packages from projects to the corresponding repository trees and generates the related metadata. Publisher then uploads the content to the download servers where it is available for the users to download.

*Warden* is an optional process that monitors running Worker instances and kills running jobs if the workers die without sending out a clean shutdown message.

*Build clients*, also known as *workers*, are processes that build the actual packages for the OBS. Workers often reside on separate hosts. A machine that hosts one or more worker processes is called a *worker host*.

The workers request the repository server for *worker code* upon start up. Worker code is the executable source code of the actual running worker process that listens to a socket and waits for commands from the repository server. Worker code is identified by hash created from the source code and it is checked each time the worker receives a build job. Differing build code versions will cause the worker to update its worker code to the version provided by the OBS and restart the instance.

The worker code implements a HTTP-based API that receives the build requests from the repository servers as well as status requests, log requests and potential build interruption requests. However, it does not actually perform the build process.

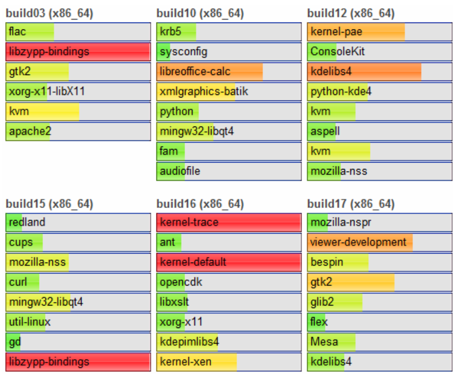


Figure 3.3: Some worker hosts building packages [20].

A worker will receive its *build code* from the repository server upon starting a build after the worker code has been updated and the worker has been restarted. This behavior allows easy updating of the workers for newer versions without the need to individually perform updates. It also makes the workers more flexible if being moved from one OBS instance to another. All worker versions are not necessarily guaranteed to be compatible, but it should be the case for most versions.

Figure 3.3 illustrates a random set of OBS workers in action. Each separate group of workers represent a worker host machine (i.e. *build03*, *build10*, *build12*). The list also displays the worker host's architecture next to the name. Each worker host can host an arbitrary number of workers that can be building concurrently. In most cases, however, the amount of workers is set (but not limited) to the amount of cores available on the build machine for maximum computational capacity utilization.

After receiving a build job and the appropriate build code, the worker commences the build process. It will inform the repository server that its status is now building. The actual build process is not in the scope of this thesis.

After a completed build, the resulting binaries will be sent back to the OBS repository server in a package that will store them for later user access.

*Source service server* is a service for source code processing. It hosts services such as source RPM[24] extraction, file integrity verification and package's version metadata updating.

OBS *front-end service* provides an interface and access control logic for the external interface applications. The provided interface works either over HTTP or HTTPS protocol, depending on the setup.

The current installment of the OBS includes two interface options for accessing the front-end: The web-based user interface and a command line interface (osc). Front-end is not examined in more detail since the emphasis of this research is on the internal communication of the OBS back-end.

| | Port |
|---|---|
| **Web interface** | 80 |
| **Front-end** | 81 |
| **Repository server** | 5252 |
| **Source server** | 5352 |
| **Source service server** | 5152 |
| **Direct access to built packages** | 82 |

Table 3.1: Defaults inbound ports for OBS

## 3.4 Communication model

The communication between the separate parts of the OBS is a very complex and wide area to cover. The aim of this section is to outline the main communication model and only examine the specifics of the parts relevant to this research.

The internal communication between the sub-processes of the repository server is handled through file-based events. In practice this means storing XML-files with the event data into the file system at designated locations for the other modules to pick them up for processing.

The communication between the running main processes (front-end, repository server and source server) is accomplished by using a HTTP- or HTTPS-based interface. Each of the three instances reserve an inbound port. The defaults ports are listed in Table 3.1.

Source service server port is listed below the line because it is not relevant in this research, nor is it a necessity for running an OBS instance.

Notice the last port in Table 3.1 is not directly related of any of the parts above it, but merely a link to a directory listing which is open by default. The actual functionality is implemented by the underlying web-server (often lighttpd[14]). This port is likely to be closed in most open instances.

The communication between the workers and the repository server is a complicated, but relevant subject for this thesis. Both the worker and the repository server instances implement a simplistic HTTP server with the workers' server code being simpler than repository server's. This means that they both work on a request-response basis and will stand-by for incoming commands.

Upon initialization, the workers first load the repository server's address configuration from *buildhost.config* file. It then proceeds to request the repository server for a worker code through a */getworkercode* call. The worker code is returned in chunked HTTP response consisting of cpio [7] encoded data that of the files the worker process needs to run. After successfully receiving the code, the worker host restarts the workers in order to take the new runtime code into use as described in Section 3.3.

| | |
|---|---|
| IDLE | The worker is idle and waiting for a command from the repository server. |
| BUILDING | The worker is already building a package. The information can be obtained with the */info* command. |
| REBOOTING | The worker has been issued a reboot command. Worker will be non-receptive to new commands during this state. |
| KILLED | The worker has been issued a completed kill order. |
| DISCARDED | The worker's job has been discarded. |
| BROKEN | The worker is broken and cannot be used. |

Table 3.2: Worker state message types

After initializing the worker code, the worker starts a state message sending loop that will send the worker's current state in XML format, as illustrated below, every 5 minutes to the repository server.

```
<worker hostarch="i686" ip="127.0.0.1" port="48417" workerid="w/1" />
```

The message contains the information of the worker's native architecture, ip address, port and worker identifier. The identifier consists of the name of the worker (in the example 'w') coupled with the index of the instance. This means that if the worker host is running multiple worker processes, they will be consequently named 'w/1', 'w/2', ..., 'w/$n$' where $n$ is the amount of worker instances running on the worker host.

The repository server maintains a list of the states of all of the workers dedicated to it and accepts new workers to the worker pool based on these state messages. The possible worker state message types are listed in Table 3.2.

Each worker instance listens to a port randomized upon each restart of the worker instance. Notice that a worker receiving a build job that requires worker code to be updated will also restart the worker process, giving it a new port.

OBS will build packages based on their update status. In other words, when a user updates the contents of a package, it will trigger a (re)build. The scheduler will schedule the build jobs by storing an XML-file with the configuration of each build under its configured job queue directory. A stripped example of such XML-file is illustrated in Program 3.1.

The dispatcher will then find the newly created job files and dispatch them as remote procedure calls (RPC) through the workers' HTTP interface. This will trigger the workers to fetch the proper build code and commence the actual building.

The build process starts by the worker downloading the necessary binaries from the repository server for compiling the package. It also figures out dependencies of the package and builds them in a clean environment as well.

```
<buildinfo project="home:Admin" repository="openSUSE_11.4"
           package="bzip2" srcserver="http://localhost.domain.fi:5352"
           reposerver="http://localhost.domain.fi:5252">
  <job>home:Admin::openSUSE_11.4::bzip2-4ee301d40b1a326195</job>
  <arch>i586</arch>
  <srcmd5>4ee3010f73fa5c72aafad40b1a326195</srcmd5>
  <verifymd5>4ee3010f73fa5c72aafad40b1a326195</verifymd5>
  <rev>12</rev>
  <reason>new build</reason>
  <needed>0</needed>
  <revtime>1308668005</revtime>
  <readytime>1316444845</readytime>
  <file>bzip2.spec</file>
  <versrel>1.0.5-47</versrel>
  <bcnt>14</bcnt>
  <release>47.14</release>
  <subpack>bzip2-doc</subpack>
  ... more sub-packs ...

  <bdep name="aaa_base" preinstall="1" runscripts="1" notmeta="1" />
  ... more dependencies ...

  <path project="home:Admin" repository="openSUSE_11.4"
        server="http://localhost.domain.fi:5252" />
  <path project="openSUSE.org:openSUSE:11.4" repository="standard"
        server="http://localhost.domain.fi:5352" />
</buildinfo>
```

Program 3.1: An example of build job XML-file (HTTP headers omitted)

The worker transmits its current state to the repository server(s) it was initially configured to service. However, upon receiving a build job, the worker system bypasses the original worker configuration and uses the configuration received in the build job message XML. Such configuration includes the repository server and source server addresses that can be seen in Program 3.1. This means that, in theory, the scheduler could choose to redirect the resulting package to an arbitrary repository server instead of the one it originates from.

# 4. REQUIREMENTS FOR THE SYSTEM

This chapter examines the problem of build system scaling when there are multiple OBS instances involved. Each OBS requires a dedicated set of workers that are idle most of the time while the need for computational resources can often surpass the capacity when it is needed. In order to utilize the resources better, this chapter introduces the concept and the requirements for the proposed solution, the *Flexible Worker Pool* (FWP).

Section 4.1 examines the motivation behind the research in more detail and the system implemented for this project. Section 4.2 describes the concept of the implemented Flexible Worker Pool system. Section 4.3 lists and provides an explanation for individual requirements set for the implemented software.

## 4.1 Motivation

Every OBS instance requires one or more workers in order to build packages for the users as described in Chapter 3. These workers are collectively called the worker pool. A single static worker pool with a set of dedicated workers is typically sufficient for any single OBS instance.

Two small setups are illustrated in Figure 4.1. The setup on the left consists of 3 workers and one OBS server instance. Such setup is very scalable since the worker count can be increased by simply adding hardware.

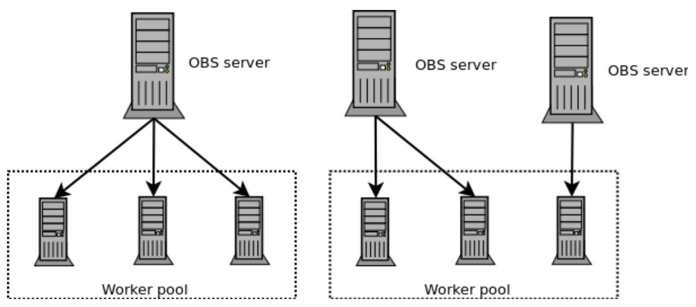


Figure 4.1: Two typical small OBS hardware configurations

Even though a single OBS instance can host virtually limitless amount of workers making it very scalable, multiple OBS instances cannot share workers with each other in the current software architecture so the setup would look something like the right-side configuration in Figure 4.1. In most OBS instances, the idle times for individual workers are relatively high. Therefore hosting multiple OBS instances with dedicated worker hosts for each instance can quickly become unpractical and expensive. This can be the case, for example, in a corporate setting where separate project groups or clients might require their separate OBS instances. Since all the projects within a repository server are, by default, visible to all the users of the OBS, it becomes difficult to draw managerial lines between the different subsets of users. This could be a problem for individual company clients who wish to develop their software in a secure and isolated environment. Having their projects exposed to other users would make such work impossible.

Being able to share the workers dynamically and securely between OBS instances based on the current need for computational power would reduce the cost of hardware required for running multiple OBS instances. It would also make the system more adaptive to changes in the OBS utilization rates. There are indicators in the OBS source code and documentation that sharing workers has been thought of but never implemented.

Such approach resembles the ideas of cloud computing presented in Section 2.2 in the sense that computational resources are requested from an external service without prior knowledge to whether there are any available. Allocating build clients in this manner would be more flexible than the current solution described in Chapter 3.

## 4.2 Concept

The *Flexible Worker Pool* (FWP) concept introduces the idea of a dynamic worker pool from which individual OBS servers can reserve workers for the duration of their build jobs. The key idea of this concept is that worker allocation should be completely dynamic and sharing workers should not make the OBS instances using the FWP aware of each other. This unawareness requirement creates some security issues that need to be addressed. These issues are listed in detail as requirements in Section 4.3.

The underlying idea is that the FWP server functions as an access point to the hardware resources, namely the workers as illustrated in Figure 4.2. The workers acquire the correct build code from the OBS server automatically and build the package which can then be delivered back to the front-end client application for the user to acquire. The service offered by FWP resembles *Platform as a Service* (PaaS) type functionality for the OBS servers. OBS servers can deploy their build code and

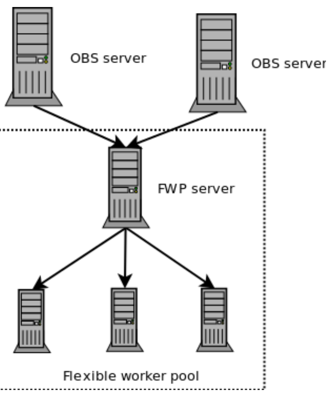


Figure 4.2: Two OBS configuration with FWP

worker code to be executed on the workers as long as the worker code provides the proper worker's API.

From the end-user's point of view, the OBS is already a software instance with some cloud-like behavior. The user typically cannot affect the exact source of the building resources and has to trust the OBS to provide them. An exception to this principle is the case of private OBS instances where the administrator hosts the workers and controls the build and worker code on the OBS server.

Similarly to users not being able to control the exact source of build resources, FWP provides the OBS instance additional resources based on the current build queues acquired from the OBS. As such, the OBS server does not require information regarding the amount of workers within the FWP service, but only has to acknowledge the need for more resources and the FWP will provide them if there are any available.

FWP concept aims to minimize the idle times and thus maximize the utilization rates of individual workers while reducing the hardware costs of hosting multiple OBS instances by making the dynamic pool of workers available for all the instances connected to it.

## 4.3 Software requirements

Several software design requirements were defined based on the desired functionality that was described in Section 4.2. The main requirements were reduced to dynamic worker allocation examined in Section 4.3.1, replaceable scheduler examined in Section 4.3.2, isolated environments examined in Section 4.3.3 and it being an external serviced as explained in Section 4.3.4.

### 4.3.1 Dynamic worker allocation

Dynamic worker allocation during runtime is a direct requirement derived from the original FWP concept described in Section 4.2. FWP should be able to dynamically allocate workers to specified OBS server instances for the required duration based on their build queues. Moving the worker allocation from one OBS instance to another is called a *handover*. Conceptually, the handover types were defined in the context of this research: *soft handovers* and *hard handovers*. Soft handover is a form of moving a worker from one OBS to another in a manner that does not require reconfiguration and restart of the actual worker process through shell commands. Hard handover is a forced handover maneuver that consists of manually modifying the configuration, cleaning up the environment and restarting the worker. The initial system specification did not determine the kind of handovers that were to be used in the final implementation.

The initial implementation concept was specified as the OBS instances being able to request workers dynamically when all the dedicated resources are being utilized. However, as a result of further research, the option for obtaining the build queues from the OBS server was discovered. By changing the requirement to being able to provide resources when required, a system monitoring multiple OBS instances could be designed. This allowed the design of better scheduling methods as the FWP is not relying on receiving requests at arbitrary times.

### 4.3.2 Replaceable scheduler

A scheduler performs the dynamic worker allocation described in Section 4.3.1. It should receive the information of dispatched builds and the existing build queues from the OBS instances in the system and allocate the workers based on that information.

An essential part of the design was to have a dynamic scheduler that can easily be replaced with another implementation if deemed necessary. In practice, this meant architecturally separating the scheduler from the main Flexible Worker Pool architecture and preferably running it as an individual process.

This also meant that FWP is not dependent on the existence of the running scheduler process and that the scheduler could be changed without restarting other FWP processes.

### 4.3.3 Isolated environments

Based on the need for security, especially in a corporate setting, the allocated workers and the OBS server should create isolated environments. This meant that other OBS instances should not be able to access the workers' environment or any data associated with it unless the worker in question is allocated to the OBS instance. This responsibility continues further as the environment should also be clean when the previous allocation ends and the worker is being allocated to another OBS instance.

Therefore, Flexible Worker Pool should create temporary virtual environments that grant access to the worker for a limited period of time and for one OBS at a time. One idea was that this could at least partly be achieved with the use of configurable software firewalls. The nature of the relationship between the workers and the OBS instance is temporary and therefore the firewall rules would have to be changed in real-time if this approach was to be used.

### 4.3.4 Service

An important requirement for the design was that Flexible Worker Pool could be offered as an external service for internal and external OBS instances. This meant that any OBS instance existing outside the main FWP instance could be connected to the worker pool in a dynamic manner in order to provide it additional resources when required.

In practice, this meant running Flexible Worker Pool as a separate instance rather than integrating it to the existing OBS infrastructure. This requirement gave the design some freedom, but also added some constraints. These constraints are examined in detail in Chapter 5.

# 5. SOFTWARE ARCHITECTURE

This chapter outlines the architecture of the implementation of the Flexible Worker Pool concept that was described in Chapter 4. It begins by introducing the design princibles in Section 5.1, followed by the description of the evolution of the communication model in Section 5.2. Section 5.3 describes the components of the final implementation and the following sections introduce each of these components separately in more detail.

## 5.1 Design principles

Two design principles were decided before the actual design phase of the project. These principles affected the resulting architecture as a whole and are listed below.

Tolerance to changes in the OBS protocol
: Modifications to the existing OBS protocol should be avoided if possible. In other words, changes in the internal OBS protocol should not require a multitude of changes to the Flexible Worker Pool implementation.

Minimal modifications to the existing OBS source code
: The amount of modifications to the existing source code should be avoided if possible. This way, no branching of the OBS would be required and OBS instances could be updated independently. Furthermore, it should make the Flexible Worker Pool easily addable to any existing system without separate software update requirements.

## 5.2 Communication model

The core of the Flexible Worker Pool system design is the communication model. Section 4 described the system as being an autonomous resource managing software that distributes workers to OBS instances based on their need for resources. Worker allocation can be done in two ways: by modifying the configuration and restarting the workers, and by redirecting the messages from the workers to OBS servers with a separate software instance handling this proxy functionality.

The handover process has to be secure to the extent that the new OBS instance should not be able to obtain any information regarding the previous build environment. This restriction is a result of the isolated environment requirement as it was described in Section 4.3.3.

Given that the OBS is a fairly large, limitedly documented system, the design on the communication model of the Flexible Worker Pool was started based on the OBS protocol independent ideas before knowing all the details of the actual system. This led to an evolution of the model as new issues emerged.

The following sub-sections describe the different phases of communication model development in chronological order and the final solution that was reached as a result. It is necessary to describe these parts and their advantages and problems in order to understand the final solution and why it was chosen over the alternatives.

### 5.2.1 Flexible worker pool as a proxy service

The first concept was to build the Flexible Worker Pool to serve as a proxy server between the workers and the OBS servers. The concept revolves around the idea that the communication between workers and OBS servers can be completely forwarded without modifying it in order to create virtual structurally different networks that are seemingly invisible to both the OBS and the workers.

**Advantages**. Proxying the data between workers and OBS' had the obvious merit of being completely independent of the OBS protocol. OBS operates on an HTTP protocol layer that is being used to send and receive XML objects. This allowed the use of a modified HTTP proxy for implementing the proxy communication model. Furthermore, with the proxy model, callback methods could be defined. These callback methods are called when the proxy server receives or sends a message allowing the modification of the redirection information as well as the message itself. This way relevant information could be captured or modified in the process which provided the design with more potential flexibility.

One great advantage of being able to control the flow and handle the packets in between is that the WPM can receive real-time status information of the state of the virtual OBS network as well as information about the status of individual workers. In terms of controlling the building (i.e. amount of builds for example), this gives the system the capability to limit the amount of builds OBS instances can build as the proxy server can choose to discard further messages after the worker has completed the build job quota given to it.

**Problems**. Some issues were discovered when it was noticed that proxying the data between the OBS instances and the workers works extremely fluently. In fact, the worker instance was completely unaware of the occurred changes in allocation. This meant that if the worker host was allocated to another OBS when it was building, the new OBS could obtain information about the build through the worker API. This sort of information leakage would be a serious security issue and it had to be addressed before the model could be securely implemented.

Furthermore, this model only allowed restarting the workers through the worker API. While this might not be a problem in a general case, it does not account for crashed workers nor can the worker's discard method be trusted to clean the environment including the cache when a handover from one OBS to another one is made.

### 5.2.2 Using remote execution scripts for worker control

In order to solve the problems that emerged in the design of the proxy service model, another approach was considered. The second idea was to use remotely executable scripts to modify the configuration of the workers and the OBS instances as well as restarting. This way, the OBS network infrastructure would actually be changed instead of using a virtual environment. This approach opened up some possibilities that the use of proxies alone could not achieve.

**Advantages** of this approach consisted of direct control over the workers, actual network infrastructure instead of one created by transparent proxies and directly configurable firewalls. In other words, allocated worker hosts could be reconfigured to serve the desired OBS instance and restarted with the firewall blocking data to the previous allocation.

**Problems**. It was apparent that the downside of this approach was the lack of real time status data from the workers. The worker status could only be monitored through polling methods or patching existing status update scripts to somehow inform the FWP. Neither of these options were elegant nor necessarily reliable. In practice, polling might result in a scenario where the worker would complete a build and start a new one without the FWP noticing. This would effectively make the scheduling very unreliable and non-deterministic. While patching the worker might in fact yields the desired result, it goes against the second design principle of not modifying the OBS source code.

### 5.2.3 The hybrid model

The third and final approach was to combine the use of proxying data with the remote execution scripts and use the strengths of both models in order to achieve the required functionality.

**Advantages**. The hybrid model uses data proxying to control the flow of data and receive real-time information of the status of the system. Remote execution scripts are used to gain direct control of the workers in order to modify their configuration and clean up the environment as well as their firewall's configuration and restart them when necessary.

**Problems**. The only discovered practical problem with this approach was its complexity. Either one of the earlier approaches alone can have their own problems when it comes to the implementation. Combining the use of both means that the practical issues of implementation have to be dealt for both. However, this issue was not insurmountable.

Further development revealed that the use of remote execution scripts would not necessarily be available for all setups. In practice, this led to researching alternative methods for performing the same actions earlier performed with the remote execution scripts. Even though the remote execution scripts are still used, their importance decreased when methods for restarting workers through their own API were found. Furthermore, methods for securing the environment for the duration of the builds were found. As a result, the use of remote execution script was made optional. The option to remove them completely was discarded due to worker restarting being more reliable using remote execution scripts.

## 5.3 Software components

Flexible worker pool consists of four independent components: the database, *Administrative User Interface* (AUI), *Worker Pool Master* (WPM) and the scheduler.

The database can be any underlying database supported by the Rails framework. In practice, it was tested with MySQL[19] and SQLite3[4]. It contains the data concerning FWP network components such as build services, worker hosts and workers and their relationships.

AUI is a web-based user interface implemented with Ruby on Rails application framework for system administrators. It is also used as the access point to the database for the WPM. It grants the administrator the tools for monitoring and controlling the state and structure of the virtual proxy network.

WPM is an independent process that handles the proxy threads based on the routing information it requests and receives from the database through AUI. WPM is responsible for proxying the data from the worker hosts to the correct OBS repository servers and vice versa. WPM uses *Worker Pool Infrastructure Module* (WPIM) for retrieving and storing data of the network structure. WPIM includes the functionality to perform requests to the AUI API and store the received runtime information.

The scheduler is a separate process that monitors the build queues of the repository servers connected to FWP and allocates the workers based on these queues. The manual allocation functionality is also accessible through the AUI. This means that running the scheduler is optional although often recommended for optimal performance.

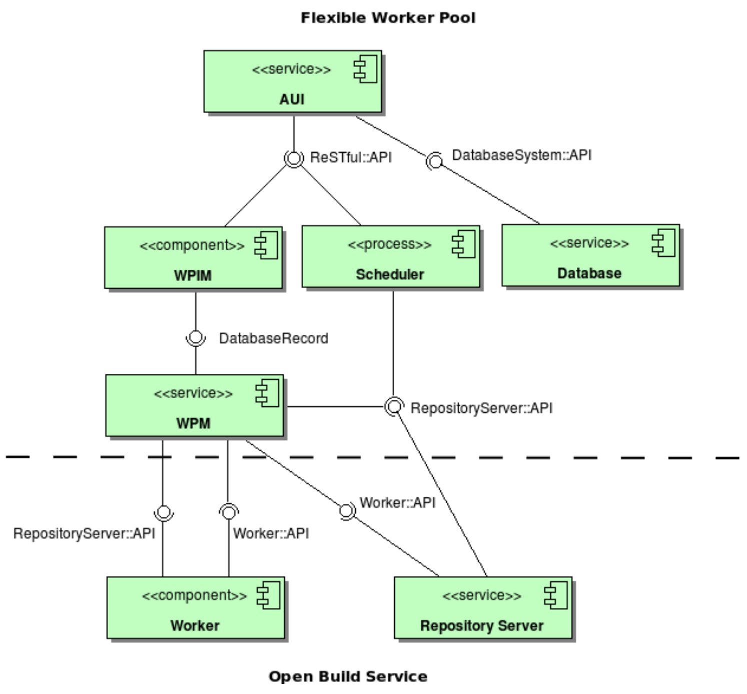


Figure 5.1: Component level structure of the Flexible Worker Pool

Figure 5.1 illustrated the relationships between software components. The relationships to OBS components (*Worker* and *Repository server*) can be seen. The individual components are described in detail in the following sub-sections.

## 5.4 Worker Pool Master

Worker pool master is responsible for conveying the messages from the individual worker hosts to the repository server and vice versa. This is achieved by running individual proxy threads for each worker host and repository server. The proxies use the data from AUI to forward the messages to correct addresses. WPM also updates the worker states and allocation values in the database when builds are commenced. Furthermore, it performs the handovers when remaining allocations reach zero and all build jobs have been completed. The automatic allocations are handled by the scheduler as described in Section 5.7. Optionally, the allocations can be set manually through the AUI regardless of the existence of the scheduler.

WPM consists of three major parts: Worker proxies, repository server proxy, and dummy repository server.

*Worker proxies* are started for each running worker within the FWP. Each worker proxy forwards the requests from the repository server directly to the appointed worker instance.

*Repository server proxy* is the main thread of the WPM. It forwards the requests from the worker hosts to the assigned OBS repository servers. It is also responsible for updating the worker's state information to the database and performing handovers when necessary. Repository server proxy also changes worker port from worker's state messages to correspond the appropriate worker proxy's port that generally differs from the actual worker's port.

*Dummy repository server* is a singular thread that provides some functionality of a repository server to the unassigned (or possibly some error state) workers. Since a worker does not start if it cannot receive a proper worker code, one will be provided for it by the dummy repository server.

Worker Pool Master serves as a data transfer medium in a rather transparent manner. In practice, this does not mean that WPM intentionally hides its existence but that the OBS communicates with it as if it was the repository server/worker. As a result of the design of OBS communication model, the actual traffic between the OBS and the workers goes both through the WPM and directly between the instances.

The workers send their state information and default requests to the repository server IP address or hostname they were given in the configuration file upon initialization. However, because individual build job communication works directly between the worker host and repository server as described in Section 3.4, the high bandwidth communication such as binary uploads and binary downloads bypass the proxies making the solution more efficient.

On a lower abstraction level, the classes include and consist of a multitude of structure classes. While the majority of the functionality of each class resides in the main classes, *RepoProxy* and *WorkerProxy*, it is important to understand the role of each sub-class in order to understand the system. The overall class structure and the details of the individual classes as illustrated in Figure 5.2.

### 5.4.1 RequestParser

*RequestParser* class parses the different kinds of HTTP requests sent by the workers and the repository server and returns instances of appropriate request classes, denoted by the *Request* suffix in the class name. It uses the query string associated with the request to identify the type of the message. If an unidentified request is passed to the *RequestParser*, it returns an instance of itself.

*StateRequest* class represents a state update message from the worker. The instance parses the values from the given URL. These values consist of the worker's id (in format: *worker host name/instance #*), architecture (f.ex. "i586"), listened port and state. The class also implements the method for modifying the port in the query port number translation.

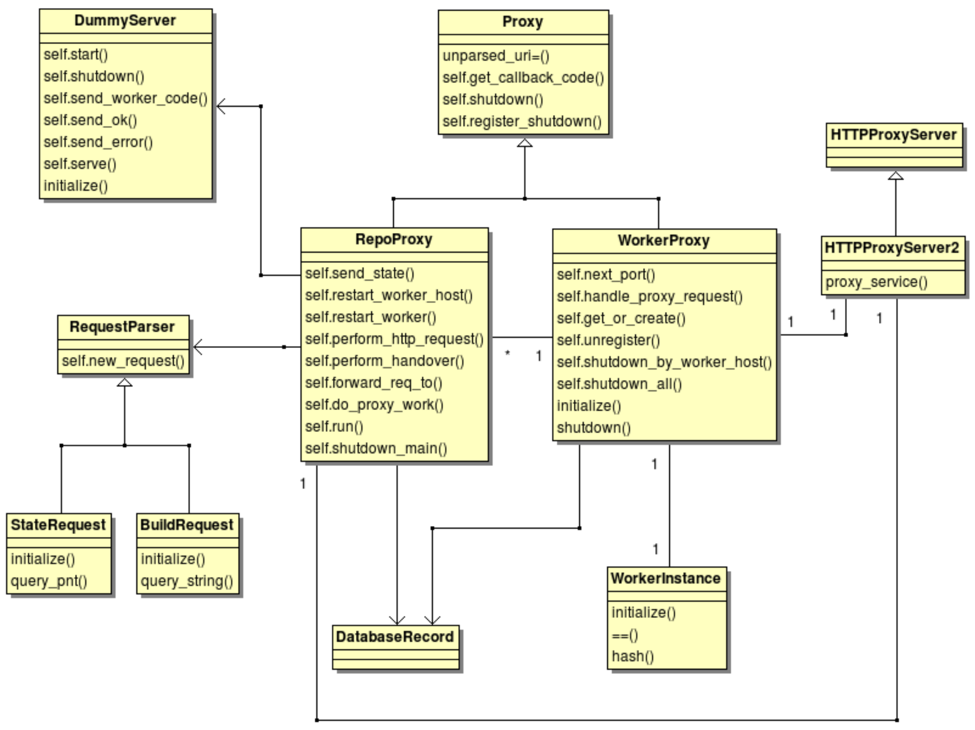


Figure 5.2: Worker Pool Master's class structure.

*BuildRequest* class represents a build job initiated by the repository server. It includes the necessary information about the build job, such as the target worker id and the worker and build code hashes. The actual build job data is stored as XML in the original request but is not parsed by the *BuildRequest* class.

### 5.4.2 RemoteExecutionScript

*RemoteExecutionScript* is an instantiable class. Each instance represents a shell script that can be executed on a remote machine through an SSH2 connection[35]. The chosen SSH2 implementation was a Ruby Gem called *Net::SSH*[2] that uses the system's underlying SSH2 implementation for forming connections. This is a good approach for automatic authentication because both the underlying SSH2 implementation and Net::SSH are widely used and thoroughly tested, making the choice secure as well.

### 5.4.3 DummyRepoServer

Class *DummyRepoServer* instantiates an individual thread dedicated to servicing non-dedicated worker hosts. When a worker instance is started, it requires the worker code to run. This will be a problem if, upon start up, the worker is not allocated to any OBS instance. In such cases, the worker code needs to be provided to the worker hosts manually. For such cases, the *DummyRepoServer* is used.

*DummyRepoServer* receives calls from the worker such as idle state messages and worker code requests and answers them properly. In practice, this means returning the worker code cpio encoded package when requested and answering the state updates accordingly. Since the WEBrick proxies are being used for the actual data transfer, this solution was simpler and far more convenient to implement than integrating these responses to the actual proxy server code.

### 5.4.4 WorkerProxy

Class *WorkerProxy* is an instantiable class. One worker proxy is instantiated for each worker. It is being used by the OBS server instance to communicate with the individual workers.

Each instance of the *WorkerProxy* listens to a single port. The port number is chosen by selecting the first free port with a running number greater than 3000. In other words, the worker proxies will receive ports (assuming non-reserved) 3001, 3002, ..., etc. in the order of initialization. The role of the worker proxy is to perform the necessary modifications to the messages that are send to the workers.

Besides potentially modifying the messages, the *WorkerProxy* class can also choose to discard messages that originate from illegal addresses such as unregistered OBS instances. Furthermore, it implements methods for shutting down the proxies.

Each instance of the *WorkerProxy* class instantiate a single WEBrick proxy server[29] thread for the actual proxy functionality. WEBrick proxies operate on callback methods. The main functionality is a *Proc* object stored in the instance variable *@callback_proc* that is passed to the proxy thread as a variable upon initialization. It will be called when the proxy receives a message. In theory, WEBrick proxy server offers functionality for defining callbacks for return messages as well, but these callbacks are not utilized in the WPM implementation. Instead, all the responses are returned to the original sender as they are received from the repository server.

Class *WorkerProxy* implements a *class method* interface that is used by the callback code. Class methods are similar to static methods (in comparison to languages such as C++) in the sense that they can be called without instantiating the class through the class interface. This interface was necessary for using the WEBrick proxy as the thread is not running inside the *WorkerProxy* class. In practice, the callback sets the forward IP address and port for the message or, in case of an error, discards it by throwing an exception.

Instance of the *WorkerProxy* class is identified by an unique instance of *Work-*

*erInstance* class. It contains the necessary data to make the distinction between workers (IP address, name and port).

### 5.4.5 RepoProxy

*RepoProxy* class maintains a single proxy thread for forwarding the messages from workers to repository servers. The main proxy functionality is handled by a WEBrick proxy server instance, similar to *WorkerProxy* class. The internal implementation of the *RepoProxy* class relies completely on class methods and, thus, it is never actually instantiated. It listens to the same port as OBS repository servers. In fact, this currently sets the limitation of not being able to run WPM on the same host as the repository server. It also limits the WPM to the extent that if the OBS instances have been configured to use unconventional ports, it will only be able to service one of these ports.

Technically it might be possible to run an instance of WPM for each separate port of the connected OBS instances, but the concurrency issues resulting from this have not been investigated. This design decision was reached due to the internal repository server implementation issue where the repository server sends the port it listens to the worker. Changing this from the message itself (not the headers), would have been possible, but this would have resulted in all the data transfers going through the repository proxy, including the heavy binary data transfers. The decision was made to prefer efficiency over adaptivity on this issue.

Upon receiving a message from a worker host, the repository proxy thread sends a query to the AUI in order to fetch the associated worker host information. An exception will be thrown as a result of non-existing worker hosts. The exception will cause the message to be discarded and the information about the sender to be logged. Errors in the proxy process will typically only discard the message and future messages will work properly if they are coming from validated sources.

Most of the messages proxied by the repository proxy are worker state messages. Other than that, the worker does not need to be in contact with the OBS server instance, unless it is commencing a build. State messages are the most important for WPM since they deliver the worker state and actions to it in real-time. Since the listened port of the *WorkerProxy* instance differs from the actual worker's port and the port is included in the state message query string, the repository proxy performs a port number translation before forwarding the message, changing the port from the actual port of the worker instance to the port of the corresponding worker proxy.

Another important function of the *RepoProxy* class is to forward the */getworker-code* requests. Depending on the existence of the allocation for the worker host the request originates from, the message will be forwarded to either the allocated OBS server instance or the *DummyRepoServer* instance.

After completing a build, the worker will send it directly to the repository server so no *RepoProxy* is required in the transaction.

### 5.4.6 Communication between classes

The structure of WPM consists of the main repository proxy thread that initializes the separate worker proxy threads. The repository proxy is a non-instantiable class that functions much like a singleton[10].

*WorkerProxy* maintains list(s) of the *WorkerProxy* instances. These lists are not directly accessible outside the *WorkerProxy* class, however, there are methods for getting and creating *WorkerProxy* instances.

*RepoProxy* communicates with the individual *WorkerProxy* instances through *WorkerProxy* class method interface. It fetches the *WorkerProxy* instance handles identified by the *WorkerInstance* class instances it can generate from the necessary worker host and worker information. In practice, this is required when the repository proxy receives a state message from a worker that is, or will as a result of a handover, be assigned to a repository server. *RepoProxy* will create the *WorkerInstance* based on the data received in the state request and start the appropriate *WorkerProxy* thread unless it's already running.

## 5.5 Worker pool infrastructure module

The worker pool infrastructure module represents the system's internal data structure. The structure is similar to the database structure and, thus, similar to the AUI's model structure. Such design pattern is called *Active Record*[6].

The first consideration was to use the AUI models directly, and thus the use Ruby on Rails' *ActiveRecord*[12] to access the database directly was considered. However, it was decided that having a single access point to the database is better design structurally. It was also concluded that this design should solve some potential concurrency issues with the database usage.

Instead of using Rails' *ActiveRecord*, a new class was implemented with a similar interface. *DatabaseRecord* class is functionally and syntactically similar to *ActiveRecord*. However, instead of directly accessing the database, *DatabaseRecord* uses an external RESTful API[5] interface provided by the AUI.

## 5.6 Administrative user interface

*Administrative User Interface* (AUI) was implemented using the Rails 2.3 framework for Ruby (usually referred to as *Ruby on Rails*). This framework was chosen because it is also used in the OBS web-based user interface implementation.

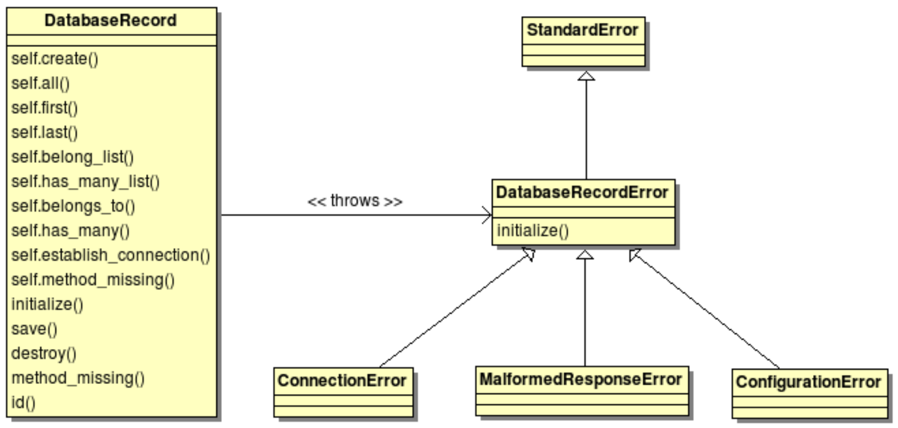


Figure 5.3: WPIM class structure.

The main function of the AUI is to provide the user and the WPM the means to modify the routing information from worker hosts to OBS server instances. AUI implements a RESTful HTTP interface for WPM and the scheduler. This interface communicates with sent and received JSON[3] objects. Furthermore, it offers a web-based interface for the user for making direct modifications in the database.

The class structure of the AUI follows the Model-View-Controller (MVC)[25] structure. Each resource in the database is represented by a model class and controlled by its own controller as illustrated in Figure 5.4. Individual views provide the user with the means to view and modify the data in the model classes through the controller. Such structure is typical for Rails applications.

As illustrated in Figure 5.4, three identifiable resources can be seen in the class structure: *BuildService*, *WorkerHost* and *Worker*. Each is controlled solely by its dedicated controller. Besides the method *index* that lists all the resources of that type, the other methods are always targeted to an individual database resource.

Classes *ActionController::Base*, *ApplicationController* and *ActiveRecord::Base* are the base classes for the structure provided by Rails. They provide the actual database connectivity as well as the means to deliver relevant data to the views for the client application to receive.

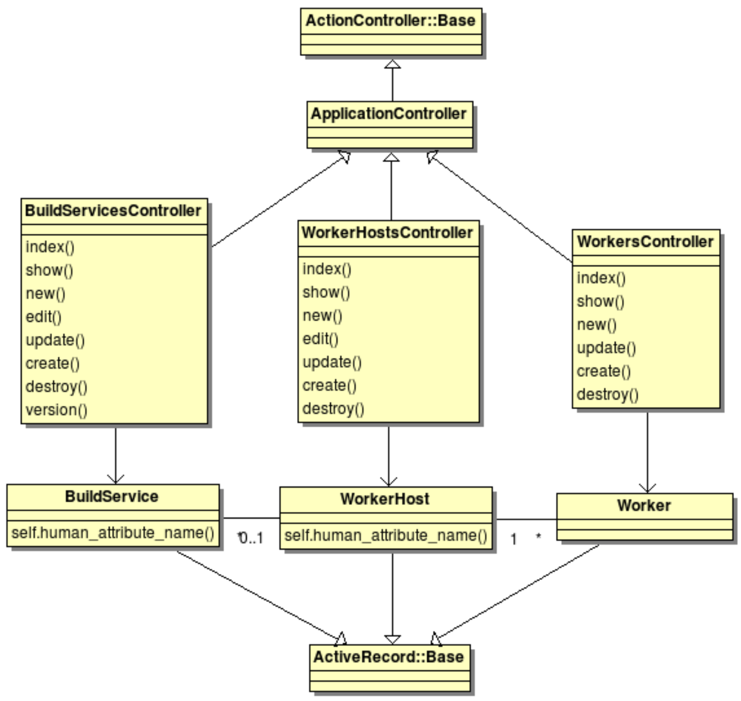


Figure 5.4: Administrative user interface class structure.

Every action of a controller can have a view. In Rails applications, views are displayed by the framework and no new classes need to be implemented for them. Instead, views are implemented by providing Rails with properly formatted HTML templates with embedded Ruby (ERB)[17], or by using Rails' plug-ins for pre-formatted objects such as JSON.

### 5.6.1 Controllers

The flow control and the actual program logic of the software are located in the controllers. The controllers decide, generally based on the users actions, which action to perform and which view to show as a result as well as what information to include in it. Controllers access the resources through the model classes and convey the information to the views where it is displayed to the user or the client application that made the request.

*BuildServicesController* handles the creation, modification, deletion and fetching of information regarding the build service records in the database. *WorkerHostsController* handles the creation, modification and fetching of information regarding the worker host records in the database. Each worker host resource represents a single physical or virtual host with a unique IP address. *WorkersController* handles the creation, modification and fetching of information regarding the worker records in the database.

| Resource | Views |
|---|---|
| *BuildService* | Edit |
| | Index |
| | New |
| | Show |
| *WorkerHost* | Edit |
| | Index |
| | New |
| | Show |
| *Worker* | Index |

Table 5.1: Worker states and their representative strings

### 5.6.2 Models

All model classes are inherited from the Rails *ActiveRecord::Base* class. *ActiveRecord::Base* provides the classes with the interfaces to access the database. Each model class represents its equivalent table in the database. This means that the instance variables of a model class correspond to the fields of the table in the database. In practice, this means that Rails programs are written on a rather high level of abstraction and the actual database interface is not visible to the programmer.

*BuildService* instance represents an individual build service in the database. It stores build service variables such as *name*, *ip* and *port*, as well as WPM related data such as *scheduling priority*, *maximum dynamic worker hosts* and *online status.*

*WorkerHost* instance represents an individual worker host in the database. It stores information about the worker host as well as required routing data. This information includes the name and IP of the worker host as well as the current and next allocation lengths and targets. Notice that it contains no information of the architecture or ports listened by the worker host.

*Worker* instance represents an individual worker instance running within a worker host. It stores the information updated by the WPM upon state messages. This information consists of the name, port, state and the architecture of the worker instance. Furthermore, it stores the foreign key to the worker host, forming a many-to-one relationship between *Worker* and *WorkerHost* classes.

### 5.6.3 Views

The user interface consists of views. Each view is represented by a HTML view with embedded Ruby. For each action that user can perform there exists a view. Such actions include creating, viewing, deleting and modifying resources.

Table 5.1 lists the views by resource type. Notice that not all actions require a specific view. For example deleting an object from the database will redirect the user to the *Index* view of that resource. Saving an edited object in *Edit* view will trigger *Update* action that redirects to the *Show* action. Notice that *Worker* only has *Index* view. This is due to the relevant worker information usually being displayed in the *WorkerHost* views. The index was only implemented so that removed worker instances could manually be removed from the database.

Each view is identified by an URL that points directly to the resource and action (f.ex. */worker_host/1/edit*). This kind of referencing system provides a simple, easily accessible interface from the browser as well as the AUI API interface.

AUI provides WPM with a RESTful API to its methods. The methods are, for the most part, same as the ones used with AUI user interface with a couple of exceptions.

Firstly, the worker information can be accessed directly through the *WorkersController* unlike in the user interface where individual workers can only be removed through the *WorkerHost* view.

Secondly, the amount of data retrieved through the API is usually smaller than that displayed by the user interface. All API calls (except listing the resources) are directed at individual resources in the database.

```
{"build_service":{
   "worker_hosts":[],
   "name":"Asmodeus",
   "created_at":"2011-06-07T16:42:00Z",
   "updated_at":"2011-06-21T14:07:24Z",
   "port":5252,
   "priority":2,
   "lock_version":1,
   "max_worker_hosts":-1,
   "id":1,
   "ip":"192.168.1.100",
   "status":true
}}
```

Program 5.2: An example of a *BuildService* JSON object from AUI API.

The objects are transferred over HTTP as JSON objects. Program 5.2 illustrates a sample *BuildService* object that was received through the AUI API. AUI makes the distinction between browsers and WPM based on the action URL. In order to receive JSON responses from the AUI, ".json" has to be added to the end of the action URL. For example */build_service/1.json* would return the JSON representation of the first build service in the database.

Internal Rails implementation dictates some of the request types for the standard RESTful interface. These types for each action are listed in Table 5.2.

| Action | HTTP Request Type | In JSON API |
|---|---|---|
| *Index* | GET | Yes |
| *Show* | GET | Yes |
| *New* | GET | No |
| *Create* | PUT | Yes |
| *Edit* | GET | No |
| *Update* | POST | Yes |
| *Destroy* | DELETE | Yes |

Table 5.2: HTTP request types for API calls (Rails 2.3)

## 5.7 Scheduler

Scheduler takes care of properly figuring out the allocations for each worker host. It does not commit the actual handover or directly assign the worker hosts, but merely sets the next allocation through the AUI API. WPM will decide whether to act on the next allocation or not. The actual scheduler that performs the actions is chosen in a configuration file from a list of dynamically loaded scheduler classes.

```ruby
# Store the current priorities between scheduling rounds
@cur_priorities = Array.new

def perform_scheduling
  OBS = fetch_build_services          # Fetch OBS' from database
  OBS.queues = fetch_queues           # Fetch build queues from OBS

  # Loop as long as free workers and queues exist
  while workers.free? and queues.exist?
    # Update current priorities (internal for scheduler)
    @cur_priorities.each do |index, prio|
      prio += OBS[index].priority
    end

    # Sort the queues, highest current priority first
    obs_by_priority = sort(OBS, :by => cur_priorities)

    # Service the OBS by giving it workers
    service_obs(obs_by_priority.first)

    # Reset the priority and reduce queue
    obs_by_priority.first.reset_priority
    obs_by_priority.first.queue--
  end
end
```

Program 5.3: Ruby pseudo-code for weighted round robin algorithm.

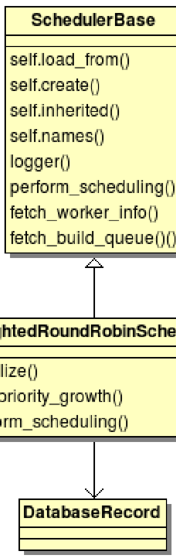


Figure 5.5: Scheduler class structure.

A simple weighted round-robin[27] algorithm shown in Program 5.3. The goal was to implement a simple and fair scheduler with the option to prioritize some OBS instances over the others. This was done by setting each OBS instance in the database with a base priority that determines the level of service it will receive from the FWP. Each scheduling round, the base priorities are added to the current priorities maintained by the scheduler between scheduling rounds. The build service queues are then sorted by their current priority and a worker is allocated. The OBS instance that receives the worker will have it's priority reset to the initial value. This loop will be continued until no free workers or no queues are left.

The algorithm is fair and thus guarantees that each OBS will get serviced at some point and that equally prioritized OBS instances will acquire the same amount of workers assuming they have similar build resource requirements.

The practical implementation of the algorithm is slightly more complex since it processes the build job queues by architectures and takes capabilities of the workers into account. This means that the scheduler has the required information to determine whether a worker is capable of completing the queued build.

The scheduler was not the main emphasis of the research and could definitely be more efficient with more advanced scheduling algorithms. This is also one of the reasons for designing the scheduler replaceable so that more advanced schedulers can be designed and easily taken into use in the future.

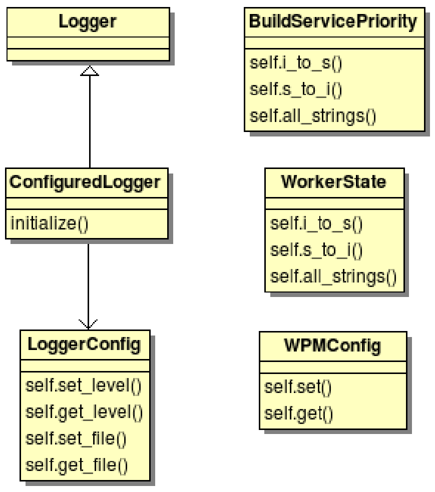


Figure 5.6: Common classes shared by WPM, AUI and the schedulers.

All the individual scheduler implementations are inherited from the base class *SchedulerBase* that implements a basic set of helper methods for the schedulers to use as illustrated in Figure 5.5. These methods include OBS and worker state check methods as well as the methods required for performing the actual worker allocations.

The scheduler classes themselves are only required to implement the method *perform_scheduling* which will return either *true* on success or *false* on failure. It was decided that the scheduler should not modify the current allocations of any of the worker hosts because this might cause some routing errors and security risks. The *perform_scheduling* method is only allowed to modify the *next_allocation* and *next_build_service* variables of a worker host. While this requirement is not monitored by the software, all scheduler designers are urged to comply with the set restraint as breaking it might result in an unstable system.

## 5.8 Common classes

In addition, WPM, WPIM, AUI and the scheduler use a shared set of helper classes. Such classes include configuration loaders and symbol conversions to readable text. These classes are illustrated in Figure 5.6.

Class *Logger* is a Ruby base-class that implements a logger with the support for different logging levels and output formats. It is being used by every part of the WPM to record actions and errors within the system.

Class *LoggerConfig* is a configuration class that hosts a set of configuration variables for separate loggers. Each part of the WPM and the scheduler has it's own logger and a matching variable set to *LoggerConfig*. In the implemented version, the

| | |
|---|---|
| IDLE | "idle" |
| BUILDING | "building" |
| KILLED | "killed" |
| DISCARDED | "discarded" |
| REBOOTING | "rebooting" |
| BROKEN | "broken" |

Table 5.3: Worker states and their representative strings

logging level and output file could be set for each logger. The output file could also be set to point to a stream such as *STDOUT* or *STDERR*.

Class *ConfiguredLogger* extends the basic logger by adding the functionality for manually configuring the loggers logging level and output type in the WPM configuration. A configured logger will receive it's logging level and output type from the *LoggerConfig* class.

Class *WorkerState* is a class shared by the AUI and the WPM. It is a simple mapper class that maps individual worker status strings to their designated integer representatives. Since both of the forementioned instances use the worker state information as it resides in the database, such shared class was required. It offers a simple interface for converting string values into integer constants and vice versa. The possible conversion values are illustrated in Table 5.3.

Class *BuildServicePriority* class is similar to *WorkerState* class. It maintains constants regarding different build service priorities. Build service priorities are used for scheduling. Each build service has a priority that represents the level of service it's expecting to receive. Since the priorities are set through the AUI, *BuildServicePriority* class is being used by both the scheduler and the AUI and is thus placed under the common classes.

Class *WPMConfig* holds the configuration variables for WPM and the scheduler. Such variables include the AUI address and the scheduler's implementation that is to be used. The variables are loaded as a part of loading *fwp_settings.rb* file upon WPM startup.

## 5.9 Implementation tools

The tools for implementing Flexible Worker Pool were chosen based on the OBS' implementation techniques. Since the OBS front-end was implemented using Ruby on Rails (2.3), it was decided that AUI would be implemented using the same programming language and framework.

WPM shares some data regarding the WPIM with the AUI as described in Section 5.8. This, along with the implementers preferences, lead into the decision to use Ruby (1.8.7) as the main programming language for WPM as well.

The underlying database system can be freely chosen from those supported by the Rails framework[11]. SQLite3 was set up as a default and was used in the development process and is therefore the most tested database system for the flexible worker pool. MySQL was also tested in a real production environment. Either way, the underlying database should not have effect on the functionality and it can be chosen freely from the pool of Rails' supported systems.

# 6. EFFICIENCY ANALYSIS

This chapter describes the efficiency analysis performed for the Flexible Worker Pool in practice and using simulations for the scheduling algorithm. Since the solution does not improve the efficiency of a worker in a single build, the analysis will concentrate on larger FWP networks with multiple OBS servers and workers and examine the performance compared to separate OBS instances with a static set of workers. Most of the analysis is based on simulations due to the limited amount of actual usage statistics.

## 6.1 Performance in a typical hardware configuration

Typical hardware configuration for an OBS network consists of an OBS server and workers running on separate hosts. The main focus of this analysis is on configurations with multiple OBS instances as those are affected by the FWP.

The simplest case to see benefit from the FWP is the case of two OBS machines with one worker. With the current OBS architecture, this worker cannot be shared between the OBS instances and must be configured to serve one OBS only. In this case one OBS instance gets 100 percent service rate and the other one gets none.

If a Flexible Worker Pool instance is added to the configuration, assuming an infinite build queue for both OBS instances, the setup guarantees 50 percent service rate at all times. The scheduler maintains a balance between accepted build jobs so neither is being favored (unless the scheduler priorities are set). Notice that the time it takes to complete a build can vary and the current implementation of the scheduler does not take this into account. In practice, this could be made more efficient by improving the scheduling algorithms so that they would estimate the amount of time spent for the build. However, this is not in the scope of this thesis.

Notice that in this case a machine for the FWP is added since with the current implementation of the FWP, it cannot reside on the same host as the repository server. While the amount of workers remains the same, a fourth host is required to properly utilize the Flexible Worker Pool. In such setting, if the FWP host is capable of efficiently functioning as a worker, the more efficient solution would be to utilize it as a worker. Both OBS instances would then be receiving service from a 100 percent dedicated worker, vastly surpassing the performance of the worker pool. However, if we assume the worker being of different architecture for faster native builds, the FWP host could not be used as a worker, thus rendering the FWP the only viable solution in order to utilize the worker for both OBS instances.

If a third OBS host is added to the configuration, there is no way to implement a working network without the Flexible Worker Pool. In such a case, only 33 percent service rates can be guaranteed. While the number is significantly lower (per OBS) compared to individual dedicated worker hosts, the fact that most worker hosts are likely to be idle most of the time makes it a viable solution considering the saving in hardware costs. For example, assuming that each of the three OBS will have a build waiting 25 percent of the time, having 33 percent service rate guarantees that all the builds will finish. However, since the builds can be queued at the same time, some OBS instances might be forced to wait for the completion of the other OBS' build. Thus, the best way to guarantee fast service delivery is to host dedicated worker hosts for priority builds, but this adds some hardware costs.

The actual advantages of the FWP can be seen more clearly when the configuration includes more host machines and non-infinite build queues. Several typical scenarios will be examined in Section 6.3. Since one of the key goals of FWP is to complete same amount (or more) of builds with less hardware, the focus was on examining the two solutions on different hardware settings and comparing their potential capacity.

## 6.2 Usage statistics

The system was initially tested with virtual machines. However, due to the limited capacity, only a handful of virtual machines could be used be used for testing and proper statistics of the usage levels could only be achieved through testing in real environment. FWP was taken into use for building MeeGo for ARM architecture.

- 1 Flexible Worker Pool
- 2 Build Services (OBS-1, OBS-2)
- 15 ARM Worker Hosts in FWP (Worker-1, Worker-2, ..., Worker-15)
- 3 Static ARM worker hosts (2 for OBS-1, 1 for OBS-2)
- 8 Static x86_64 Worker Hosts outside FWP (allocated to OBS-1)

The hardware configuration included 2 OBS instances with 18 ARM workers. Namely *OBS-1*, *OBS-2* and workers *Worker-1* through *Worker-15* in the FWP. In addition, 3 static ARM worker hosts are allocated outside the FWP, two for OBS-1 and one for OBS-2. Furthermore, the OBS-1 that was assumed to have more traffic had 8 additional static workers running on *x86_64* architecture for cross-compiling

| **Worker host** | **Builds** | **Utilization** |
|---|---|---|
| Worker-1/1 | 48 | 3.00% |
| Worker-2/1 | 38 | 5.49% |
| Worker-3/1 | 24 | 8.26% |
| Worker-4/1 | 24 | 5.96% |
| Worker-5/1 | 20 | 4.81% |
| Worker-6/1 | 14 | 10.18% |
| Worker-7/1 | 7 | 3.34% |
| Worker-8/1 | 2 | 0.03% |
| Worker-9/1 | 9 | 6.07% |
| Worker-10/1 | 1 | 0.02% |
| Worker-11/1 | 0 | 0.0% |
| Worker-12/1 | 6 | 5.31% |
| Worker-13/1 | 1 | 0.06% |
| Worker-14/1 | 0 | 0.0% |
| Worker-15/1 | 14 | 1.87% |

Table 6.1: Build statistics from a real environment

| **OBS instance** | **Completed builds** |
|---|---|
| OBS-1 | 194 |
| OBS-2 | 14 |

Table 6.2: Completed builds per OBS

ARM packages. This totaled 15 dynamically allocatable workers plus static 12 workers for OBS-1 and 1 static worker for OBS-2.

Table 6.1 includes the usage statistics from the real environment from a 24 day period of actual use of the system. During this time, numerous MeeGo utility and system builds were completed, mostly by individual developers compiling their software. As can be seen, the usage levels were fairly low on the average. The utilization percentage represents the time worker was building instead of being idle. It was noticed that the early worker instances were favored over the latter ones in terms of accomplished builds. This was due to the lack of implementation of a load balancing mechanism in the scheduler. It should also be noticed that the amount of builds does not correlate directly with the utilization rate as the different builds can take an arbitrary amount of time. This is also why the amount of builds is not completely linear from first to last.

Table 6.2 shows the amount of completed build jobs per OBS. These statistics only include the build jobs that went through the OBS. In practice, it is highly likely that the static workers were utilized as a priority and are therefore not shown in these statistics. In that sense, the statistics also show that the burst in build jobs has required more resources than statically available for the OBS, effectively making the process faster as more parallel build jobs could be completed. The amount of packages built by OBS-2 was significantly lower in total, but the duration of the build jobs was also decreased significantly due to being able to build the packages in a parallel fashion. In practice, most of the build time was allocated to OBS-1. This meant that instead of utilizing a static set of hardware resources, OBS-1 got all the resources in the pool for the duration of its build jobs, which made the process remarkably faster compared to having the worker hosts divided staticly. Worker-11 and Worker-14 had crashed during the period and were not restarted during the highest build spikes and the builds completed by them is therefore zero.

## 6.3 Simulations

In order to analyze the efficiency of the scheduler solution, several simulations were written with MATLAB®[32]. The simulation code is available in Appendices A, B, C and D. The use of simulations was needed due to the limited amount of real data available at the time of this research. These simulations simulate the weighted round-robin scheduling algorithm that was described in Section 5.7 in various different setups.

Two assumptions were made for the following simulations: all the build times are constant and similar, build frequencies (the frequency with which the build jobs are commenced in the system) can vary per OBS. The completed builds were analyzed over a 100 time unit time frame over a 1000 rounds of simulations in order to find the averages.

In scenarios 1 and 2, the hardware configuration is assumed to be the following:

- 1 Flexible Worker Pool
- 2 Build Services
- 4 Worker Hosts

In the dynamic case, both workers are connected to the network through the Flexible Worker Pool. In the static case, the workers are divided evenly for both OBS instances.

### 6.3.1 Scenario 1: Even build frequency

In this scenario each build was assumed to take 5 units of time. This means that on the average with 20 percent build frequency there is always a build waiting in the queue.

Table 6.3 illustrates the simulation results over 1000 simulation rounds. It is noticed that with 50 percent build queue frequency, the results are the similar for both configurations.

| | **OBS-1** | **OBS-2** |
|---|---|---|
| **Dynamic workers** | 39.75 | 39.76 |
| **Static workers** | 39.35 | 39.37 |

Table 6.3: Completed build jobs with even build frequency ratio

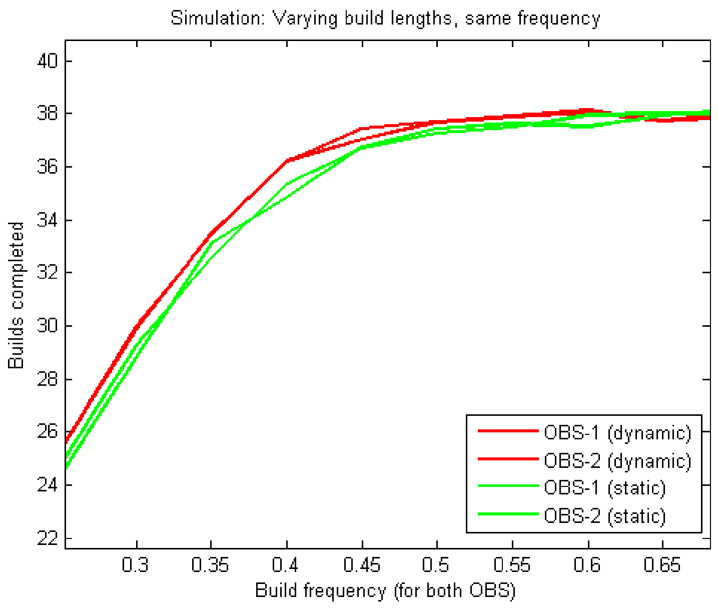


Figure 6.1: Simulation results for OBS' build frequency as equal variables

In practice, it is possible that some variance in build times will occur due to arbitrary waiting periods. It should be noted that even in this case, dynamic worker sharing always appeared to emerge slightly ahead. While the difference is quite negligible in practice, it shows that the dynamic approach can benefit the system in situations where other build service's queue is empty. Such cases are rare in the given setting.

### 6.3.2 Scenario 2: Equal increasing build frequencies

In order to determine the performance in the cases where the build frequency is the same, this scenario evaluates the performance of the static and dynamic solutions by setting both build frequencies equal in the range from 0 to 1.

Figure 6.1 illustrates the results of the simulation. The advantage of the Flexible Worker Pool can clearly be seen, especially in the middle section of the range. While the advantage is relatively insignificant, it still surpasses that of the static case. This advantage is a result of the FWP solution performing more flexible in the cases where one of the build services is idle and the other one has more than 2 jobs queued up. In such case, FWP will allocate additional resources to the OBS that requires them

| | **OBS-1** | **OBS-2** |
|---|---|---|
| **Dynamic workers** | 55.27 | 24.36 |
| **Static workers** | 39.99 | 24.77 |

Table 6.4: Completed build jobs with 3-to-1 build frequency ratio

so the utilization rates are higher and more balanced.

For lower frequencies, the results were almost identical. This was due to both OBS' having enough resources at their disposal. Once the frequencies reached higher rates, both OBS' had jobs queued up at all times. Fair allocation is effectively similar to both OBS having 2 dedicated workers in such a case and no additional resources can be gained at this point. Notice that FWP still performs at equal level compared to the static case.

Even though variance in build queues showed some advantage when FWP was used, the main advantage was seen in uneven build queues and bursts of builds rather than constant build frequencies.

### 6.3.3 Scenario 3: 3-to-1 build frequency difference

In this scenario the 3-to-1 build frequency ratio was assumed between the OBS instances. In practice, OBS-1 was pushing new build jobs 3 times more frequently than OBS-2.

As illustrated in Table 6.4, a vast improvement can be seen in the amount of completed builds for OBS-1. In fact, the noteworthy statistic is that the number is significantly larger than with two single dedicated worker hosts. This is due to the fact that OBS-1 occasionally has more than two worker hosts at it's disposal. OBS-2 commences builds at a lower frequency leaving the worker hosts free for OBS-1 to use.

The lower build frequency for OBS-2 also explains why the amount of completed is significantly lower for OBS-2 compared to OBS-1. Since the frequencies affect the amount of builds the OBS' are trying to build, the numbers are not directly comparable between scenarios.

In this case, a minor difference can be noticed between the static and dynamic worker sharing method for OBS-2 with the lower build frequency. The dynamic version appears to consistently give a lower amount of completed builds. While this is practically negligible, it can still be explained by the fact that in the static setting, the OBS always has two worker hosts in it's disposal, while in the dynamic case, there are times when OBS-2 has to wait for the worker hosts to be released by OBS-1.

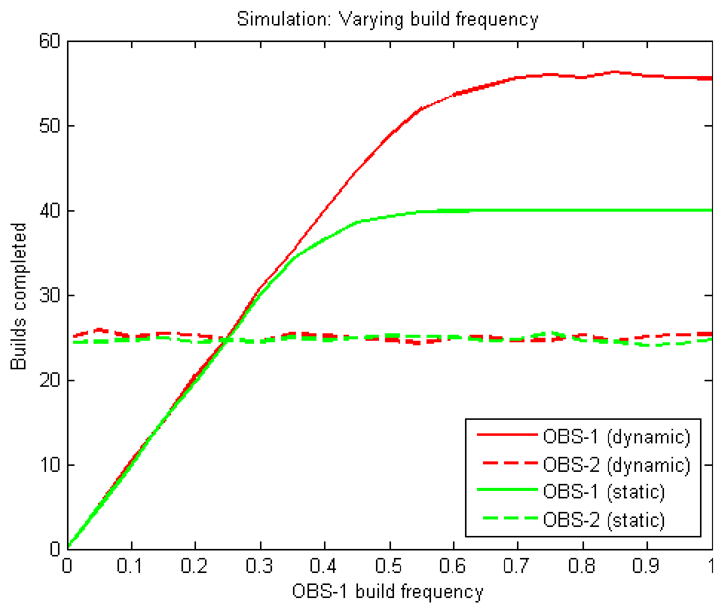


Figure 6.2: Simulation results for OBS-1's build frequency as a variable

### 6.3.4 Scenario 4: Increasing non-equal build frequency

The third scenario examined the case where the build frequency was set as a variable in order to draw a graph to examine the benefit gained from using the Flexible Worker Pool. The build frequency for OBS-1 was set to 0.25 while the frequency for OBS-2 was set to run from 0 to 1.

Figure 6.2 illustrates the results. The light gray lines show the amount of completed builds for the static dedicated workers. The two lines reaching higher build counts illustrate the amount of completed builds for the FWP solution. The figure shows that the amount of completed builds for OBS-2 in both cases was close to the same. With such low build frequency and a dedicated worker, it is reasonable to assume that OBS-2's worker requirements were close to fulfilled throughout the simulation. Such was the case with the FWP solution as well.

However, some key points were noticed from the completed builds of OBS-1. The amount of completed builds was close to equal until a 0.3 build frequency was reached. Until this point, both OBS instances in both configurations had enough resources to build the jobs in their queues. By the 0.5 build frequency limit, the static case had reached its full capacity.

For the Flexible Worker Pool, however, the results for OBS-1 rapidly surpassed those of the static case from 0.3 build frequency and forward. This is because FWP allowed OBS-1 to utilize OBS-2's excess building capacity for its own build jobs. Notice that using the excess capacity did not visibly cut resources away from OBS-2. OBS-1 reached it's maximum building capacity around 0.7 build frequency. At this point, the amount of completed jobs surpassed that of the static case by 40% (16 builds) per simulation.

# 7. CONCLUSIONS

The goal of this research was to design and implement a build resource sharing system for OpenSUSE's Open Build Service in order to reduce hardware costs and achieve higher utilization rates in individual OBS instances. A working solution was discovered that utilized the idle resources when needed and performed similar to the static case when no idle resources were available. In practice, FWP could achieve higher utilization rates of build resources and thus, complete more builds with less hardware compared to the existing static system.

The efficiency analysis of the solution showed the advantages of the system compared to the static dedicated workers. Due to the low utilization rates in real environment, the system's effectivity could only be measured properly in the general case of only one OBS requiring additional resources at once. Such cases proved that the solution is more efficient as the OBS could utilize all the resources in the worker pool and complete the build jobs faster as a result. In a static case, the OBS would have been limited to the workers initially dedicated to it.

In order to measure the performance in other types of situations, scheduler simulations were created. The simulations displayed that advantage exists even in evenly distributed build queues, even though it was smaller in such cases. The main advantage of the system can be seen when the build queues are uneven and unbalanced between the OBS instances. Such scenarios occur when one or more OBS instances are not utilizing their build resources to the fullest extent. The static build host network left those resources idle, but the FWP solution utilized them when there were any available.

In real life scenarios, the builds often stack up in queues in bursts, effectively meaning that the need for resources is sudden and the distribution shows spikes. In such cases, it is important to be able to utilize maximum computational capacity to get these job chunks built as soon as possible. As displayed by the test results, these are the kind of scenarios where FWP performs the best.

The practical efficiency of the scheduler, and thus FWP, could likely be improved by introducing advanced scheduling mechanisms, that would take additional variables into account. While it is very difficult to estimate the build time of a single package, caching the build times of frequently built packages could be one way to help with the estimation.

It was also discussed that FWP is a cloud-like system providing a PaaS type service for the OBS. In theory, adding any work stations or hosts as temporary workers to the FWP could be considered. Some additional development for that would be needed in order to make them function when they are idle, however, it was considered to be a viable option for future development.

Compared to the research by Ville Seppänen that utilizes the AWS, the related relevant challenges were for the most part different. The use of AWS posed challenges related to the time it required to upload the worker infrastructure to the cloud. Thus, the essential difference in the solutions was that the cloud solution surpasses the potential scalability of a single FWP instance in most cases at the cost of speed. Another difference is the cost. FWP still offers a limited set of hardware resources that are purchased as a one time investment by the service provider. The cloud bursting solution relies on an external service that is paid for based on the usage.

In his research, Seppänen concluded that the cloud solution is a cost-effective solution for the problem. It is, similar to FWP, the most efficient in the cases with a large number of packages compiled in a short period of time as opposed to sustained workloads. His research also stated that the sustained workloads are probably cheaper to handle in-house. One of the key differences is that FWP is often used in an in-house setting. While the scalability is still limited by the amount of workers in the FWP (usually less workers than by using actual cloud services), the performance in such setting is enhanced.

Using the cloud also poses its challenges regarding reliability. The research stated that one of the main concerns was the inconsistent behavior of the cloud service. Sometimes the workers would not start or could not connect to the OBS through the SSH and no debug information could be received. The platform offered by FWP is generally smaller, more controlled and provides log information of the potential shortcomings. This makes FWP a more reliable service at the cost of limiting the maximum size of the OBS network infrastructure to the hardware resources available in-house.

# BIBLIOGRAPHY

[1] Amazon Web Services LLC. Amazon Web Service. [WWW], 2011. [accessed on 17.08.2011]. Available at: `http://aws.amazon.com/`.

[2] Buck, J. Net::SSH Ruby Library. [WWW], 2008. [accessed on 27.09.2011]. Available at: `http://net-ssh.rubyforge.org/`.

[3] Crockford, D. RFC 4627: application/json. [WWW], 2006. [accessed on 27.09.2011]. Available at: `http://www.ietf.org/rfc/rfc4627.txt?number=4627`.

[4] Dr. Hipp, R. SQLite. [WWW], 2000. [accessed on 27.09.2011]. Available at: `http://www.sqlite.org/`.

[5] Fielding, R. T. Architectural Styles and the Design of Network-based Software Architectures. University of California, Irvine, 2000. Available at: http://www.ics.uci.edu/ fielding/pubs/dissertation/top.htm.

[6] Fowler, M. Patterns of enterprise application architecture. 2003. Addison-Wesley. p. 160-163.

[7] Free Software Foundation, Inc. Cpio. [WWW], 2000. [accessed on 16.10.2011]. Available at: http://www.ics.uci.edu/ fielding/pubs/dissertation/top.htm.

[8] Free Software Foundation, Inc. GNU General Public License. [WWW], 2007. [accessed on 22.2.2011]. Available at: `http://www.gnu.org/licenses/gpl.html`.

[9] Free Software Foundation, Inc. GNU Make. [WWW], 2010. [accessed on 12.06.2011]. Available at: `http://www.gnu.org/software/make/`.

[10] Gamma, E., Helm, R., Johnson, R., Vlissides, J. Design Patterns. 1995, Addison-Wesley. p. 128.

[11] Hansson. Ruby on Rails. [WWW], 2003. [accessed on 05.07.2011]. Available at: `http://rubyonrails.org/`.

[12] Hansson, D.H., Kemper, J. ActiveRecord. [WWW], 2009. [accessed on 02.07.2011]. Available at: `http://rubyforge.org/projects/activerecord/`.

[13] Johnson, S. Diagram showing overview of cloud computing. [WWW], 2009. [accessed on 04.10.2011]. Available at: `http://commons.wikimedia.org/wiki/File:Cloud_computing.svg`.

[14] Kneschke, J. lighttpd. [WWW], 2003. [accessed on 27.09.2011]. Available at: http://www.lighttpd.net/.

[15] Lubkin, D. DSEE: a software configuration management tool. [WWW], 1991. [accessed on 10.08.2011]. Available at: http://findarticles.com/p/articles/mi_m0HPJ/is_n3_v42/ai_10916486/.

[16] Marshall, P., Keahey, K., Freeman, T. Elastic Site: Using Clouds to Elastically Extend Site Resources. [WWW], 2010. [accessed on 16.10.2011]. Available at: http://www.nimbusproject.org/files/elasticsite_ccgrid_2010.pdf.

[17] Masatoshi, S. ERB - Ruby Templating. [WWW], 2003. [accessed on 27.09.2011]. Available at: http://www.ruby-doc.org/stdlib/libdoc/erb/rdoc/.

[18] Matsumoto, Y. Ruby. [WWW], 1995. [accessed on 27.09.2011]. Available at: http://www.ruby-lang.org/en/about/.

[19] MySQL AB. MySQL. [WWW], 1995. [accessed on 27.09.2011]. Available at: http://www.mysql.com/.

[20] Novell, Inc. OpenSUSE Build Service. [WWW], 2010. [accessed on 22.02.2011]. Available at: https://build.opensuse.org/.

[21] Novell, Inc. openSUSE KIWI Image System. [WWW], 2010. [accessed on 09.08.2011]. Available at: http://en.opensuse.org/Portal:KIWI.

[22] Peter M., Timothy G. The NIST Definition of Cloud Computing. [WWW], 2009. [accessed on 11.11.2011]. Available at: http://www.nist.gov/itl/cloud/upload/cloud-def-v15.pdf.

[23] Python Software Foundation. Python Programming Language. [WWW], 1990-2011. [accessed on 27.09.2011]. Available at: http://www.python.org/.

[24] Red Hat. RPM Packaging Manager. [WWW], 1993. [accessed on 28.09.2011]. Available at: http://rpm.org/.

[25] Reenskaug, T. MVC. [WWW], 1979. [accessed on 27.9.2011]. Available at: http://heim.ifi.uio.no/~trygver/themes/mvc/mvc-index.html.

[26] Ville Seppänen. Elastic Build System in a Hybrid Cloud Environment. Master's thesis, Tampere 2011. Technical University of Tampere.

[27] Silberschatz, A., Galvin, P. B.; Gagne, G. Operating System Concepts (8th ed.). 2010, John Wiley & Sons (Asia). p. 194.

[28] Sommerville, I. Software Engineering. 5th edition. 1998. Addison-Wesley. p. 690-696.

[29] Takahashi, M., Gotou Y. WEBrick documentation. [WWW], 2000. [accessed on 10.03.2011]. Available at: `http://www.ruby-doc.org/stdlib/libdoc/webrick/rdoc/index.html`.

[30] Tamski, M. OBS Internal Architecture. Tampere 2010, Nomovok. Unpublished report. 1 p.

[31] The Linux Foundation. MeeGo. [WWW], 2011. [accessed on 27.09.2011]. Available at: `https://meego.com/`.

[32] The MathWorks, Inc. MATLAB®. [WWW], 1994-2011. [accessed on 27.09.2011]. Available at: `http://www.mathworks.se/products/matlab/`.

[33] W3C. SOAP Version 1.2 Part 1: Messaging Framework (Second Edition). [WWW], 2007. [accessed on 28.09.2011]. Available at: `http://www.w3.org/TR/soap12-part1`.

[34] W3C. Extensible Markup Language (XML) 1.0 (Fifth Edition). [WWW], 2008. [accessed on 28.09.2011]. Available at: `http://www.w3.org/TR/REC-xml/`.

[35] Ylonen, T. RFC 4251: The Secure Shell (SSH) Protocol Architecture. [WWW], 2006. [accessed on 27.09.2011]. Available at: `http://tools.ietf.org/html/rfc4251`.

[36] Zissis D., Lekkas, D. Addressing cloud computing security issues. [WWW], 2010. [accessed on 11.11.2011]. Available at: `http://www.sciencedirect.com/science/article/pii/S0167739X10002554`.

# A. MATLAB® FWP SIMULATOR METHOD

```
% FlexibleWorkerPool efficiency analyzer - simul1
%
% Parameters:
% -simu_length            Time for one simulation
% -build_length           Vector containing the possible build lengths
% -base_prios             Base priorities for the build services
% -freq_by_bs             Probability of new build coming to the queue
% -worker_allocation      Worker allocation for dedication
% -random_seed            Seed for the rng
%
% Returns:
% -completed_by_worker    Completed build jobs by worker
% -completed_for_bs       Completed build jobs list by target build serviec
% -completed_total        Total completed builds
% -queued_total           Total builds queued up
% -usage_avg              Average workers in use
function [completed_by_worker, completed_for_bs, completed_total, \
          queued_total, usage_avg] = simul1(simu_length, build_length, \
          base_prios, freq_by_bs, worker_allocation, random_seed)

rng(random_seed)

build_services = size(base_prios, 2);
workers = size(worker_allocation, 2);

% Dynamically adjusted priorities
dyn_prios = base_prios;

 % Build queues
queues_by_bs = int64(zeros(1, build_services));

% Record keeping for completed builds
completed_for_bs = zeros(1, build_services);

% Workers
% Holds progress for individual workers
progress_by_worker = zeros(1, workers);
```

```
% Holds completion totals for workers
completed_by_worker = progress_by_worker;

% Allocated workers are considered dedicated
worker_dedication = worker_allocation > 0;
idle_avg = 0; % Average idle workers
queued_total = 0;

for j=1:simu_length,
  % Add builds to queues (by chance) -> will be either 0 or 1
  add_build_pr = int64(rand(1, size(queues_by_bs,2)) - 0.5 + freq_by_bs);
  queued_total = queued_total + sum(add_build_pr);
  queues_by_bs = queues_by_bs + add_build_pr;

  % Process the workers
  progress_by_worker = progress_by_worker - 1;

  % Find finished workers, -1 = no job, 0 = completed
  finished_workers = progress_by_worker == 0;

  % Add to completion statistics
  completed_by_worker = completed_by_worker + finished_workers;

  % Free the finished workers
  worker_allocation(not(worker_dedication)) = \
              worker_allocation(not(worker_dedication)) .* \
                ((-1 * finished_workers(not(worker_dedication))) + 1);

  % Reset the -1 to 0
  progress_by_worker = max(progress_by_worker, 0);

  % Start build jobs on dedicated worker hosts
  dedicated_workers = find(worker_allocation > 0 & worker_dedication);

  for d_alloc=1:size(dedicated_workers, 2),
    w = dedicated_workers(d_alloc);
    if (queues_by_bs(worker_allocation(w)) > 0) && progress_by_worker(w) == 0
      queues_by_bs(worker_allocation(w)) = \
                     queues_by_bs(worker_allocation(w)) - 1;
      completed_for_bs(worker_allocation(w)) = \
                     completed_for_bs(worker_allocation(w)) + 1;
      progress_by_worker(w) = build_length(randi(size(build_length, 2), 1));
    end
  end

  % Find the free dynamic worker count
  free_worker_count = \
          size(find(worker_allocation == 0 & not(worker_dedication)), 2);
  % Handle dynamic allocation
  for alloc=1:free_worker_count,
    % Update build service priorities
    dyn_prios = dyn_prios + base_prios;

    % Build services with no queues do not require service
    tmp_prios = dyn_prios .* (dyn_prios & queues_by_bs);

    % Find the index of the biggest priority
    serviced_bs = find(tmp_prios == max(tmp_prios), 1);
    if queues_by_bs(serviced_bs) > 0
      queues_by_bs(serviced_bs) = queues_by_bs(serviced_bs) - 1;
      completed_for_bs(serviced_bs) = completed_for_bs(serviced_bs) + 1;
      dyn_prios(serviced_bs) = base_prios(serviced_bs); % Reset the serviced p

      % Find the first free worker and allocate it and add the build job
      free_worker = find(worker_allocation == 0, 1);
      worker_allocation(free_worker) = serviced_bs;
      progress_by_worker(free_worker) = \
          build_length(randi(size(build_length, 2), 1));
    end
  end

  idle_avg = ((idle_avg * (j-1)) + size(find(worker_allocation == 0), 2)) / j;
end

%completed_by_worker
%completed_for_bs
completed_total = sum(completed_by_worker);
usage_avg = (workers-idle_avg);
```

# B. SIMULATION CODE 1: VARYING AMOUNT OF WORKERS

```
% Simulation 1
%
% Simulate through varying amount of workers for two build services
% with given frequencies.

% General variables
s_length = 100;
b_length = [5];
iterations = 1000;
seed = 'shuffle';

% Build services
base_prios = [1 1];
base_freq = [0.75 0.25];

dyn_avg_builds = [];
sta_avg_builds = [];

dyn_avg_completed = [];
sta_avg_completed = [];

worker_max = 6;

disp('Running simulation using varying worker counts')

disp('Step 1: Dynamically allocated workers')
```

```
for workers=2:2:worker_max,
    worker_allocations = zeros(1, workers);

    avg_builds = 0;
    avg_usage = 0;
    avg_completed = zeros(1, size(base_prios,2));

    for j=1:iterations,
        [c_by_w, c_for_bs, c_total, q_total, usage] = \
                  simul1(s_length, b_length, base_prios, \
                           base_freq, worker_allocations, seed);

        avg_builds = (avg_builds * (j-1) + sum(c_by_w)) / j;
        avg_completed = avg_completed + c_for_bs;
    end
    avg_completed = avg_completed / iterations;

    dyn_avg_completed = [dyn_avg_completed; avg_completed];
    dyn_avg_builds = [dyn_avg_builds avg_builds];
end

dyn_avg_completed
dyn_avg_builds

disp('Step 2: Static workers - divided 50-50')

for workers=2:2:worker_max,
    worker_allocations = [ones(1, workers/2) ones(1,workers/2)*2];

    avg_builds = 0;
    avg_usage = 0;
    avg_completed = zeros(1, size(base_prios,2));

    for j=1:iterations,
        [c_by_w, c_for_bs, c_total, q_total, usage] = \
                   simul1(s_length, b_length, base_prios, \
                            base_freq, worker_allocations, seed);

        avg_builds = (avg_builds * (j-1) + sum(c_by_w)) / j;
        avg_completed = avg_completed + c_for_bs;
    end
    avg_completed = avg_completed / iterations;
    sta_avg_completed = [sta_avg_completed;avg_completed];
    sta_avg_builds = [sta_avg_builds avg_builds];
end

sta_avg_completed
sta_avg_builds

plot(2:2:worker_max, dyn_avg_builds, '--r', \
      2:2:worker_max, sta_avg_builds, 'g');
title('Builds completed by worker count');
xlabel('Workers');
ylabel('Builds completed');
legend('Dynamic workers', 'Static workers');
```

# C. SIMULATION CODE 2: VARYING FREQUENCIES

```
% Simulation 2
%
% Simulate through varying build queue frequencies for two build services
% with given frequencies.

% General variables
s_length = 100;
b_length = [5];
iterations = 100;
seed = 'shuffle';

% Build services
base_prios = [1 1];

dyn_avg_builds = [];
sta_avg_builds = [];

dyn_avg_completed = [];
sta_avg_completed = [];

disp('Running simulation using varying build queue frequencies')

disp('Step 1: Dynamically allocated workers')

workers = 4;

worker_allocations = zeros(1, workers);

avg_builds = 0;
avg_usage = 0;
avg_completed = zeros(1, size(base_prios,2));

for freq=0:0.05:1,
    base_freq = [freq 0.25];
    for j=1:iterations,
        [c_by_w, c_for_bs, c_total, q_total, usage] = \
              simul1(s_length, b_length, base_prios, base_freq, \
                       worker_allocations, seed);

        avg_builds = (avg_builds * (j-1) + sum(c_by_w)) / j;
        avg_completed = avg_completed + c_for_bs;
    end
    avg_completed = avg_completed / iterations;

    dyn_avg_completed = [dyn_avg_completed; avg_completed];
    dyn_avg_builds = [dyn_avg_builds avg_builds];
end


dyn_avg_completed
dyn_avg_builds

disp('Step 2: Static workers - divided 50-50')

for freq=0:0.05:1,
    worker_allocations = [ones(1, workers/2) ones(1,workers/2)*2];

    avg_builds = 0;
    avg_usage = 0;
    avg_completed = zeros(1, size(base_prios,2));
    base_freq = [freq 0.25];

    for j=1:iterations,
        [c_by_w, c_for_bs, c_total, q_total, usage] = \
          simul1(s_length, b_length, base_prios, base_freq, \
                   worker_allocations, seed);

        avg_builds = (avg_builds * (j-1) + sum(c_by_w)) / j;
        avg_completed = avg_completed + c_for_bs;
    end
    avg_completed = avg_completed / iterations;
    sta_avg_completed = [sta_avg_completed;avg_completed];
    sta_avg_builds = [sta_avg_builds avg_builds];
end

sta_avg_completed
sta_avg_builds

plot(0:0.05:1, dyn_avg_completed(:,1), 'r', ...
     0:0.05:1, dyn_avg_completed(:,2), '--r', ...
     0:0.05:1, sta_avg_completed(:,1), 'g', ...
     0:0.05:1, sta_avg_completed(:,2), '--g', 'LineWidth', 2);
title('Simulation: Varying build frequency');
xlabel('OBS-1 build frequency');
ylabel('Builds completed');
legend('OBS-1 (dynamic)', 'OBS-2 (dynamic)', 'OBS-1 (static)', \
      'OBS-2 (static)');
```

# D. SIMULATION CODE 3: VARYING FREQUENCIES, VARYING BUILD LENGTHS

```
% Simulation 3
%
% Simulate through varying static build queue frequencies and varying
% build lengths for two build services.

% General variables
s_length = 100;
b_length = [4 4 8];
iterations = 200;
seed = 'shuffle';

freq = 0.4;

% Build services
base_prios = [1 1];

dyn_avg_builds = [];
sta_avg_builds = [];

dyn_avg_completed = [];
sta_avg_completed = [];

disp('Running simulation using varying build queue frequencies')

disp('Step 1: Dynamically allocated workers')
```

```
workers = 4;

worker_allocations = zeros(1, workers);

avg_builds = 0;
avg_usage = 0;
avg_completed = zeros(1, size(base_prios,2));

for freq=0:0.05:1,
    base_freq = [freq freq];

    for j=1:iterations,
        [c_by_w, c_for_bs, c_total, q_total, usage] = simul1(s_length, \
               b_length, base_prios, base_freq, worker_allocations, seed);

        avg_builds = (avg_builds * (j-1) + sum(c_by_w)) / j;
        avg_completed = avg_completed + c_for_bs;
    end
    avg_completed = avg_completed / iterations;

    dyn_avg_completed = [dyn_avg_completed; avg_completed];
    dyn_avg_builds = [dyn_avg_builds avg_builds];

end

dyn_avg_completed
dyn_avg_builds
```

```
disp('Step 2: Static workers - divided 50-50')

worker_allocations = [ones(1, workers/2) ones(1,workers/2)*2];

avg_builds = 0;
avg_usage = 0;
avg_completed = zeros(1, size(base_prios,2));

for freq=0:0.05:1,
    base_freq = [freq freq];

    for j=1:iterations,
        [c_by_w, c_for_bs, c_total, q_total, usage] = simul1(s_length,\
               b_length, base_prios, base_freq, worker_allocations, seed);

        avg_builds = (avg_builds * (j-1) + sum(c_by_w)) / j;
        avg_completed = avg_completed + c_for_bs;
    end
    avg_completed = avg_completed / iterations;

    sta_avg_completed = [sta_avg_completed;avg_completed];
    sta_avg_builds = [sta_avg_builds avg_builds];
end

sta_avg_completed
sta_avg_builds

plot(0:0.05:1, dyn_avg_completed(:,1), 'r', ...
     0:0.05:1, dyn_avg_completed(:,2), 'r', ...
     0:0.05:1, sta_avg_completed(:,1), 'g', ...
     0:0.05:1, sta_avg_completed(:,2), 'g', 'LineWidth', 2);
title('Simulation: Varying build lengths, same frequency');
xlabel('Build frequency (for both OBS)');
ylabel('Builds completed');
legend('OBS-1 (dynamic)', 'OBS-2 (dynamic)', \
         'OBS-1 (static)', 'OBS-2 (static)');
```