Investigating the Possibility of an Active/Active Highly Available Jenkins

Daniel Olausson

Master of Science Thesis
Stockholm, Sweden 2013

TRITA-ICT-EX-2013:119
Investigating the Possibility for an Active/Active Highly Available Jenkins

Author
Daniel Olausson
danola@kth.se

Thesis co-worker
Mikael Stockman
miksto@kth.se

Supervisor and Examiner
Johan Montelius

Industrial Supervisors
Adam Aquilon
Fatih Degirmenci

School of Information and Communication Technology
KTH Royal Institute of Technology

Keywords
High Availability, Load Sharing, Continuous Integration, Active/Active, Jenkins
Acknowledgement

First, I would like to thank our supervisors, Adam Aquilon and Fatih Degirmenci, for their guidance and support during our work; many great ideas were hatched during our discussions. Secondly, I want to thank all the people that made it possible for us to do this project.

Thanks to Sten Mogren for setting the project in motion to begin with. Thanks to Jens Hermansson for being a great manager, making us feel like one in the team.

Finally yet importantly, I want to thank my friend and co-worker in this thesis, Mikael Stockman, we make a great team.
Abstract

Jenkins can play an important role in software development. If Jenkins would be shut down because of failure, it would mean a halt in the development process. This thesis investigates the possibility of increasing the fault tolerance of Jenkins by running multiple instances of Jenkins in parallel and let them cooperate and share the load. In case failure occurs to one Jenkins, another Jenkins can take over. The major restriction was to not modify the source code of Jenkins.

The results of the investigation are that Jenkins can be modified to achieve high availability in the form of an active/active solution. Events in Jenkins related to build jobs can be replicated, these include: creating, modifying and deleting build jobs, also replication of builds can be replicated. The data replication technique had to be done asynchronously which introduced problems such as data collisions and ping-ponging. No solutions were found that could avoid data collisions; however, they can be detected and resolved.

A major issue is that other plugins in Jenkins could not be made aware of there being more than a single Jenkins. This issue and areas such as synchronization of user accounts, and the behavior of Jenkins slaves, needs to be further investigated.

Sammanfattning

Jenkins kan ha en viktig roll i utveckling av mjukvara. Om något skulle hända med Jenkins som leder till att den blir avstängd betyder det att utvecklingsprocessen kan bli påverkad. I den här rapporten undersöks möjligheter för att höja felsäkerheten hos Jenkins genom att kunna köra flera instanser av Jenkins parallellt. Dessa ska kunna samarbeta och dela lasten med varandra och om ett fel inträffar hos en Jenkins ska en annan Jenkins kunna ta över. Restriktionen var att källkoden av Jenkins inte fick modifieras.


Ett av de största problemen var att andra plugin i Jenkins inte kunde göras medvetna om att det fanns fler än en Jenkins. Detta problem, och delar så som användarkonton och Jenkinslavar behöver undersökas ytterligare.
# Table of Contents

1  **INTRODUCTION** ......................................................................................................................... 1

   1.1  **BACKGROUND** .................................................................................................................. 1

   1.2  **PROBLEM** ......................................................................................................................... 1

   1.3  **LIMITATIONS** .................................................................................................................... 2

   1.4  **RESEARCH APPROACH** .................................................................................................... 2

   1.5  **OUTLINE** .......................................................................................................................... 3

2  **HIGH AVAILABILITY** .................................................................................................................... 5

   2.1  **CLUSTER CONFIGURATION** ............................................................................................... 5

3  **ACTIVE/ACTIVE** ................................................................................................................................ 9

   3.1  **DATA REPLICATION** ......................................................................................................... 10

   3.2  **DATA COLLISIONS** ............................................................................................................. 11

   3.3  **PING-PONGING** ............................................................................................................... 15

   3.4  **REFERENTIAL INTENSITY** ................................................................................................. 15

4  **JENKINS** ........................................................................................................................................ 17

   4.1  **INTRODUCTION TO JENKINS** .......................................................................................... 17

   4.2  **INTERNAL STRUCTURE** .................................................................................................... 18

   4.3  **PLUGINS** ............................................................................................................................ 20

5  **RELATED WORK** ................................................................................................................................ 23

   5.1  **CLOUDBEES’ HIGH AVAILABILITY PLUGIN** ..................................................................... 23

6  **HIGH AVAILABILITY FOR JENKINS** ............................................................................................ 25

   6.1  **CLUSTER CONFIGURATION** ............................................................................................... 25

   6.2  **ADVANTAGES OF A PLUGIN** ............................................................................................... 26

   6.3  **PLUGIN LIMITATIONS** ....................................................................................................... 26

   6.4  **DATA REPLICATION FOR JENKINS** .................................................................................. 26

   6.5  **COMPARING RESOURCES IN JENKINS** .......................................................................... 32

   6.6  **DATA COLLISIONS** ........................................................................................................... 32

   6.7  **PING-PONGING** ................................................................................................................ 35

   6.8  **SIMULTANEOUS JOB START** ............................................................................................. 36

7  **CONCLUSION** .................................................................................................................................. 37

8  **FUTURE WORK** .............................................................................................................................. 38

9  **REFERENCES** .................................................................................................................................... 39
Definitions

Node
A computer, server, Java Virtual Machine (JVM) or an application.

Failure
An event that causes the node to malfunction, to not respond or misbehave, or that the actual machine is shut down.

Failover
A failover is the principle of switching users from a failed node to a healthy node.

Jenkins
Refers to the setup of one Jenkins master including any potential slaves it may have.

Single Point of Failure
In system that consists of many components, a single point of failure is a failure to a component that will cause all other components to fail.

Fault Tolerant
A component that is fault tolerant is not easily made to fail.

Build Job
A build job is a sequence of tasks that a Jenkins will execute.

Build
A build is an execution of a build job.

Database
A database is considered to be a place where an application can store its state persistently.

List of Abbreviations

API
Application Programming Interface

CI
Continuous Integration

HA
High Availability

HTTP
Hypertext Transport Protocol

RAM
Random Access Memory

TCP
Transmission Control Protocol

UDP
User Datagram Protocol

XML
Extensible Markup Language
List of Figures

Figure 2-1 An example of an active/standby system, a front end directs the users to the active application ......................................................... 6

Figure 3-1 An example of an active/active system, a front end spread the users across the active applications ....................................................... 9

Figure 3-2 An example of a collision using version numbers ......................................................... 13

Figure 3-3 An example of a non-collision replication using version numbers ................................. 13

Figure 4-1 An illustration of the structure of the build queue in Jenkins ............................. 19
List of Code Examples
Listing 4-1 An example of an implementation of the QueueDecisionHandler extension point ..................................20
Listing 6-1 Example of how to retrieve the XML file as a String from an item ..................................................27
Listing 6-2 How to get the name of a build job in the onDeleted method .........................................................27
Listing 6-3 An example of an implementation of the SaveableListener extension point ..................................28
Listing 6-4 Retrieving the build number in the onStarted method of a RunListener ..................................28
Listing 6-5 Retreiving the relative path in the onFinalized method .................................................................29
Listing 6-6 Creating a build job in Jenkins ........................................................................................................29
Listing 6-7 This code example describes the different operations that can be done when replicating build jobs ..........................................................................................................................30
Listing 6-8 Set the next build number to be just by a build job ........................................................................31
1 Introduction

In software development an increase in productivity is always welcomed. This can be achieved in different ways where this thesis will touch upon one of these, namely Continuous Integration (CI).

This chapter will give the background which will set the context of this thesis, followed by an explanation of the problems experienced in this area. Later the problem statement, objectives, research approach, limitations and the outline will be introduced.

1.1 Background

In software development it is common to work in teams, the size of a team can vary from a few to very many. Regardless of size the team will encounter problems when the individual pieces of work are going to be combined to form the product that they are working on.

Continuous integration is the practice of regularly integrating all software developers’ code with the software project, often several times per day. After integration has been made the code is built and automated tests can be run. The purpose is to reduce the problems that arise if the integration is made seldom, e.g. one developer’s code may become incompatible with the rest of the project.

A Continuous integration tool is often executed on a dedicated server referred to as a build server, ensuring a clean environment where building and testing can be done.

Jenkins is a popular tool for CI. It is an open source project with many contributors and it can be extended to increase the functionality for its user in the form of plugins [1, p. 3].

A possible Jenkins configuration consists of a Jenkins master and a set of slaves. The master is in charge of scheduling and monitoring build jobs, and it can distribute these build jobs to the slaves. A build job could for example be to build software.

1.2 Problem

Jenkins can have an important and central role in a software project. If the service that Jenkins provides would become unavailable it can result in a stop in productivity until Jenkins is available again.

A solution would be to make Jenkins more tolerant to faults. Failover solutions exist where one Jenkins is in standby and is ready to be started in the event of failure to the main Jenkins. Although this solution has better fault tolerance than nothing it is not without problems. A Jenkins that is in standby cannot be up and available for service until it has been started, the process of starting Jenkins can be time consuming. The downtime of Jenkins is reduced but during the start-up of the standby Jenkins the
service will still be unavailable. Another issue is the waste of computer power of having a standby Jenkins master waiting and not being used.

A better solution would instead be to have both Jenkins masters available and being able to share the load between them. This concept of having multiple applications running simultaneously and share the load is called an active/active solution. In this solution a failover will only take the time it takes to redirect users from the failed application to an active one.

As mentioned previously Jenkins can be extended by the use of plugins. Although though much can be done with a plugin, Jenkins is still restricted. The alternative would be to modify the source code of Jenkins, but this would mean that each time the Jenkins community releases a new version of the source code the same modification would have to be repeated. To prevent this, the modification would need to be accepted by the Jenkins community. This could take time and in the end not even get accepted.

1.2.1 Problem Statement
How should Jenkins be extended to become an active/active application so that two or more Jenkins masters can cooperate and share the load, and in the case of failure act as backups to each other? Which available techniques for high availability are applicable to Jenkins and which are not?

1.3 Limitations
This thesis focused on build jobs, their configuration and the execution of them. Areas related to other plugins, Jenkins slaves, the user accounts and security were not investigated.

Memory, hard drives, network and other hardware related areas was considered to be reliable and redundant. The split brain problem was not considered because the network is deemed to be reliable.

Modifications to the Jenkins source code were not considered.

1.4 Research Approach
A prototype was developed together with this investigation to confirm which techniques could actually be implemented for Jenkins. The prototype was developed as a plugin to Jenkins which used JGroups [2] to communicate with the other Jenkins masters.

JGroups is a toolkit that can create and maintain a cluster of nodes that can send messages to each other. The framework also provides support for failure detection.

By using JGroups more energy could be spent on investigating possible techniques for high availability, rather than spending it on a custom clustering solution.

The prototype was developed to handle multiple Jenkinsses. However, due to the need of a reliable broadcast implementation some parts are not working correctly for
more than two Jenkinses. Section 6.4.1, Communication Channel, describes why reliable broadcast is necessary.

1.5 Outline

Chapter 2 will briefly explain high availability and different node configurations. Chapter 3 explains the active/active configuration in more detail. Chapter 4 will describe Jenkins, a general introduction as well as an explanation its internal structure. Chapter 5 will present the related work that has been made with high availability for Jenkins. Chapter 6 will explain which techniques in high availability that can be applicable to Jenkins and which cannot. Chapter 7 presents the conclusions and chapter 8 discuss the future work.
2  High Availability

The purpose of high availability is to minimize the down-time of a service that is being provided [3]. To do this the system that provides the service has to be fault tolerant. If the system consists of a single computer one could increase the fault tolerance by introducing redundancy. For example the hard drives could be configured in a redundant array of independent disks (RAID), thus in a case of failure to one hard drive there is still others left that can be used.

This principle of redundancy can be expanded to include multiple network interfaces and multiple computers working together to achieve higher fault tolerance.

When an application is being provided in a high available fashion one can let each node in the system execute that application. The users of the application are then distributed across these nodes, and in the case of failure the users can be moved to another node. This act, the moving of users to another node that acts as a backup, is called failover.

The way of managing and distributing users between the nodes can be done in different ways. A common solution is to have one active node and one passive node. The passive node can act as a backup in the case of failure to the active node. Another solution is to have multiple active nodes where all nodes can serve the users.

The following sections will describe these configurations.

2.1  Cluster Configuration

A cluster is a set of nodes that together provide the user with a single interface. This means that a user will see the cluster as one system. Should failure occur at one node the functionality that the node provides can be transferred to a node that is alive.

The set of nodes can connect to a shared database where only the active node has the access rights. If more than one node can access the database at the same time there is a risk of database corruption if they try to update the same resource.

To detect if a node has failed, heartbeat messages are used. It works by letting each node in the cluster periodically send out heartbeat messages to its neighbours. If a node fails to send a heartbeat message the listening node will declare that node as failed and initiate a failover.

For the heartbeat to be reliable it needs to be highly available, this is often solved with redundant network interfaces [4, p. 2]. If no such reliability exists it could mean that a node sees another node as down but it actually is not, e.g. when a heartbeat message gets lost. This can lead to problems with split brain, which happens if the system is split into two parts because a failure to the network. Both parts will think that
the other part has failed and will continue to operate. The problem is that when the connection between them is restored the state will be inconsistent.

The nodes in the cluster can be configured in different ways, the three most common ways being active/standby, active/active and parallel database. This thesis focuses more on the active/active configuration which will be discussed in the next chapter, but a short description of each three is given below.

2.1.1 Active/Standby
In an active/standby configuration one node is active and another is in standby. The standby node can take over if the active one should fail. When the active node is alive the standby node can be idling or running less critical applications that can be terminated if needed.

When a failure to the active node happens its application will be restarted on the standby node. During the process of restarting the application the service that the application provides will be unavailable. How long it takes for the application to be restarted depends on the application itself but a delay of a couple of minutes is common [4, p. 6]. Figure 2-1 shows a simple example of this configuration.

![Figure 2-1 An example of an active/standby system, a front end directs the users to the active application](image)
2.1.2 Active/Active

In an active/active configuration all the nodes are active and executing the common application, where they have their own local copy of the application database. In the case of failure the users of the failed node can be redirected to another node which now has to be able to serve both its own users and the failed node's users. For this to work it is important that the local database at each node is synchronized with the others.

2.1.3 Parallel Database

In this type of configuration many nodes can run the same application against the same database. It is therefore critical that the database can handle simultaneous access to it. The Oracle’s Real Application Cluster (RAC) is an example of such a database [5].
3 Active/Active

In an active/active system high availability is achieved by spreading users across multiple nodes. All the nodes in the system are active and can share the load between them. Each node has a local copy of a common application database and these copies are constantly being synchronized with each other. Because all nodes have an identical copy of the database it means that the users can be switched over to use another node in the case of failure. Due to the synchronized databases the act of switching the users can be done fast, within seconds is common [4, p. 4].

A simple illustration of an active/active system is given in Figure 3-1 below.

![Figure 3-1 An example of an active/active system, a front end spread the users across the active applications](image)
3.1 Data replication

The local copy of the database at each node needs to be kept in synchrony with each other. To do that the changes made to each of the local copies of the database will have to be detected and applied to the local database at the other nodes. These operations are managed by a replication engine.

3.1.1 Replication Engine

A basic replication engine consists of three parts; the extractor on the source node that monitors the source database for changes, the applier on the target node that receives the changes from an extractor and applies it to the target database. The third part is the communication channel between the extractor and the applier.

The extractor will extract changes made at the source database by monitoring a change queue, where the change queue can be e.g. an audit trail or a change log [6, p. 2].

The applier will receive changes from an extractor at the replication engine of another node and apply these changes to its own database, the target database.

The communication channel is network connection between the replication engines that makes it possible for them to send information to each other.

In an active/active system there are two major techniques used to achieve synchronization between the database copies, namely, asynchronous data replication and synchronous replication.

3.1.2 Asynchronous Replication

Changes to the source database are detected by the replication engine which then sends these changes to the other databases where they can be applied.

Advantages

The application does not need to be aware of the underlying replication mechanism, therefore the application itself rarely needs to be modified. This makes asynchronous replication easy to implement [7, p. 60].

Since the application is independent of the replication engine the application does not have to wait until the replication has finished, this improves the performance and scalability of the system.

Disadvantages

During the time it takes for the replication engine to send the change to another replication engine and for that to apply the change, the two databases are out of sync with each other. This time delay is referred to as replication latency [7, p. 56].

Should the node fail with changes still in the change queue, it would mean that the changes will not be replicated to the other databases. However, if the change queue is
implemented as a file on disk the changes can be replicated when the node is back online.

Other problems with using asynchronous replication are data collisions and ping-pong. Data collisions can happen if nodes update the same resource within the time of the replication latency. Ping-pong will happen when the extractor cannot distinguish between a change made by the applier and a change made by the application itself.

3.1.3 Synchronous Replication

Synchronous replication does not have the issues that asynchronous replication has. When changes are to be replicated all the involved resources are locked by a read/write lock which will prevent data collisions. This means that either the changes are replicated or they are not, because all the resources have to be locked and if some resource cannot be locked the replication will not be performed.

Because all data items need to be locked before a change can be replicated there will be no data loss in the case of node failure.

Synchronous replication has its own issues. Because everything has to happen synchronously the application cannot continue until the database transaction has finished. This waiting time is known as application latency and can affect the application negatively, making it feel less responsive [8, p. 6].

Another problem is that deadlocks can occur if two applications try to lock a series of resources. One application acquires the lock for resource A then tries to resource B. The second application has acquired the lock for B and tries to lock A, the two applications are now in a deadlock.

The application often requires modification to be able to support synchronous replication technology [8, p. 7].

3.2 Data Collisions

In an active/active system with asynchronous replication the replication latency can introduce data collision problems. If a change is made at the same time at two nodes the data replication engine at both nodes will send this change to the other database. If the exchange happens within the time of the replication latency it will cause the databases to be out of sync with each other. The databases will have different versions of the same resource and none of them are correct.

To prevent this from happening there are primarily two ways of solving it. One way is to avoid the collisions all together. The other is to detect that a collision has happen and then resolve that collision.
3.2.1 Collision Avoidance

A great method of dealing with collisions is to avoid them altogether. The following are ways of accomplishing this.

Master node

A node can be selected to be the master, and the other nodes have to then contact this master in order to make changes. Since only one node makes changes to the database no data collisions can happen.

Synchronous replication

Synchronous replication can be used. All resources that are affected by the replication are protected by a read/write lock that prevents simultaneous modification.

Database partitioning

Similar to the master node approach but here the database is divided into partitions where each partition is owned by only one node. Only the owner of a partition can modify it which means that there will be only one node modifying a particular partition at a time resulting in no data collisions.

3.2.2 Collision Detection

If data collisions cannot be avoided, something has to be done to detect that it has happened. There are many methods for detecting data collisions. Below are a few ways of doing it.

Versioning

For each resource in the database two version numbers are kept, the previous and the current version number, which are incremented by one when updated. When that resource is replicated, the version numbers can be sent with it. At the receiving end the replication engine can verify that its current version number is the same as the received previous version number. If they differ there has been a collision, as can be seen in Figure 3-2.

Figure 3-3 shows an example of a non-collision replication.
Figure 3-2 An example of a collision using version numbers

Figure 3-3 An example of a non-collision replication using version numbers
Before-image Comparison

This method is very similar to the previous version number method but instead of comparing version numbers, the resources are compared instead.

The replication engine can keep a record of the previous version of a resource and when replicating send it together with the updated resource. At the receiving end the replication can engine check if its current version of the resource is the same as the received previous version. If they are the same there has been no data collision and the new version of the resource can be applied.

Periodic Database Comparison

Data collisions using this method are not detected right away. This has to be taken into consideration if it can be tolerated in the chosen application.

A process can periodically check the local database of each node and compare them for differences and if a collision is detected apply the appropriate method of resolving it.

3.2.3 Collision Resolution

A data collision has been detected and something has to be done to correct it. The nodes involved have to make a decision of which version to keep, it is important that the same decision is reached at all nodes.

Ways of resolving the collision are presented below.

Unique ids

A node precedence can be introduced by assigning a unique id to each node. In the case of a data collision the version of the resource that belongs to the node with the greatest id will be the version that is chosen for all nodes.

Timestamps

Timestamps can be used to distinguish which change is the latest. The resource with the latest timestamp will win.

Ignoring them

If updates are frequent the data collision can be corrected by itself when the updates are non-colliding.

Manual resolution

If data collisions are rare they can be resolved manually when detected.
3.3 Ping-Ponging

When a change is detected by a replication engine it will send that change to the other replication engines in the system. At the receiving end, that change will be applied. If the replication engine at the receiving end detects this newly applied change it will send it back to where it came from, and the original sender will do the same. The change will be sent back and forth indefinitely. This phenomenon is called ping-ponging or data looping.

In order to prevent ping-ponging, the replication engine has to be able to differentiate between a change made by the application, and a change being applied by the applier. These three are ways of doing it.

3.3.1 Database Partitioning

The database can be partitioned so that a specific resource is owned by only one node. Only the owner may update it which means that the replication engine can be configured to only replicate resources owned by its node.

3.3.2 Data Content

The content of the resource can have information about the source of the change. If the change originated at another node the replication engine can be configured not to replicate it.

3.3.3 Control Table

If it is not possible for the replication engine to determine where the change originated then a control table can be used. A control table is a table where the applier can enter a record of the change before it applies it to the target database. When the extractor then notices a change to the source database it will check the control table to see if it contains a record of that change. If the extractor finds a match it will not replicate the change.

3.4 Referential Integrity

When replicating data between nodes it is important that a sequence of operations made at the source node are made in the same order at the target node.

If the replication engine is multithreaded the applier can create a separate thread for each operation. The replication engine must make sure that the operation of each thread is made in the correct order.

A single threaded replication engine does not have this problem as long as the changes that are sent over the communication channel arrive in the correct order. If the underlying transport protocol is TCP it will be guaranteed that changes are being sent in the correct order. However, if UDP is used the changes may arrive in a different order than they were sent in. The replication engine has to be configured to handle this.
4 Jenkins

In this chapter Jenkins will be explained. The chapter starts with a brief introduction of Jenkins. After this the internal structure of Jenkins will be presented, followed by an explanation of how plugins work in Jenkins.

4.1 Introduction to Jenkins

Jenkins is a popular continuous integration tool for build servers [9]. Jenkins is open source and written in Java. The Jenkins project has a rich community of developers that work on the core, there is also many plugins being developed to extend the functionality of Jenkins. The lead developer is Kohsuke Kawaguchi which is also the creator of Jenkins [10]. He now works at CloudBees with their enterprise version of Jenkins which has a solution that provides high availability for Jenkins, which will be described in chapter 5.

More specifically Jenkins schedules and monitors the execution of repeated build jobs. A build job can be thought of as a sequence of tasks to be performed, for example, a build job could be to first compile your source code and later run tests on it and deliver the result. Each execution of a build job is called a build.

Jenkins can be managed in different ways. The primary way is through a web interface, but there is also a Command Line Interface (CLI) and a HTTP API [11] [12].

The way a build job is started, or triggered, can vary greatly. A common trigger is the source code management (SCM) trigger in which a change to a version control system, e.g. Git [13] or Subversion [14], is detected. For example when a software developer makes a change to the code base of a project, Jenkins can then detect this and schedule an execution of that build job. Common triggers include the timer trigger, for when build job execution is wanted on a regular basis, and the build job trigger when a build job should trigger another build job. The latter is useful for when build jobs depend on each other and executing one before the other could cause unknown behaviour.

One of Jenkins key strengths is the capability of extending it with plugins. A plugin can do anything from changing a small detail in the web interface to more advanced things like providing triggers for build jobs.
4.2 Internal Structure

This section will describe the internal structures that are of importance in this thesis. First off is an explanation of how Jenkins persists its data, followed by an explanation of how and where a build job is executed.

4.2.1 File System

Jenkins uses a directory referred to as the Jenkins home directory to stores details related to build jobs, builds and plugins. The configuration files in the Jenkins home directory are in the form of XML files. They can be re-serialized into objects when Jenkins starts and whenever a change to a configuration is made it will be written to disk. However, if the files are manually changed Jenkins will not recognize them until Jenkins is either restarted or all configurations are reloaded.

The Jenkins home directory also contains subdirectories, where the jobs and builds directories are explained below.

Jobs

In the /jobs directory Jenkins stores all build jobs that it has created. Each build job has its own directory that consists of a configuration file config.xml, a file nextBuildNumber and a directory /builds. The only content of the file nextBuildNumber is the next build number for the build job, which is incremented with one for every time the build job is executed. The /builds directory contains information about every execution of the build job.

Builds

In the /builds directory a new directory is created for each execution of the build job. This directory is named after the date and time of the execution and consists of three files, two XML files and a log file. The first file, builds.xml, describes how the execution went, how long it took, if it succeeded or not etc. The second file, changelog.xml, contains information about any change to the source code. The third file, log, contains the log output from the execution of the build job.

In Linux systems there are symbolic links in the /builds directory of a build job, they are pointing to each build directory. They are named after the build number of the build that they are pointing to.

4.2.2 Jenkins Slaves and Executors

When a Jenkins master decides to start the execution of a build job it can either do it itself or choose to delegate this task to a Jenkins slave. The slave’s only job is to execute this task and report the result back to its master.

Both the Jenkins master and the slaves can have a number of executors, where each executor can execute one build job. Before a build job is executed an executor needs to be assigned to it. This is done in the build queue.
4.2.3 Build Queue

A build job will be put into this queue when it has been triggered to start. It will wait in the queue to be assigned an executor that can run it. Only when an executor becomes available or the build job is cancelled will it leave the queue. The general notion is that only one instance of a build job can be in the queue at a given time and later only one executor may run it.

When a build job is triggered and about to be put into the build queue a check is made to confirm that it has the rights to do so. This decision can be modified from a plugin that has a class which extends the extension point ScheduleDecisionHandler.

The ScheduleDecisionHandler, extension points and plugins will be explained in the next section.

The build queue consists of a series of stages, which can be seen in Figure 4-1. Common for all stages in the queue is that a build job can be aborted at any time.

Figure 4-1 An illustration of the structure of the build queue in Jenkins

The build job begins its journey in the queue in the waiting-list. The next stage in the queue is the blocked jobs where the build job can be executed but are prevented to do so because, either another instance of the build job is currently running or that it is being blocked via a QueueTaskDispatcher. The QueueTaskDispatcher is another extension point that will be explained in the next section.

When the build job has passed the blocked job stage it will enter the buildables stage. Here the build job can be immediately executed but it is waiting for a free executor to be assigned to it.

The last stage is the pending stage where the build job is being handed over to an executor but the execution has not started yet.
4.3 Plugins

A great feature of Jenkins is that it can be extended with plugins. A plugin can extend many parts of Jenkins, customizing it to fit the goals of the user. Plugins use extension points to achieve this.

4.3.1 Extension Points

An extension point is a place in the code where a plugin can contribute features or functionality to Jenkins.

The extension points are interfaces or abstract classes that can be implemented or extended by a plugin. Normally the plugin will have inner classes that implement an extension point. When a plugin has an implementation of an extension point, that implementation has to be registered for Jenkins to be able to recognize it and use it. This is accomplished by annotating the class with the \texttt{@Extension} annotation, and when Jenkins call \texttt{getExtensionList(Class<T> extensionType)} a list of implementations of that extension point is returned.

In the example in Listing 4-1, an implementation of the \texttt{QueueDecisionHandler} is shown, it will deny a build job a place in the queue if the name of the build job contains “badJob”.

```java
@Extension
public static class MyQueueDecisionHandler extends
  QueueDecisionHandler {
  @Override
  public boolean shouldSchedule(Task p,
                               List<Action> actions) {
    if (p.getName().contains("badJob")) {
      return false;
    } else {
      return true;
    }
  }
}
```

Listing 4-1 An example of an implementation of the QueueDecisionHandler extension point

The continuation of this section will introduce extension points that are relevant for this thesis. A complete list of extension points is available here [15].

\textbf{SaveableListener}

The \texttt{SaveableListener} will be notified about save actions on objects that implement the \texttt{Saveable} interface. When notified the \texttt{onChange} method will be called:

\begin{verbatim}
onChange(Saveable o, XmlFile file)
  When a Saveable object is saved to disk this method will be called. It gets the Saveable object
\end{verbatim}
and its corresponding XML file that will be written to disk.

**ItemListener**

When changes are made to an Item the ItemListener will be notified. In Jenkins a build job is represented by an Item, Item's also implement the Saveable interface.

- **onCopied(Item src, Item item)** This method will be called when an Item has been created by copying from another Item.

- **onCreated(Item item)** This method will be called when an Item has been created.

- **onDeleted(Item item)** When an Item is about to be deleted this method is called.

- **onLoaded()** When Jenkins is starting up it will load all build jobs from disk and when all of them have been loaded this method is called.

- **onRenamed(Item item, String oldName, String newName)**
  After an Item has been renamed this method is called with both the old name and the new name.

- **onUpdated(Item item)** This method is called after the configuration of a build job has been updated.

**RunListener**

The Runlistener is a listener that will be notified about executions of build jobs.

- **onCompleted(R r, TaskListener listener)** When a build is completed this method will be called.

- **onDeleted(R r)** When a build is about to be deleted, this method will be called right before this happens.

- **onFinalized(R r)** After the build has been moved to the completed state and been written to disk, the onFinalized method will be called.
onStarted(R r, TaskListener listener)

This method is called when the execution of a build job has started.

QueueDecisionHandler

The *QueueDecisionHandler* is used when deciding if a build job should be placed in the build queue or not.

shouldSchedule(Queue.Task p, List<Action> actions)

This method is called when a build job is about to be placed in the queue. Jenkins will iterate over all implementations of *QueueDecisionHandler*’s and calling *shouldSchedule* for each of them. The build job will only be placed in the queue if all *QueueDecisionHandler*’s return true.

QueueTaskDispatcher

The *QueueTaskDispatcher* is used when deciding if a build job can be taken out of the build queue and if it can be executed.

canTake(Node node, BuildableItem item)

This method is called whenever the queue is deciding to execute the build job on a given node, where a node is the base type of a Jenkins slave.

Returning a null value means that the node assignment is acceptable, returning a non-null value means that the build job was blocked.

canRun(Queue.Item item)

The *canRun* method is called when the build job is ready to execute immediately. If a non-null value is returned the build job is blocked and placed in the blocked stage in the queue.
5 Related Work
The wish for a high availability solution for Jenkins has been expressed many times. As of writing this thesis there is only one commercial solution available that provides high availability for Jenkins, the high availability plugin from CloudBees. This plugin will be explained in this chapter.

5.1 CloudBees’ High Availability Plugin
CloudBees is a company that provides a non-free version of Jenkins, called Jenkins Enterprise [19]. This version of Jenkins offers a high availability plugin [20].

Their high availability plugin use the active/standby solution which means that only one Jenkins master will be active. The other Jenkins master will be in standby waiting for the active instance to fail.

Both the active and the standby Jenkins will be configured in a way so that they use the same Jenkins home directory. Normally this is not recommended because if both were active they could overwrite each other files. However, if only one Jenkins is active at a time no corruption of the files can occur.

If the standby Jenkins notices a failure to the active Jenkins, it can put itself in active mode and since they share the same home directory it already have the correct state. The time it takes for the standby Jenkins to become active depends on the time it takes for it to start up which is normally a few minutes. However this time will grow longer the more build jobs and builds that exists.

The communication between the Jenkineses and the detection of failure is managed by JGroups [2]. At the front end a reverse proxy is used to point the users to the active Jenkins instance, and when a failure occurs the users are redirected to the newly active instance.
6 High Availability for Jenkins

This chapter will describe which techniques that can be applied to Jenkins to achieve a level of high availability that fits the problem statement. Many of the techniques will not be possible to apply because of the restriction of not modifying the source code of Jenkins.

First the different kinds of clustering configurations will be compared to each other. This is followed by an explanation of how changes in Jenkins could be monitored and replicated. Later problems regarding to data collisions, ping-ponging and simultaneous jobs starts will be discussed.

6.1 Cluster Configuration

In a high availability solution for Jenkins there are different cluster configurations that can be used. The Jenkinses can either be in an active/standby, active/active or a parallel database configuration.

As mentioned in the problem statement the Jenkinses should be able to cooperate and share the load. This means that the active/standby configuration is not viable. That leaves the active/active and parallel database configurations.

6.1.1 Parallel Database

Jenkins uses a directory to persist its configurations as files. This means that if the Jenkinses would use the parallel database configuration they would have to share the same home directory. The problem with this is that the Jenkinses would change and create files without the other Jenkinses being aware of it. This will cause strange behaviours where one build job could exist on one Jenkins and not exist on another.

6.1.2 Active/Active

In an active/active configuration the Jenkinses can have separate home directories. This removes the problem that the parallel database configuration has with Jenkinses modifying files and them not being visible to other Jenkinses.

However, this introduces the problem of keeping the home directories synchronized using either synchronized or asynchronous replication. Another problem is how to make a Jenkins aware that one of its files being updated by the replication engine. To solve these problems in Jenkins there are two options, either to modify the Jenkins core or creating a plugin for Jenkins.

A plugin is the only choice because of the limitations, see section 1.3. The reasons why and the limitations of this choice is described in next two sections.
6.2 Advantages of a Plugin

Before explaining why a plugin is the better choice there are other approaches that also needs to be considered.

Jenkins can be accessed through a HTTP API where some functions are available. Therefore a more low level approach is possible where the home directory of each Jenkins could be monitored for changes. These changes could then be applied using the HTTP API. However, the HTTP API is limited and cannot be used to apply all possible changes that can occur. The Jenkins CLI suffers the same limitations.

A plugin has access to more functionality in Jenkins and can thus do more. The main benefit of using a plugin to provide high availability for Jenkins is that the Jenkins core and the plugins are nearly independent. There is a small possibility that an update to the Jenkins core will change some extension points that a plugin relies on, in that case the plugin might become faulty.

6.3 Plugin Limitations

There are limitations when using a plugin. A plugin in Jenkins rely on listeners to be notified of any changes that have occurred. The listeners will only get notified after the actual change has happen. There are also issues with security where certain data structures are private and not reachable from a plugin. The result of this is that synchronous replication cannot be used, and asynchronous replication is therefore the only option when using a plugin for Jenkins.

6.4 Data replication for Jenkins

Data replication in Jenkins has to be done asynchronously. It is also important that the data is updated in the RAM of Jenkins, not only on disk.

This section will explain how an asynchronous replication engine works in Jenkins.

6.4.1 Asynchronous Replication Engine

The replication engine is made up of three parts, the extractor, the applier and the communication channel. This section will describe how these can be done in Jenkins by using the functions available to a plugin.

Extractor

In the replication engine the extractor monitors the changes made to the source database. When the extractor detects a change it will send it to the target Jenkins over the communication channel.

By letting the plugin have classes that implement extension points which can be notified about changes in Jenkins, the plugin will detect changes and decide what to do.
Build Jobs

Changes made to build jobs are detected by using the ItemListener extension point. When a build job is created or modified the methods onCreated(Item item) and onUpdated(Item item) will be called. Both of these events mean that there is a new version of a build job configuration that has to be sent to the other Jenkinuses.

The easiest way would be to just send the item to the other Jenkinuses, but these are unfortunately not serializable. However, the serialized version of the object, the XML file can be sent. Listing 6-1 shows how the XML file can be retrieved as a String, which can later be sent to the target Jenkins.

```java
@Override
public void onUpdated(Item item) {
    String xml = Items.getConfigFile(item).asString();
    // Send xml to the target Jenkins
}
```

Listing 6-1 Example of how to retrieve the XML file as a String from an item

When a build job is renamed or deleted the methods onRenamed(Item item, String oldName, String newName) and onDeleted(Item item) is called. In these cases the build job configuration does not need to be sent. When deleting a build job only the name needs to be sent, and when renaming a build job the old name and the new name has to be sent. The name of the build job is needed to find the build job at the target Jenkins. Listing 6-2 shows how to get the name of the build job.

```java
@Override
public void onDeleted(Item item) {
    String jobName = item.getName();
    // Send jobName to the target Jenkins
}
```

Listing 6-2 How to get the name of a build job in the onDeleted method

The ItemListener works well in most cases but not always, there are ways of modifying a build job without the ItemListener being notified. This is the case in the 1.502 version of Jenkins.

To go around this problem the SaveableListener can be used, an example is shown in Listing 6-3. This listener is notified of all changes made to a build job but it cannot distinguish between them. It cannot for example know if a build job has been created or if the configuration has been changed.

If all changes are going to be detectable by the plugin it therefore has to implement not only the ItemListener but also the SaveableListener. The problem with doing this is that an update to the build job configuration will trigger both listeners which will then cause two replications to happen. To prevent such behaviour the ItemListener can be
used solely for detecting when a build job is created, deleted or renamed. The 
SaveableListener can be used to monitor any updates to the build job configurations, it 
will be notified through the onChange(...) method.

```java
@Extension
public static class MySaveableListener extends SaveableListener {
    @Override
    public void onChange(Saveable o, XmlFile file) {
        if (o instanceof Job<?, ?>) {
            String jobName = ((Job<?, ?>) o).getName();
            try {
                String xml = file.asString();
            } catch (IOException e) {
                e.printStackTrace();
            }
        }
    }
}
```

Listing 6-3 An example of an implementation of the SaveableListener extension point

Builds

The execution of build jobs can be monitored by the RunListener. It will be notified 
when a build is started, completed, finalized and deleted. The builds are different from 
build jobs in the way that a build consists of multiple files that needs to be replicated.

The onStarted(...) method is invoked when a build has been started. The build 
number can be replicated here which will have been incremented by one. How this is 
done is shown in Listing 6-4.

```java
@Override
public void onStarted(Run<?, ?> r, TaskListener listener) {
    int buildNumber = r.getParent().getNextBuildNumber();
    String jobName = r.getParent().getName();
    // Replicate build number
}
```

Listing 6-4 Retrieving the build number in the onStarted method of a RunListener

When the onFinalized method is invoked the build has finished its execution and all 
the files have been written to disk.

To replicate the build the files has to be sent to the target Jenkins. The files cannot 
be retrieved directly from the Run Object, they have to be manually retrieved from disk. 
The path to the directory where the files are can be retrieved from the Run Object. The 
problem is that the path is absolute meaning that the path starts from the root of the 
file system and not from the Jenkins home directory. This can be a problem if the 
Jenkins home directory is located at different places on the different machines that run
Jenkins. The applier will then try to create a file in a directory that does not exist. It is therefore important to make this path relative to the Jenkins home directory. This is shown in Listing 6-5.

```java
@Override
public void onFinalized(Run<?, ?> r) {
    String jobName = r.getParent().getName();
    // Get the absolute path
    String absPath = r.getRootDir().getAbsolutePath();
    // Split the path and save the part which starts from /jobs
    String relPath = absPath.split("(?=jobs)")[1];
    // Read all the files found in absPath
    // and send them together with the relPath
    // to the target Jenkins
}
```

Listing 6-5 Retrieving the relative path in the onFinalized method

**Applier**

In the replication engine the applier receives changes from an extractor on another Jenkins master. The applier’s task is to apply these changes to its own Jenkins master.

It is important that the applier tries to apply these changes as if they were made by the Jenkins itself. Otherwise Jenkins might not be aware of that its files have been changed. There can also be other that plugins have listeners that needs to be notified in case a change is made to any of the configurations in Jenkins.

By applying the changes in this matter it creates problem for the extractor at the same Jenkins. The listeners that the extractor relies on will get notified when the applier applies the changes, this creates ping-ponging. Methods of dealing with ping-ponging are described in section 6.7.

**Applying Build Jobs**

The applier will receive different kinds of operations that are related to build jobs. It will receive operations to create, modify, rename or to delete a build job. To apply these operations the applier use internal methods in Jenkins.

To create a build job the applier use a method called `createProjectFromXML`, this is shown in Listing 6-6.

```java
Jenkins.getInstance().createProjectFromXML(String name, InputStream xml)
```

Listing 6-6 Creating a build job in Jenkins

The other operations require that the *Item* of a build job is retrieved. The *Item* has methods that will enable these operations.

To modify, rename and to delete a build job can be done as described in Listing 6-7.
Listing 6-7 This code example describes the different operations that can be done when replicating build jobs

Applying Builds

There are three operations that the applier can receive that relate to builds. One is to create a build, the second is to delete a build and the third is to update a build number. Operations on builds are very limited, a plugin in Jenkins is not allowed to create or modify a build. However, deleting builds is not a problem.

To create a build in Jenkins without having access to the appropriate method can be done by reloading the build job that the new build belongs to. The first step is to create the files on disk that it received from the extractor and later start a reload of that job. What this does is similar to what Jenkins does when it is starting up. When Jenkins is starting up it will go through all build jobs, and for each build job go through each build, and loading everything into memory. Although a build can be created in Jenkins it is not a very effective method. If the build job has many builds then creating a new build will cause a reload of all builds, not only the new one, which might take time.

Newer versions of Jenkins has a feature called lazy loading which means that Jenkins only loads a portion of the builds, if more builds needs to be accessed they will be loaded at that time. This will reduce the time for creating a build manually with the proposed method. The best thing would be to have access to the methods in Jenkins that can create and modify a build.

Updating the build number for a build job can be done as shown in Listing 6-8.
TopLevelItem topLevelItem = jenkins.getItem(jobName);
if (topLevelItem instanceof Job) {
    Job<?, ?> job = (Job<?, ?>) topLevelItem;
    try {
        job.updateNextBuildNumber(nextBuildNumber);
    } catch (IOException ex) {
    }
}

Listing 6-8 Set the next build number to be just by a build job

Communication Channel

In the replication engine the communication channel is responsible to create a network connection that the extractor can use to send changes to an applier.

In a Jenkins setup with more than two Jenkins masters there has to be some kind of reliable broadcast implemented.

Consider the case when there are three Jenkineses, A, B and C, that are connected to each other. Jenkins A modifies the configuration of a build job and the change is then replicated to Jenkins B and C. The change is successfully sent to Jenkins B but before Jenkins A can send the change to Jenkins C, a failure occurs to Jenkins A. The result is that Jenkins B and Jenkins C now have different versions of the same resource and they are not aware of it. If reliable broadcast is used it is guaranteed that every Jenkins in the system will receive the change, even if a failure to the sender occurs.

Reliable broadcast was not implemented in the prototype.
6.5 Comparing Resources in Jenkins

The next section describes how data collisions are handled in Jenkins. Some of the techniques involved in data collisions rely on comparing two resources to verify that they are either equal or different. In Jenkins this becomes a problem because many of the resources being sent are objects converted to XML files, which are difficult to compare. A way of solving this would be to let Jenkins re-serialize the XML file into an object which could then be compared. Unfortunately, this does not work because there is no `equals` method implemented, only the default `equals` found in the Object class is available.

To compare the resources the XML files would have to be interpreted by an XML parser which could then build a data structure that can be compared.

This was not implemented in the prototype.

6.6 Data Collisions

When using asynchronous replication there are problems with data collision. The way of dealing with collisions is either to avoid them or to detect and resolve them.

6.6.1 Collision Avoidance

To avoid data collisions Jenkins would have to be able to apply any of the following techniques.

**Master Node**

A single Jenkins in the system would have to be selected as the master. The master would be the owner of all build jobs. Only the master would be able to create, change or delete build jobs.

However, these kinds of restrictions are not possible because there is no way for a plugin to know when a resource is going to be updated. Even if it could there is no option in Jenkins to prevent it from happening.

**Database partitioning**

This is similar to the master node idea but instead of one Jenkins being the master of all the build jobs the ownership would be distributed. The problem is the same as for the master node idea, that these kinds of restrictions are not possible.

**Synchronous replication**

Using synchronous replication is not possible due to the fact that plugins cannot be notified of changes to Jenkins before they happen.
6.6.2 Collision Detection

Data collisions cannot be avoided in Jenkins which means that they would have to be detected and resolved instead. This section will discuss the three techniques previously mentioned in section 3.2.2.

Versioning

Versioning in Jenkins can be done by letting the plugin store version numbers for each resource in the Jenkins home directory. These version numbers can be stored in a file in the same directory as the resource.

Before-image Comparison

This approach is similar to versioning but instead of storing version numbers, the previous version of the resource itself is stored. This approach relies on comparing resources which is a problem, and is described in section 6.5.

Periodic Database Comparison

The plugin can periodically initiate a comparison of the other Jenkinses resources. In Jenkins there is an extension point called AsyncPeriodicWork that can be used for this. This extension point is an extended version of the PeriodicWork extension point, where the difference is that AsyncPeriodicWork is specially designed for tasks that can be time consuming. A separate thread will be spawned so that the plugin will not interfere with the normal operation of Jenkins [21].

This approach relies on the same principle as the before-image comparison approach. That is, it compares resources with each other, and therefore has the same problem.

6.6.3 Collision Resolution

When a data collision has happen there are several ways Jenkins could resolve them. Any of the following techniques are applicable to Jenkins to resolve collisions.

Unique ids or Timestamps

The plugin at each Jenkins will be given a unique id that can be used to create a node precedence. A plugin with a greater id will have the power when deciding which versions of a resource that should be chosen in the case of a collision.

Another way of creating node precedence is to use timestamps. When a change to a resource is made the time will be stored. When there is a collision and a decision of what version to choose, the version of the resource with the latest timestamp will be chosen.
**Ignoring them**

A way of dealing with collisions is to ignore them. This technique can be used when knowing that there will be frequent non-colliding updates in the near future. The collision will then be resolved automatically.

However, this technique is not exact and leaves room for unexpected behaviour and should therefore not be the first technique to consider.

**Manual resolution**

Data collisions may be rare or frequent depending on the number of Jenkinses in the system and on the amount of load. If data collisions are rare a system administrator could be contacted by the plugin when a collision is detected. The system administrator could then resolve the collision manually.

**An Ordering Problem**

Data collisions can be detected and resolved as described in the previous section. There is however another problem that needs to be considered.

In a system with three Jenkinses, A, B and C, the order in which a resource is updated becomes an issue. If Jenkins A and Jenkins B update the same resource at the same time there is an uncertainty to which order the updated resources will be received at Jenkins C. The update from Jenkins A could be delayed by for example network related problems, which will cause the update from Jenkins B to be received first. Jenkins C will then get the resource from A and a data collision is detected. If timestamps are used all Jenkinses will choose the correct resource.

A more problematic scenario is when Jenkins A updates a resource, Jenkins B receives this update and also make an update to the resource. Before the initial update from Jenkins A has been received by Jenkins C, the updated resource from Jenkins B is received.

This can for example allow an event about a build job modification to arrive before the event that creates the build job has arrived. The modification event is dropped. The result is that Jenkins C will have version of the build job configuration that is different from the ones at Jenkins A and B.

To solve this causal order broadcast needs to be implemented. Causal order broadcast ensures that events are delivered in order of causality.
6.7 Ping-Ponging

Ping-ponging, or data looping, can happen if the replication engine cannot distinguish a difference between a change applied by the applier and a change made by the actual application.

In Jenkins this happens when the plugin receives a change from another plugin, and applies that change to Jenkins. The listeners will then be notified of a change in Jenkins but there is no way of knowing the cause of that change, if it was caused by the applier or Jenkins.

The general solution to this problem is to make the replication engine capable of distinguishing the cause of the change. Only one of the three techniques explained in this section will be applicable to Jenkins, the control table.

6.7.1 Database Partitioning

This technique has previously been discussed in chapter 2 both as a way of preventing ping-ponging and as way of avoiding collisions.

Database partitioning is not a possible solution for the ping-ponging problem in Jenkins for the same reasons that it is not a solution for avoiding data collisions, see section 6.6.1. It is not possible for a plugin to intercept and prevent access to a resource without modifying the source code of Jenkins.

6.7.2 Data Content Tracking

If the resource contains information about which Jenkins it originates from the replication engine could chose to only replicate changes coming from its local Jenkins. The problem with this technique is that Jenkins itself would have to add this information to the resource before the plugin is notified. That would require modification to the source code of Jenkins.

6.7.3 Control Table

Implementing a control table is the only technique that does not require any modifications to the source code of Jenkins.

The plugin will keep a record of received replicated changes in the control table. When the change is applied the plugin will get notified and it can then check in the control table for an entry for this change. The plugin will only replicate the change if the control table does not contain an entry.

The problem of comparing resources in Jenkins will also be notable in this case if the resource itself is put into the control table.
6.8 Simultaneous Job Start

In a setup with two or more Jenkinses the build job configurations will be identical. This can create problems when the build jobs are configured to execute at a given time causing all Jenkinses involved to start the same build job simultaneously. This behaviour is not acceptable because it could make the results of the build job execution corrupt. On a single Jenkins setup the build queue makes sure that only one instance of a build job can be executing, unless it is specified in the configurations.

To find a solution to this problem when using a plugin for Jenkins the QueueTaskDispatcher and QueueDecisionHandler extension points are of interest.

6.8.1 A Simple Approach

The most simple and forward approach would be to check each Jenkins if it is currently executing the build job. If none of the other Jenkinses are executing the build job, the build job can be started.

This approach is unfortunately not possible with a plugin as of writing this thesis. If two or more Jenkinses decide to simultaneously try to start a build job they will all succeed, because none of the Jenkinses are currently executing the build job. This means that they will all proceed to start the same build job at roughly the same time. This is a typical problem in concurrency which is usually solved with a lock.

To be able to solve this problem in Jenkins the source code would have to be modified to make the process of starting a build job mutual exclusive. Meaning that when a Jenkins first decides to try and start a build job, a lock will be taken. This lock makes sure that when other Jenkinses tries to start the build job they cannot because the lock is locked. The lock is later released by the Jenkins who acquired it when the build job has successfully been started or if it has been blocked.

6.8.2 A Better Approach

A better approach does exist and was developed but cannot be disclosed due to commercial reasons.
7 Conclusion

Jenkins is a tool for continuous integration used in software projects to automate the process of building and testing software. Today Jenkins is a single point of failure meaning that if any failure should happen to Jenkins no building or testing could be done. This thesis has investigated the possibility of making Jenkins more highly available by letting two or more Jenkins masters work together and share the load. The restriction being that the source code of Jenkins could not be modified. That meant that a plugin for Jenkins was the method of choice to make it possible. Different techniques and solutions for making highly available systems were studied, and together with a thorough investigation of the Jenkins source code, conclusions could be made. To test and verify the techniques that were deemed viable, a prototype plugin for Jenkins was created. The prototype also opened up doors to problems and new ideas that could otherwise have been missed, should a solely theoretical investigation been made.

Many techniques for high availability have been presented in sections 2 and 3. Some are applicable to Jenkins and some are not. The result of this thesis shows that Jenkins can be made into a highly available application. Events made at one Jenkins master can be replicated to another; these include creating, changing and deleting a build job. Events regarding execution of build jobs can also be replicated. A big problem when having multiple Jenkins masters is that the other plugins in Jenkins are not aware of there being more than one Jenkins. To solve this problem the source code of Jenkins needs to be modified, however no obvious solution was found.

Two different cluster configurations for high availability were considered: the parallel database and the active/active approach. These were considered because they enable multiple nodes to be active and sharing the load at the same time. Other approaches such as the active/standby will have a node in standby that cannot cooperate with the active one. The parallel database approach was discarded early because when applied to Jenkins it would mean that the Jenkins masters would have to share the same home directory. This configuration could therefore lead to inconsistency when a file is modified without other Jenkins masters being aware of it. The active/active approach turned out to be the better choice.

When the active/active approach is applied to Jenkins, the events at each Jenkins would have to be replicated. It was shown that this could only be done asynchronously which lead to problems with data collisions and ping-ponging. Solutions for data avoidance were shown in section 6.6.1 to not be applicable to Jenkins without modifications to the source code. Therefore, data collisions would have to be either detected and resolved or deemed ignorable. The problem with ping-ponging was shown in section 6.7 to be solved with the control table technique.
In conclusion, it has been shown that a highly available Jenkins can be achieved, though with several issues. The Jenkins masters can cooperate and share the load by spreading users across the cluster, and in the case of failure, the users can be moved to a healthy Jenkins master.

8 Future Work

This investigation did not cover some important parts of Jenkins, these needs to be studied further. As described earlier a big problem is that the other plugins are not aware of it being more than one Jenkins master. No obvious solution was found to this problem, and therefore a more focused investigation in this area has to be made.

Other important areas not covered by this investigation include: synchronization of user accounts, the security for these user accounts and how the Jenkins slaves behaves in a setup with multiple Jenkins masters.
9 References


