Investigating the Possibility of an Active/Active Highly Available Jenkins

Mikael Stockman

Master of Science Thesis
Stockholm, Sweden 2013

TRITA-ICT-EX-2013:118
Investigating the Possibility of an Active/Active Highly Available Jenkins

Master of Science Thesis
2013-06-03

Author
Mikael Stockman
miksto@kth.se

Thesis co-worker
Daniel Olausson
danola@kth.se

Academic Supervisor and Examiner
Johan Montelius, KTH

Industrial Supervisors
Fatih Degirmenci
Adam Aquilon

School of Information and Communication Technology
KTH Royal Institute of Technology
Acknowledgements

First, I would like to thank my supervisors Adam Aquilon and Fatih Degirmenci for their help and guidance throughout this investigation. Many ideas and solutions to issues were formed in the many discussions we had.

I would also like to thank Sten Mogren and everyone else who initially came up with the idea, formulated the problem to investigate, and helped us getting started with this thesis.

I am also very thankful to Jens Hermansson for being a great, inspiring manager, making you feel as a part of the team.

Finally, I would like to thank my friend and thesis co-worker Daniel Olausson, who made this thesis a joy.
Abstract

Jenkins is a continuous integration engine, which is a single point of failure since it is designed to run on one server only. If the server is brought down for maintenance, Jenkins will be unavailable. This report investigates the possibility of having multiple Jenkins masters running in parallel that can cooperate and share the load. If one Jenkins is not able to respond to requests, another Jenkins should be there and take over the load and the responsibilities. The restriction is that the source code of Jenkins must not be modified.

The results show that the core features of Jenkins related to jobs and builds can indeed be adapted to a work in an active/active solution, but that other external plugins for Jenkins cannot be guaranteed to work properly. The data replication between Jenkins masters must be done asynchronously and data collisions will therefore be an issue that needs to be considered. In conclusion, modifications are required on Jenkins but also the plugins, if every feature of Jenkins is to be adjusted to work in an active/active solution.

Sammanfattning


Table of Contents

1 INTRODUCTION ............................................................................................................. 1
  1.1 BACKGROUND ........................................................................................................... 1
  1.2 PROBLEM .................................................................................................................. 2
  1.3 LIMITATIONS ............................................................................................................ 2
  1.4 RESEARCH APPROACH ............................................................................................ 3
  1.5 OUTLINE ................................................................................................................... 3

2 HIGH AVAILABILITY ........................................................................................................ 5
  2.1 HIGH AVAILABILITY CLUSTER CONFIGURATIONS .................................................. 5
  2.2 ACTIVE/ACTIVE .......................................................................................................... 7

3 ISSUES WITH ASYNCHRONOUS REPLICATION ........................................................ 11
  3.1 REPLICA Vinyl to several NODES .............................................................................. 11
  3.2 DATA LOSS ................................................................................................................ 11
  3.3 DATA COLLISIONS .................................................................................................... 11
  3.4 REFERENTIAL INTEGRITY ......................................................................................... 16
  3.5 PING PONGING OR DATA LOOPING ...................................................................... 16

4 JENKINS ........................................................................................................................ 19
  4.1 A SHORT INTRODUCTION TO JENKINS ................................................................. 19
  4.2 THE INTERNAL JENKINS STRUCTURE ..................................................................... 20
  4.3 JENKINS PLUGINS ..................................................................................................... 22

5 RELATED WORK ............................................................................................................. 27
  5.1 CLOUDBees’ High Availability Plugin ..................................................................... 27

6 HIGH AVAILABILITY APPLIED TO JENKINS ................................................................ 29
  6.1 DIFFERENT CLUSTER CONFIGURATIONS ............................................................... 29
  6.2 ADVANTAGES OF A PLUGIN .................................................................................... 29
  6.3 LIMITATIONS WITH A PLUGIN ............................................................................... 30
  6.4 BACKWARD COMPATIBILITY FOR EXISTING PLUGINS ....................................... 30
  6.5 REPLICATING JOBS AND BUILDS ......................................................................... 30
  6.6 THE ASYNCHRONOUS REPLICATION ENGINE ...................................................... 31
  6.7 DATA COLLISIONS .................................................................................................... 37
  6.8 COLLISION RESOLUTION ......................................................................................... 39
  6.9 DATA LOOPING ......................................................................................................... 40
  6.10 SIMULTANEOUS JOB STARTS .................................................................................. 41
  6.11 JOINING A JENKINS CLUSTER ............................................................................... 42

7 CONCLUSIONS ............................................................................................................... 43

8 FUTURE WORK .............................................................................................................. 44
Definitions

Some words have different meanings depending on the context. Here is a list of such words, and how they are used in this thesis.

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

Failure/Fault
An event that causes a node to stop responding to any request. Such an event can for instance be that the application has stopped responding or the entire server was shut down.

Failover
When a node fails, a backup node can step in and run the applications on behalf of the failed node. The process where the failed node is replaced by the backup node is called a failover.

Jenkins
Refers to a Jenkins master, or a Jenkins system as a whole including slaves etc.

Job/Build job
A task in jenkins which can be executed repeatedly. It can be seen as a series of instructions of what should be performed, when the job is executed.

A build
An execution of a job

Single point of failure
A part in a larger system that consists of many parts, which if it fails causes the entire system to fail.

Database
Any kind of organized collection of data, including the internal state of an application.
**List of Abbreviations**

<table>
<thead>
<tr>
<th>Abbreviation</th>
<th>Full Form</th>
</tr>
</thead>
<tbody>
<tr>
<td>API</td>
<td>Application Programming Interface</td>
</tr>
<tr>
<td>CI</td>
<td>Continuous Integration</td>
</tr>
<tr>
<td>CLI</td>
<td>Command Line Interface</td>
</tr>
<tr>
<td>HA</td>
<td>High Availability</td>
</tr>
<tr>
<td>HTTP</td>
<td>Hyper Text Transfer Protocol</td>
</tr>
<tr>
<td>TCP</td>
<td>Transmission Control Protocol</td>
</tr>
<tr>
<td>UDP</td>
<td>User Data Protocol</td>
</tr>
<tr>
<td>URL</td>
<td>Uniform Resource Locator</td>
</tr>
<tr>
<td>XML</td>
<td>Extended Markup Language</td>
</tr>
</tbody>
</table>
List of Figures

Figure 2-1 A schematic illustration of an active/passive system. Dashed connections are not used. ................................................................. 6

Figure 2-2 Schematic illustration of an active/active system ........................................... 7

Figure 3-1 A sequence diagram illustrating a situation where no data collision occurs... 14

Figure 3-2 A sequence diagram illustrating a situation where a data collision occurs..... 14

Figure 4-1 The possible stages for jobs in the queue ......................................................... 21

Figure 6-1 An illustration of the issue with late messages when replication is done to more than one Jenkins. Jenkins A makes a modification that is replicated and first received by B. B makes a new modification that is replicated and received by C before A’s modification......................................................... 39

Figure 6-2 An illustration of a correct order of delivery, preserving causal order ............ 40
List of Code Examples

Listing 4-1 A code snippet from ItemListener.class showing how onCreated(...) is invoked on all defined extensions that is an ItemListener. The method all() returns every extension of the type ItemListener......................................................22

Listing 6-1 Example of how an object can be serialized into XML that can be used to create a clone of the job .................................................................32

Listing 6-2 Example of how to get the XML representation of a job, from the XML file in the home directory ........................................................................32

Listing 6-3 Example on how to handle renamed and deleted jobs ........................................33

Listing 6-4 Example of how to filter out jobs when using the SaveableListener ...............33

Listing 6-5 Example on how to get the next build number for a job ....................................34

Listing 6-6 Example on how to extract the path to the build file directory relative to the Jenkins home directory ......................................................35

Listing 6-7 How to programmatically create jobs in Jenkins. The argument name is the name of the job, and xml is the XML data that describes the configuration of the job.................................................................................................36

Listing 6-8 How to retrieve a job from a job name. The object job is necessary if a job is to be renamed, updated or deleted.........................................................36

Listing 6-9 How to rename, delete and update a job..............................................................36

Listing 6-10 How to update the next build number for a job................................................37
1 Introduction

Improving productivity in software development is always of interest. Continuous integration (CI) is a way of working where developers integrate their code others’ at a daily basis. This way every developer has access to a recent version of the common code base, and integrating other developer’s code will be less troublesome. In CI, usually a CI Server automates tasks such as building and testing the latest version of a software project. A CI Server provides a clean environment and avoids problems where project builds and tests depend on the environment of a developer’s local computer.

1.1 Background

A very popular application for running automated tasks is Jenkins [1]. Jenkins is an open source project with a strong community and is actively being developed [2, p. 3]. In short, Jenkins schedules and monitors the execution jobs, which can run either on the Jenkins master or on one of the Jenkins slave nodes that can be connected to the master. A job can theoretically be anything, but is usually related to building software projects and executing automated tests on the built product. Therefore, Jenkins plays a very central role in the development process, and an interruption in the availability of Jenkins may be very costly. It is designed to be installed on one server only, which means that a failure in that server will cause Jenkins to fail, making the service Jenkins provides unavailable.

There are many different techniques available for application for which an outage just cannot be afforded. They rely on automatic detection of failure in the application and will upon failure take action to restore the availability of the service that the application provided. The most basic approach to achieve high availability is to have a redundant set of servers, where the application will be executing on one of them. Should the application or the server hosting it fail, the application will be started on one of the other available servers. This operation is referred to as a failover. While the application is starting, the service that it provides will be unavailable. To remove even this short time of unavailability, techniques exist for having multiple instances of an application active that cooperate and share the load. A failover in such a system is much faster, as the backup servers already have the application in a running state, ready to accept the tasks of a failing node [3, p. 1]. Such a solution imposes several requirements on the application, and cannot easily be applied to any application.
1.2 Problem

Jenkins is today a single point of failure, which means that failure in Jenkins, or in the server on which Jenkins is installed, will make the service it provides unavailable.

Today no free solution to achieve high availability exists. One commercial solution is available however, which works by having one active Jenkins and one or more in standby. A standby Jenkins will not be started unless the primary Jenkins fails. Not only does this solution lead to a resource waste, but also while the backup Jenkins is starting up, Jenkins will remain unavailable. Therefore, a solution where multiple Jenkins masters can cooperate and share the load is preferred.

Although Jenkins is an open source project, the source code of Jenkins must not be modified. Any change would need to be submitted and accepted by the Jenkins community, or it must be reapplied any time Jenkins is updated to a new version. It was decided that there was a too large risk for the investigation to be delayed, if it was necessary to wait for any change to be reviewed and accepted. In addition, a lot of extra testing would have been required to make sure that any submitted change would not introduce any new bugs in Jenkins. Therefore, a solution has to be truly non-intrusive, where modifications to Jenkins are avoided.

1.2.1 Problem Statement

Is it possible to extend Jenkins without modifying its source code, so that multiple Jenkins masters can cooperate and share the load, and in case of failure in one Jenkins step in and take over the responsibilities and load of the failed Jenkins? Which of the available techniques for high availability are applicable to Jenkins?

1.3 Limitations

This thesis was carried out with the restriction that the source code of Jenkins may not be modified, and some issues are not possible to solve without modifying the source code of Jenkins. Exactly what modifications that might be necessary will not be investigated.

Some features of Jenkins were not considered in this thesis. A Jenkins masters can be connected to a group of Jenkins slave nodes. It was not investigated how several Jenkins masters can share the same Jenkins slaves between each other.

Finally, there is an issue known referred to as the split-brain problem. If a network connection stops working, a Jenkins cluster may be split into two separate groups. In this situation there is no way for one Jenkins to know if the Jenkinsees that disappeared have crashed or if they are in fact okay but there is a problem with the network connection. For this investigation, this problem has not been accounted for.
1.4 Research approach

Along with the investigation a prototype was developed. The prototype was used to make sure that theories were actually usable and could be implemented. Possibly more important, the prototype gave a good insight in what was not possible to do.

The prototype was developed as a plugin for Jenkins, which relies on JGroups \cite{4} for the underlying group membership management. JGroups is a toolkit for reliable messaging and can be used to create clusters whose nodes can send messages to each other. By using JGroups more energy could be put on the issues of a distributed Jenkins, rather than group membership management and message passing.

When the plugin was tested, a setup with two Jenkins masters was used, where each Jenkins was accessed directly with a unique URL. Usually there is a front end that all traffic will be routed via, which in turn directs the traffic to one of the Jenkins masters. It was decided that such a front end was unnecessary in this investigation, mainly because it would have very little, if any, effect on the results.

1.5 Outline

In chapter 2, different techniques that can be used to create highly available systems will be discussed. This is followed by chapter 3 where issues of asynchronous replication and possible solutions are presented. In chapter 4, Jenkins is introduced which is followed by a more detailed description about Jenkins plugins. In chapter 5, an already existing solution for high availability in Jenkins is presented. In chapter 6, it is discussed which of the techniques that can be used for high availability that can be used for Jenkins. Finally, in chapter 7, the report is concluded and the major findings are presented ending with possibilities for future work.
2 High Availability

High availability refers to a system design that minimizes the downtime and maximizes the time the system is available to its users [5]. To accomplish this, systems need to be prepared for the event of failure so that the time it takes to get it working again is minimized. The strategy is to add redundancy to the system so that a service automatically can be made available via a backup system if the primary system should fail. A group of servers are connected and form a high availability cluster (HA cluster), where servers can back each other up in case of failure. A cluster is in many ways presented as one single entity to the users, and a failover is in the best case completely unnoticeable for users.

2.1 High Availability Cluster Configurations

A HA clusters can be designed in many different ways, which can be divided into two major categories active/passive and active/active. In the active/passive system, only one instance of an application can be running on one server at a time. In an active/active system there may be multiple servers that are running the same application at the same time.

In any configuration, there will be some sort of health check of each active application to verify that it is responsive. A common technique for this is the heartbeat mechanism. Every node periodically exchanges heartbeat messages with each other. If suddenly a node stops sending heartbeat messages, it is good an indication that it may have failed.

2.1.1 Active/Passive

In active/passive clusters, there is a server in standby ready to step in and take over if another server fails. The standby server must not run any critical applications but only applications that can be terminated as soon as a failover is about to happen.

In the case of failure on one server, its applications will be terminated and started on a backup server instead. If it is just one application and not an entire server that became unavailable, the application can first be restarted, and if that is not helping, it can be started on a backup server instead. While an application is starting up, on a new server or not, the service it provides will be unavailable. How long time it takes to start an application is obviously varying depending on the application, but a typical delay is a few minutes [6, p. 6]. If the application uses an external database to store its data as shown in Figure 2-1, an application that is moved to a backup server can continue to use this database. Another possibility is that the application stores its data as files on the local disk of the server. In this case, it might be necessary to copy those files to the backup server before the application can be re-started there. This will increase the time it takes to complete a failover.
2.1.2 Multi Instance

This solution has a lot in common with an active/passive solution, but now the backup server may run another critical application, instead of being idling. This solution does not leave resources unused. If one server fails, its applications will be moved to a backup server where they have to co-exist with the applications that were already running on that server. The risk for an overloaded backup server must therefore be considered.

2.1.3 Parallel Database

In this configuration, multiple servers can be running the same application against the same database. This requires a special database that can run in a distributed manner on several servers. An example of such a server is Oracle’s Real Application Cluster (RAC) database [6, p. 4]. This configuration falls under the category active/active in the sense that there are two active instances of the same application in parallel. However, this is not the definition used for active/active in chapter 2.2 and the rest of the report.
2.2 Active/Active

For a system with very high requirements on availability, a good solution is the active/active solution since it provides a very fast failover procedure. Each instance of the application has a local copy of the common application database, and the copies are kept synchronized with each other via data replication. The users, and thus the load, will be spread over the available servers. Any change in one database will be replicated to the other applications database. Since every server will have a complete copy of the database, a failover can be completed in seconds. It may be enough just to redirect the users from the failed server to a surviving server which is a very quick operation. As in the multi instance configuration, the load imposed on surviving nodes after a failover must be considered. In Figure 2-2 a simple illustration showing an active/active system is shown.

![Diagram of Active/Active System](image)
2.2.1 Data Replication

In active/active systems, changes to a resource in one database must be replicated to the other databases. There are two ways this change can be replicated. Either the source node, from which the change originates, applies the change to all databases at once, or only writes the change to the local database and lets an external service do the replication in the background. These techniques are called synchronous and asynchronous replication respectively.

**Synchronous Replication**

Synchronous replication only lets one node at a time change the same resource, and the change will be applied at all nodes “at the same time”. “At the same time” refers to the property that every node always will see the same version of a resource. To accomplish this, a read/write lock is acquired for each resource that will be modified and replicated, at every node. When an application has modified a resource, it cannot continue its execution until all nodes have received and applied the change. This extra delay compared to when a single database is used, is referred to as the application delay. It also affects any node that wants to read a resource that is being updated in the sense that it has to wait for the resource lock to be released before it can be read.

As always, locks may result in deadlocks. If two nodes want to acquire two locks for resource A and B simultaneously, and acquire the locks in different order, both will wait for the other one to release the remaining lock. Locks should therefore be acquired in the same order on both nodes if deadlocks are to be avoided.

The big advantage with synchronous replication is the impossibility of colliding changes which will be described in the following chapter Asynchronous Replication. The disadvantage is the imposed delay and performance penalty the application will experience.

**Asynchronous Replication**

Asynchronous replication, as opposed to synchronous replication, does not interfere with the application, and can be performed silently in the background. A replication engine at each node monitors the database for changes, which are sent to the remote replication engine where they are applied.

An asynchronous replication engine can be described as three components: an extractor, a communication channel and an applier. The extractor listens for changes in the source database, which are sent over the communication channel to the replication engine at the target nodes. At the receiving ends, the change is applied to the database by the applier.

The big advantage with asynchronous replication is that there is no performance penalty as with synchronous replication. The application can save changes to the local database, and continue its execution right away, as if there was only one instance of
the application. No locks are acquired before a resource is changed or replicated, which means that there is no risk for distributed deadlocks.

Another good property is that the extractor and the applier are loosely coupled. The applier can work with a very different type of database than the extractor is observing. If the data needs to be modified to fit the target database, e.g. unit conversion, the applier has the possibility to do so.

In spite of the many advantages of asynchronous replication, several issues need to be considered when designing an asynchronously replicated system. This is covered in the following chapter Issues with Asynchronous Replication.
3 Issues with Asynchronous Replication

When deciding whether asynchronous replication is the appropriate solution, several questions need to be answered. Is it okay to lose data if a node fails? What should happen if two nodes modify the same resource simultaneously?

3.1 Replication to several nodes

If replication is done to more than one node, a reliable broadcast protocol must be used. Reliable broadcast ensures that if any node receives a message, all other nodes that have not failed will receive it. Why is this important? Consider this simple example.

Three nodes, A, B and C, are connected as an active/active system. Node A creates a new resource and begins to replicate it. The resource is successfully replicated to B, but before A had the time to replicate it to C, A crashes. As a result the resource will be created on B but not C. A reliable broadcasting protocol ensures that if node B receives the instruction to create the resource, also C will receive this instruction. In the case when replication is only performed to one other node, this issue does not exist. The resource is either replicated to all target nodes or none.

3.2 Data Loss

When a node fails, it is possible that there are changes that have been performed locally that have not been replicated yet. Usually changes are buffered in some sort of queue before they get transferred to the target nodes. When a server fails, there may be changes in the queue waiting to get replicated, which will be lost. If the queue exists as file on the disk, the changes may be replicated when the server is brought back online, but until then the changes will not be accessible by any other node. If data loss cannot be tolerated, synchronous replication has to be used.

3.3 Data Collisions

In asynchronously replicated systems, there is a delay from when a change is performed until it has been replicated and applied to the target databases. This is known as the replication delay. Should an application modify a resource for which another change is about to be replicated by another node, a data collision has occurred. The risk for data collisions is directly related to the replication delay. The longer time it takes for a change to be applied to other databases, the greater the risk that the resource will have been modified by another node before the replication has completed.

When a data collision occurs, each node involved may end up with different versions of the same resource, which all might be incorrect. To solve the problem of data collisions there are three approaches: to prevent collisions from happening, to
detect the collisions and resolve them, or to represent a change as a relative difference.

### 3.3.1 Replication of Relative changes

If changes are represented as a relative change instead of an absolute value, simultaneous changes would not overwrite each other, but in combination result in a third and correct value. Consider an active/active application executing on two nodes which both have an integer with a value of five in their databases. If both simultaneously increment this value by one they will both have a new value, six. If absolute replication would be used, both applications would replicate the new value six, which would be the final value. However, since the value was incremented two times, one time on each node, the final value should be seven. With relative replication, the applications would not replicate the new value six, but instead the relative difference, which is one. This way the applications would first do the local addition of one, and when they receive the replication message from the other node they would increment the value by one again. Both nodes would now have the correct value of seven.

This technique can be very useful, but cannot be applied to every kind of data. Text files for instance, cannot always be replicated this way. Also this technique introduces a risk. If a resource is corrupted for any reason, it will continue to be corrupted no matter how many relative changes that are received. If however absolute replication would be used, any corrupted resource would get reset to a hopefully correct value when a change is received from another node.

### 3.3.2 Collision Avoidance

If representing changes as relative differences is not possible, a good way of handling collisions is to not let them happen. Below are three ways to avoid data collisions.

**Database Partitioning**

The database can be split in sub-partitions, which are said to be “owned” by a certain node. Only the owner of a partition has the right to modify the data in it. Other nodes who want to modify a resource in this partition would have to pass a modification request to the owner of the partition. Consequently, each resource may be modified by one node only, and collisions would thus be avoided. This technique can be very difficult to implement since it is not always obvious how data should be divided among the nodes. In addition, there must be a mechanism to move the ownership of partitions if a node would fail, as well as when a new node is connected to the cluster.
**Master Node**

One node is selected as a master node which is the only one who has the right to modify data. All changes would have to be forwarded to the master node which applies the change and then replicates the updated value to the other databases. If the master node would fail, there must be a mechanism to select a new master node. This approach can be seen as a special version of database partitioning, in which all partitions belong to one single node.

**Synchronous Replication**

Synchronous replication avoids collisions by acquiring read/write locks for each resource that is going to be modified. As a result only one node at a time can modify a resource and no risk for data collisions exists. For a more complete description of synchronous replication, see chapter 2.2.1.

3.3.3 Collision Detection

In systems where data collisions cannot be avoided, it is usually necessary to detect and resolve them. This chapter presents three ways to detect data collisions.

**Resource Versioning**

Each resource has version number that is incremented every time the resource is modified. In the stable state, every node will have the same version number for the same resource. Obviously, the resources need to be equal as well. Whenever a resource is changed, its version number is increased.

When a change is replicated both the pre-update version number and the current version number is attached to the message. By knowing which version that a received change was based on, the receiver can check if the change was based on the same version of the resource as the local current version. If the received pre-update version number matches the local current version of the resource, the change is safe to apply. This is shown in Figure 3-1. However, if both nodes modified the same resource simultaneously, both will have incremented the local version number. This means that the received pre-update version number will not match the local current version. This scenario is shown in Figure 3-2.
Figure 3-1 A sequence diagram illustrating a situation where no data collision occurs.

Figure 3-2 A sequence diagram illustrating a situation where a data collision occurs.

**Attaching Pre-update Copy**

When a change for a resource is replicated, a copy of the resource from before the update is attached to the message. When the message is received, the pre-update version of the resource is compared with the local version of the resource. In the normal case, the received pre-update version should match exactly with the local resource. However if both nodes modified the same resource simultaneously, the received pre-update resource will not match the current local version of the resource, which means that a data collision has occurred. This method of detecting collisions is
based on the same principle as the one that uses version numbers, but instead of comparing version numbers, the entire resources are compared.

**Periodic Database Comparison**
In this approach, collisions are not detected right away. Instead, they will be detected by a periodic comparison of the two databases. If it is detected that a resource in one database does not match the same resource in the other database it means that there has been a data collision.

### 3.3.4 Collision Resolution
When a collision has been detected, there are several ways to handle it. Each node needs to decide which version of the resource to keep, and which to discard. Most important of all is that the nodes reach the same decision. Below are three ways of handling a data collision.

**Node precedence**
Each node is given a unique rank, which can be just a number. When a data collision occurs, the change that was received from the node with the highest rank will be the chosen one.

**Time stamps**
It is not uncommon that resources have an extra property with the time they were last modified on. By comparing the timestamps of the two different received changes, the nodes can choose the change that has to most recent time stamp or vice versa.

**Ignore it**
In some cases, an inconsistent state can be tolerated. If for instance the resource is updated frequently, it might be okay to ignore a data collision, as the inconsistent state will be overwritten by any consecutive non-colliding update.
3.4 Referential Integrity

It is important that changes are applied at the target node in the same order as they were observed at the source node. For example, an update of a resource cannot be applied before the creation of it, and a replication engine should never let this happen.

If the replication engine is multi-threaded where several independent threads with their own stream of changes are used, it is important that the receiver applies the received changes in the correct order.

Single threaded replication engines does not suffer from this issue as long as messages sent over the communication channel are received and delivered in the same order as they were received. If TCP is used as the underlying transportation protocol it is guaranteed that data will be received in the same order as it was sent. If UDP is used however, messages might not arrive in the order they were sent and the replication engine must therefore have a solution to handle this.

3.5 Ping Ponging or Data Looping

When asynchronous replication is used it can happen that two nodes are replicating the same change back and forth between each other forever. The reason for this is as follows.

Consider a replication engine that monitors a database. Whenever a change is made to the database, it will be replicated to a target database. When the change is applied to the target database, its replication engine will as instructed react to the change and replicate it back to the source node. At the source node, the replication engine receives the message, applies the change once again, and the looping has begun.

To prevent changes from looping the replication engine must be able to determine the source of a change, and replicate only those changes that were made by the local application.

There are several ways data looping can be prevented including the following:

3.5.1 Database Partitioning

One way to avoid the problem of looping replications is to partition the database. Each partition and its contained resources are owned by one single node, which means that it can only be modified by that node. If the replication engine is configured to only replicate changes to resources that it is the owner of, no data looping would occur. Changes to resources owned by other nodes, cannot possibly originate from the local node, and should therefore not be replicated.
3.5.2 Tracking the Source

If the replication engine can check which node that was the last one to modify each resource, it can be configured to only replicate changes where the local node is set as the last modifier of the resource.

3.5.3 Control Table

If the replication engine has no possibility to determine the source node of a change, it can track received replicated changes with a control table. When a change is received, the applier enters a transaction id for that change in the control table and applies the change. When the extractor observers the applied change, it reads the transaction id for the observed operation and looks for that id in the control table. If this id is found in the control table, the extractor should not replicate the change as it was just received as a replicated change.

A similar approach would be to store a copy of the received replicated resources in the control table. When an extractor observes a change to a resource, the change is only replicated if it the new version of the resource is different from the version that is stored in the control table. This way only changes made by the local application would get replicated since they will not be found in the control table.
4 Jenkins

In this chapter, the CI tool Jenkins will be presented. For a complete guide on how to use Jenkins, the book “The definitive guide to Jenkins” [2] is highly recommended.

4.1 A Short Introduction to Jenkins

Jenkins in short is an open source application for running automated tasks and schedules and monitors the execution of repeated jobs such as building the latest version of an application or running automated tests. It is primarily managed via a graphical web interface, but there is also a command line interface known as Jenkins CLI [9] and a HTTP API [10] to control Jenkins.

The heart of Jenkins is the build jobs, which can be thought of as a task to be performed in a build process. A job can however do a great variety of things, including generating documentation, deploying a successfully built application to a web server or to measure code quality metrics [2, p. 21]. Each execution of a job is referred to as a build, and is kept in a build history, often together with the built product(s) known as the artifact(s).

When building a software project, Jenkins begins with checking out the source code, normally from a version control system such as Subversion [11] or Git [12]. The term source code management (SCM) will hereafter be used as a term for any system that can be used by Jenkins to retrieve the source code for a project.

There are several ways a job can be started, the simplest way being clicking the “build”-button in the web interface. However, some automation is usually desired which is why other build triggers are available. A job can be configured to be run periodically, e.g. every other hour, or to be started once another job has complete, or when changes in the SCM have been detected.

A key strength of Jenkins is the rich flora of open source plugins available to extend its functionalities.
4.2 The Internal Jenkins Structure

Jenkins consists of many parts and it would be too much to describe all of them. This chapter will describe those parts that were identified as relevant to this investigation.

4.2.1 Persistent Storage

Jenkins stores all important data in a directory called the Jenkins home directory. In this directory, Jenkins stores build job configurations, build artifacts, user accounts etc. Any file path will use this home directory as the root directory.

Jenkins uses XML files for storing configurations of Jenkins itself and for jobs, builds and plugins etc. When starting up, Jenkins reads the XML files and uses them to initialize Jenkins’ internal state. When a change is performed in Jenkins, such as an updating job configuration, the object representing that job is first updated to the new version. Then the object is turned into an XML file that is stored in the home directory. Note that changes made on files do not take effect until Jenkins is either restarted or its configuration is reloaded.

Jobs

In Jenkins, jobs saved in the directory /jobs/<jobName>/. Inside this there are two files and one other directory. One file is config.xml which contains the configurations for the job, and the other is nextBuildNumber that contains a number representing the id to be used for next build of this job. Anytime a build job is started the number is incremented by one.

Finally there is a directory labeled builds where the files for all completed builds for this job are stored.

Builds

In the directory /jobs/<jobName>/builds the files for all completed builds are stored. There will be no files for a build that is still executing. Each build will be stored in its own directory in which at least three will be found: build.xml, changelog.xml and log. The file builds.xml contains information about the build such as start time, end time and if the build was successful or not. The file changelog.xml contains the changes in the source code for this build, and log contains the log output from the execution of the build job.

In Linux systems there are also symbolic links in the builds –directory. For each build directory there will also be a symbolic link pointing to it. The symbolic link has the build number of the build it is pointing to as name. There will also be some other symbolic links, lastStableBuild, lastUnstableBuild, lastFailedBuild etc. Which build folder they point to should be self-explanatory.
4.2.2 Executors and Slaves

To manage the load of Jenkins, there is a limit on how many jobs that can be running simultaneously. Jenkins has a number of Executors, which can execute one and only one job each. As long as a build is ongoing, the executor will be busy. If a high number of jobs need to be running at the same time it is possible to add other computers as Jenkins slaves to increase the number of executors. However, slaves do not perform any scheduling of jobs. A slave is assigned jobs by the Jenkins master.

If a build is triggered and all executors are busy on the master and on the slaves, the build will stay in a queue while waiting for an executor to become free.

4.2.3 The Queue

Anytime a job start is triggered, the job will be placed in the build queue. Jobs will stay in this queue until the job either is cancelled or is taken for execution. This part of Jenkins is one of the more difficult parts to adapt to an active/active system. There can only be one entry for each job in the queue and before the job is added to the queue, a check is made to verify that it is okay to queue the job. Here plugins have the right to contribute to the decision and can prevent a job from being queued at all. This is accomplished with an extension point called ScheduleDecisionHandler.

When a job has been placed in the queue it will pass through several stages before it is actually picked from the queue to be executed which is shown in Figure 4-1. A job can be cancelled at any stage in the queue.

![Figure 4-1 The possible stages for jobs in the queue](image)

In the first stage, called waiting list, jobs will stay for a short period of time, called the quite time. After this delay, the job will move on to the next stage, which is either blocked or buildable. Jobs in the buildable stage can be taken for execution immediately. Jobs in the blocked state are jobs that could be executed immediately but is blocked for some reason. Possible reasons are that another build of the same job is executing or because a plugin that has implemented the extension point QueueTaskDispatcher is denying an execution for this job.
4.3 Jenkins Plugins

Jenkins is easily extended with new functionalities by using plugins. There are a large number of plugins available where most of them are open source projects. Plugins can modify almost any aspect on Jenkins, thanks to the great set of extension points.

4.3.1 Extension Points

The term extension point refers places in the Jenkins source code where plugins can insert their own functions and behaviors. Some of them can be used to change the visual appearance, some can be used to perform periodical tasks in the background, and some will extend jobs with new capabilities. A complete list of the available extension points can be found on the webpage for Jenkins [13].

In Jenkins, an extension point is defined as a class with one or more methods defined. A plugin developer that wishes to use an extension point will create a class that extends this extension point and overrides the desired methods. The methods will be invoked by Jenkins to notify the plugin of certain events. A good example is the extension point `ItemListner` which among others has the method `onCreated(Item item)`. Any time a job is created, all extensions of the type `ItemListner` will be looped over, and the method `onCreated(...)` will be invoked on each one of them. Below in Listing 4-1 is a code snippet from `ItemListner.class` which shows how `onCreated(...)` is called.

```java
public static ExtensionList<ItemListener> all() {
    return Jenkins.getInstance().getExtensionList(ItemListener.class);
}

public static void fireOnCreated(Item item) {
    for (ItemListener l : all())
        l.onCreated(item);
}
```

Listing 4-1 A code snippet from `ItemListner.class` showing how `onCreated(...)` is invoked on all defined extensions that is an `ItemListner`. The method `all()` returns every extension of the type `ItemListner`.
4.3.2 Some of the Available Extension Points

There are a great number of extension points available and plugins can even create their own extension points. In this chapter, some of the extension points that are relevant to this thesis are presented. For each extension point, there will be a short description of what it is used for, followed by a list of the methods that Jenkins invokes to signal different events or when plugins can contribute to decisions.

RunListener

An execution of a build job is internally referred to as a run. A RunListener has several methods, which are called to signal events related to these runs. These are the methods:

- `onCompleted(Run r, TaskListener listener)`
  This method is invoked once a run has completed.

- `onFinalized(Run r)`
  This method is invoked once a run as completed and the related files have been saved in the home directory.

- `onStarted(Run r, TaskListener listener)`
  This method is invoked when a job has been picked from the queue and has started its execution.

- `setUpEnvironment(…)`  
  This method is invoked before source code is fetched from an SCM. It allows a plugin to contribute with additional properties or variables to the environment.

- `onDelete(Run r)`
  This method is invoked when a run has been deleted from the build history.

ItemListener

Any time job is created, renamed, updated or deleted there is a corresponding method in ItemListener that is invoked. These are the methods:

- `onCreated(Item item)`
  This method is invoked when a job is created. The argument item contains the job that was created.

- `onCopied(Item src, Item item)`
  This method is invoked when a job is created as a copy of another job. The argument src is the job that was used as a base for the new job that is provided as the argument item.
onDeleted(Item item)
This method is invoked when a job has been deleted. The argument in this method contains the deleted job.

onRenamed(Item item, String oldName, String newName)
This method is invoked when a job has been renamed. What the arguments are used for should be self-explanatory.

SaveableListener
Every object that can be serialized and persisted to disk will implement the interface Saveable. To get notified when an object is saved to disk, the extension point SaveableListener can be used. This is the method:

onChange(Saveable o, XmlFile file)
This method is invoked when the object has been serialized and saved in the home directory. The first parameter represents the object that was serialized and the second the produced XML file.

QueueDecisionHandler
This extension point can be used by plugins to decide if a job can be put in the queue or not.

boolean shouldSchedule(Queue.Task p, List<Action> actions)
This method is invoked by Jenkins when a job is about to be placed in the queue. The return value is a boolean which if true means that the job can be put in the queue, according to the specific QueueDecisionHandler that was called. If one of the QueueDecisionHandlers would return false, the job will not be placed in the queue.

QueueTaskDispatcher
This extension point is used by Jenkins to decide whether a queued job can be taken for executed or not. The methods in this extension point return null or an object that extends CauseOfBlockage which as the name suggests contains a reason for why a job could not be executed. If a non-null value is returned, the job in question will get a blocked state in the queue.

These methods are invoked before a job is picked from the queue for execution, and during queue maintenance when jobs internally are moved between the stages in the queue.
**CauseOfBlockage canRun(Queue.Item item)**

This method is used to test if a job is allowed to be executed for the moment. An example of a return value is an instance of `BecauseOfBuildInProgress`, which signals that the job cannot be picked from the queue because another builds for this job is currently executing.

**CauseOfBlockage canTake(Node node, BuildableItem item)**

This method is used by Jenkins to test if a job can be executed on a certain slave or not. An example of a return value is an instance of `BecauseNodeIsOffline`, which as the name suggest means that the job cannot be executed on that slave because it is not online. In case the job must not be executed at all, regardless of which slave, the method `canRun(...)` should be used instead.
5 Related Work

The use of Jenkins is widely spread and the desire for a highly available Jenkins master has been expressed several times in the mailing list for Jenkins here [14], here [15] and here [16]. The solution that is recommended is an active/passive design.

There is one existing solution to increase the availability of Jenkins which is a plugin developed by CloudBees which unfortunately is not free or open source.

5.1 CloudBees’ High Availability Plugin

For the pure version of Jenkins there is no high availability solution yet, but there is a non-free version of Jenkins called Jenkins Enterprise [17] [18] which offers a high availability solution through their high availability plugin [19]. Jenkins Enterprise is maintained by CloudBees where among others the creator of Jenkins, Kohsuke Kawaguchi, is employed [20].

This high availability solution is of the active/passive kind of solution. This means that there may be several Jenkins masters but only one will be active. The others will be in standby ready to be started in case the active instance should fail.

Each Jenkins instance will be configured to use the same Jenkins home directory, which is fine as long as it is guaranteed that only one Jenkins at a time will be active. In case the active Jenkins fails, one of the backup Jenkinses will be started up. During the time the backup Jenkins is starting, it will not be able to serve inbound requests or builds and the Jenkins will be unavailable. The startup time for Jenkins is typically a few of minutes, but will be increased if there are many builds with large build histories.

Since the backup Jenkins will use the same home directory, global configurations, user accounts, completed builds and job configurations will survive a failover. Any build that was in progress on the first Jenkins will however be lost and there will be no attempt of restarting them. Also, users that were signed in will need to re-authenticate after a failover.

Behind the scenes, JGroups is used to manage the cluster and to select which Jenkins that should be the active one.
6 High Availability Applied to Jenkins

This section will go through the presented techniques than can be used when designing a highly available system. With the restriction that the source code of Jenkins must not be modified, many of the techniques will be proved unusable.

First, the possible configurations of a high availability cluster will be compared to each other. This is followed by a discussion about how changes to jobs and builds should be detected and replicated. Finally, the issue with simultaneous job starts will be discussed.

6.1 Different Cluster Configurations

As presented in chapter 2.1, HA clusters can be set up in many ways. The requirement for this thesis was a solution where two or more Jenkins masters can run in parallel while cooperating and sharing the load. The only two cluster configurations that comply with this requirement are parallel database and active/active. The other configurations meant that there would be one passive node. For reasons that will be explained below, parallel database solutions was rejected, which left active/active as the only possible solution.

Parallel database applied to Jenkins means that every Jenkins master will be configured to use the same home directory. Since each Jenkins is unaware of that other Jenkinses is using the same home directory they will happily modify the files behind the backs of each other. Not only would the job configurations be affected but basically any configurations and files that are modified at runtime by Jenkins. The correct behavior is to update both the files and Jenkins internal state. To cover up for every possible scenario is simply not a good and reliable solution.

By using active/active, configuration modifications that are made by one Jenkins would still need to be replicated. But in this case the replication is controlled and only performed on command. Issues and side effects from replication is therefore much more easy to spot and handle.

6.2 Advantages of a Plugin

Since active/active was the only viable solution, there will be two or more databases copies that need to be synchronized with each other.

Low-level approaches for data replication were discussed such as listening to changes in the file system and replicate them. By listening to file changes in one Jenkins’ home directory, new job configurations that were saved to config.xml can be applied to other Jenkinses by using the HTTP API or Jenkins CLI. An active/active solution needs access to the internal state of Jenkins, which cannot be accomplished with these two methods. With a plugin it is easy to view internal data structures, and
by the use of extension points even the behavior of Jenkins can be modified. Therefore a plugin is the more powerful and flexible solution.

### 6.3 Limitations with a Plugin

Although a Plugin can read almost any part of the internal state of Jenkins, several things cannot be done due to security restrictions. Data structures may be private and only accessible via read-methods (getters). There are also special annotations that are used to further restrict the abilities to modify some data structures or call methods.

When a resource has been updated in Jenkins, plugins can, as described, be notified of such events by using extensions points. Unfortunately, there is no way for a plugin to know in advance that a resource is going to be modified.

### 6.4 Backward Compatibility for Existing Plugins

One severe problem with an active/active solution based on a plugin is the backward compatibility for other plugins that may be installed on the Jenkinses. It is not possible to make every plugin aware of that several Jenkins masters are cooperating. Decisions of plugins will therefore be made based upon local information only, which may have undesired consequences.

A solution to this problem would most probably require modifications both to the source code of Jenkins, and to many of the existing plugins. Exactly which modifications that are required to solve this issue, will not be investigated any further.

### 6.5 Replicating jobs and builds

In an active/active solution, the end user should not have to be aware of which of the Jenkins masters they are using. Any change made on one Jenkins should be applied to all Jenkins masters with replication. When replicating a change it is necessary to do it in a way so that not only the internal state of the target Jenkinses is updated, but also any related file that is used by Jenkins to persist configurations.

Several techniques discussed in this chapter require that job configurations can be compared to each other. Unfortunately, the class Job.class does not implement the method equals and comparing objects with that method is therefore useless. One way to compare two jobs is to compare their XML representations. However care must be taken since XML strings may be different depending on the version of Jenkins and to operating system on which is installed on. For example, the character for line endings in Windows and Linux differ. In addition, if the order of the properties differs between the XML strings, a simple comparison would consider them different, even though the contained properties are identical. To not falsely believe that two jobs are different even if they are not, the tool XMLUnit [21] or similar can be used. XMLUnit can be configured to not look at the formatting of the XML string, but only at the contained properties.
As presented earlier, there are two kinds of replication: synchronous and asynchronous replication.

6.5.1 Synchronous Replication
In synchronous replication, a read/write lock would need to be acquired for each resource that is going to be modified, at every Jenkins master. By the design of Jenkins, there is no way a plugin can be notified that a resource such as a job is going to be modified before it happens. Only after a change has been applied, plugins are notified. This makes the use of synchronous replication impossible since there is no point in the code where locks can be acquired. As a result, asynchronous replication is the only option left. This means that issues such as data collisions must be considered and handled.

6.5.2 Asynchronous Replication
The asynchronous replication and the replication engine can be implemented as a part of a plugin. How the replication is performed will be presented in the following chapter The Asynchronous Replication Engine.

6.6 The Asynchronous Replication Engine
An asynchronous replication engine can be described as the three parts extractor, communication channel, and applier. How the extractor and applier handle observed changes will vary depending on the kind of operation that is to be replicated and which type of resource in question. A job creation is different from a modification of an existing job and has to be treated accordingly, both by the extractor and by the applier.

The following two chapters will, in more detail, explain how the extractor, applier and communication channel can or cannot be designed.

6.6.1 The Communication Channel
Plugins must be able to communicate and share information such as new job configuration. In the plugin JGroups was used to communicate between the two Jenkinses. A remote procedure call (RPC) abstraction was used to notify the plugin on the target Jenkins about changes. Note that since changes were replicated to only one other Jenkins, reliable broadcast was not used. However, there are protocols included in JGroups which enable total order multicasting, which could be useful if many Jenkinses are connected in a cluster.

6.6.2 The Extractor
The extractor is the part of the replication engine that observes and listens for changes. When a change is detected, it is sent over the communication channel to the remote Jenkins where the change is applied.

In Jenkins, the best way to detect changes is to develop a plugin and use the available extension points. This way the plugin can be automatically notified whenever
a change to a resource has occurred. What happens next depends on the type of resource in question and which kind of operation that was performed.

**Jobs**

When detecting changes to jobs, the extension point `ItemListener` is very useful. It is easy to distinguish the kinds of operations from each other as all result in a different method being invoked.

When the plugin is notified of a job creation or a job configuration update, the methods `onCreated(Item item)` or `onUpdated(Item item)` will be invoked. These events both mean that there is a new job configuration that needs to be sent to the target Jenkins. The argument `item` that is provided in these methods is the object containing the job that should be replicated to the other Jenkinses. In order to send the job to the target Jenkins, the XML representation of the job must be acquired. One way to get it is to convert the object `item` to its XML representation in the same way as Jenkins does when it saves to objects to disk. How that is done is shown in Listing 6-1. Another way is to read the configuration file from Jenkins’ home directory as done in Listing 6-2.

```java
@Override
public void onUpdated(Item item) {
    String xml="<?xml version='1.0' encoding='UTF-8'?>
    "
    xml += new XStream2().toXML(item);
    //Send xml to the target Jenkins
}
```

Listing 6-1 Example of how an object can be serialized into XML that can be used to create a clone of the job

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

Listing 6-2 Example of how to get the XML representation of a job, from the XML file in the home directory

When it comes to job renaming and job deletion, no new job configurations need to be sent. When a job is going to be deleted, it is sufficient to send the name of the job that is going to be deleted, and when a job is going to be renamed, the old name and the new name is sent to the target Jenkins. The reason that only the names of the jobs need to be sent is that the name of a job is not stored in the job configuration file. The name of a job is instead determined from the name of the directory that the job is stored inside. In Listing 6-3 it is shown how the listeners for these two events can be used.
@Override
public void onRenamed(Item item, String oldName, String newName){
  //Send oldName and newName to the target Jenkins
}

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

Listing 6-3 Example on how to handle renamed and deleted jobs

A caveat
In the current version of Jenkins (1.502), onUpdated(...) is not always called when it should. There are shortcuts for disabling a job and to edit its description. When any of those ways to update a job are used, onUpdated(...) will not be invoked by Jenkins, even though a new configuration is saved. Because of this behavior it is necessary to use the extension point SaveableListener if all possible modifications of a job are to be caught. If SaveableListener is used, any modification to any configuration will trigger this listener, and jobs modifications must therefore be filtered out which could be done as shown in Listing 6-4. Also a SaveableListener is not able to distinguish different types of operations such as job creation or job modification from each other.

@Extension
public static class MySaveableListner extends SaveableListener {

  @Override
  public void onChange(Saveable o, XmlFile file) {

    if (o instanceof Job<?, ?>) {
      String jobName = ((Job<?, ?>) o).getName();
      try {
        String xml = file.asString();
        //Update the target Jenkins with xml
      }
    }
  }

Listing 6-4 Example of how to filter out jobs when using the SaveableListener

The use of the SaveableListener does unfortunately mean that an updated job configuration will be reported both in ItemListener and in the SaveableListener and the same event will get replicated twice. Ignoring this issue will not cause any problems as job operations will result in the same state regardless of the number of times they are
applied. There is one special case though which has to be handled. When a job is created, the `SaveableListener` will be called, before `onCreated` in the `ItemListener`, and send a modification instruction to the target Jenkins. If the applier tries to modify the job that is not created yet, an exception will be thrown, which is why an applier always should check if a job exists before trying to modify it. If the applier receives an instruction to modify a job that does not exist locally, it should ignore the instruction since it most likely will be followed by a create instruction with the same configurations.

**Builds**

The events related to the execution of jobs can be observed by using the extension point `RunListener`. The plugin can with `RunListener` be notified when a build have started, completed, finalized and been deleted. The replication of builds is quite different from jobs, and there are several files that need to be replicated, not just one XML string. The biggest difference is seen in the applier at the receiving end though, which will be discussed in the following chapter about the applier.

When `onStarted(…)` is invoked it signals that a build has started its execution. In the web interface of Jenkins it is listed and a progress bar shows the estimated time left for the build. This behavior is not possible to replicate since the data structure that holds the ongoing builds cannot be modified in any possible way by a plugin. One piece of data should, and can however be replicated. It is the next build number for the job, which will have been incremented by one when `onStarted(…)` is called. Every build should always have a unique build number, and it is therefore important that every Jenkins is notified whenever the build number for a job is incremented. Otherwise several builds with the same build number might be created.

How the new build number is acquired is shown in Listing 6-5.

```java
@override
public void onStarted(Run<?, ?> r, TaskListener listener) {
    int nextBuildNumber = r.getParent().getNextBuildNumber();
    String jobName = r.getParent().getName();
    //Update the next build number for the job
    //on the target Jenkins
}
```

Listing 6-5 Example on how to get the next build number for a job

When `onFinalize(Run<?, ?> r)` is called it is a signal that a build has completed and that the related files have been created in the home directory. To replicate this build, all of these files will have to be sent to the target Jenkins. These files cannot be acquired from method calls on provided `Run r`, but the files will have to be read
manually from disk. Luckily, the Run r has methods that can be used get the file path to the directory where the files are stored. Note that the file path will be absolute, and not with the home directory as the root directory. Therefore, a file path that is relative to the home directory should be extracted. The reason for this is that the path to the home directory might not be the same on the source Jenkins and the target Jenkins. An absolute path would probably not save the files in the right directory on the target Jenkins. An example of how the relative path can be extracted is shown in Listing 6-6. The string “jobs” is the name of the directory that contains all jobs and their build history, and is placed right in the root of the home directory. Therefore, the string “jobs” is used as the delimiter to separate the absolute path into two parts: the path to the home directory and the relative path to the directory that should be replicated.

Finally, be sure to not forget the symbolic links that are created on Linux systems for the builds.

```java
@Override
public void onFinalized(Run<?, ?> r) {
    String jobName = r.getParent().getName();
    String absPath = r.getRootDir().getAbsolutePath();
    String relativePath = absPath.split("(?=jobs)")[1];
    //Read all files and directories found at absPath
    //and send them together with relativePath to the
    //target Jenkins
}
```

Listing 6-6 Example on how to extract the path to the build file directory relative to the Jenkins home directory

### 6.6.3 The Applier

The applier is the receiver of changes that have been observed and transferred by an extractor on another Jenkins. It is the applier’s task to perform the same changes on its Jenkins.

An applier should try to mimic the behavior or the normal workflow of Jenkins as much as possible. If a job is modified on one Jenkins, it should be modified in a similar way on every other Jenkins, to which the change is replicated. Otherwise, the Jenkinses might get different states and configurations, which is not a desired scenario. For example a job modification means that several extensions with listeners for this event will be called. The applier should therefore apply changes in a way such that these listeners are called. Note that this will be a cause of data looping which will be discussed in chapter 6.9.

How modification are applied depends on which resource type that is affected. This chapter will first go through how operations on jobs should be applied, followed by how builds are handled.
**Applying Job Operations**

The applier will receive four different kinds of operations related to jobs: creation, modification, renaming and deletion. Jenkins provides a method for each one of these operations, which can be used by the applier, which are all used Jenkins internally.

Job creation is the easiest operation and the method call is as follows in Listing 6-7.

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

Listing 6-7 How to programmatically create jobs in Jenkins. The argument *name* is the name of the job, and *xml* is the XML data that describes the configuration of the job.

The remaining operations are operations on already existing jobs. Therefore the job in question must be acquired before any operation can be performed on it. In Jenkins a job is retrieved as shown in Listing 6-8.

```java
TopLevelItem topLevelItem = Jenkins.getInstance().getItem(jobName);

if (topLevelItem instanceof Job<?,?>) {
    Job<?,?> job = (Job<?,?>) topLevelItem;
    //The object job can now be used when the job is going to be renamed, updated or deleted
}
```

Listing 6-8 How to retrieve a job from a job name. The object *job* is necessary if a job is to be renamed, updated or deleted.

When an object has been retrieved for the job in question, it may be used in the three ways presented in Listing 6-9.

```java
//To rename the job
job.renameTo(String newName)

//To delete the job
job.delete()

//To update the job configurations. The xml document must be given as a Source.
job.updateByXml(Source source)

//An example on how to create StreamSource from a String
//by using the Apache IOUtils
Source source = (Source) new StreamSource(IOUtils.toInputStream(xml)));
```

Listing 6-9 How to rename, delete and update a job
Replicating Builds

Builds cannot be created or modified in the same way as jobs can. All ongoing and completed builds are stored in an object that can only be modified by the core of Jenkins. This leads to a couple of implications.

The impossibility to modify the object that contains the builds for a job, might at glance lead to the thought that builds may not be replicated at all. But there is a possible way to accomplish that. When a job is updated with an XML string with new configurations, Jenkins will go through the directory where the build files are stored for that job and add any builds that are found the build history. This behavior can be exploited to add received builds to the build history. When the applier receives the build files from another Jenkins, it can save these files to the directory /jobs/<jobName>/builds/. Then the applier updates the job with its own configurations. As a result, the build that only existed as files in the home directory will put in the build history and will be visible to Jenkins. Since the builds are added to Jenkins by reloading the job, the extension point RunListener will not be called. Hence there is no risk that the build replication can loop.

The final piece in the build replication is to update the next build number for a job when it has been incremented on one Jenkins. This is accomplished as shown in Listing 6-10.

```java
TopLevelItem topLevelItem = Jenkins.getInstance().getItem(jobName);
if (topLevelItem instanceof Job) {
    Job<?, ?> job = (Job<?, ?>) topLevelItem;
    try {
        job.updateNextBuildNumber(newNextBuildNumber);
    } catch (IOException ex) {
    }
}
```

Listing 6-10 How to update the next build number for a job

6.7 Data Collisions

If asynchronous replication is used for Jenkins, data collisions will be an issue that needs to be handled. Generally, the issue is solved either by avoiding data collisions or by allowing data collisions to happen and then resolve them. When it comes to Jenkins collision avoidance is not possible, and collisions must be detected and resolved.

6.7.1 Data Collision Avoidance

All presented solutions to avoid collisions would need some kind of modification to the source code of Jenkins.
**Database Partitioning and Master Jenkins**

These two techniques require the ability to intercept modifications and in some cases prevent them from happening. A plugin in Jenkins can however only be notified when a modification has already been performed, and then it is too late to abort it. These two techniques can therefore not be applied to Jenkins.

**Synchronous Replication**

The remaining solution for collision avoidance, synchronous replication, has been discussed in chapter 6.5.1, and it was concluded that it is not a possible solution, basically for the same reasons as above.

**6.7.2 Collision Detection**

As it is not possible to prevent collisions from happening, one needs to decide if they need to be resolved or if collisions can be tolerated. The techniques for collision detection that discussed in this chapter are resource-versioning, attached pre-updated copy, and periodic database comparison.

**Periodic Database Comparison**

With this solution, each Jenkins would periodically compare its resources and configurations with another Jenkins’ resources and resolve any found difference. Since this comparison is only performed with certain intervals, a data collision will not be resolved immediately when it occurs. Note that comparing resources is not always trivial. See chapter 6.5 at page 30 for more information.

**Attaching a Pre-Update Copy**

This solution means that anytime a resource has been updated and is replicated, the previous version of the configurations will be sent along with the new version. The receiver then checks if the received previous version matches the local configurations. If there is a mismatch, it means there has been a data collision.

This approach is the most disk space consuming one for Jenkins. When resources are modified and plugin is notified, only the latest version of the resource is given as an argument. This means that the plugin needs to maintain a database where a copy of each resource is kept. The copy would not be modified by Jenkins when a resource is updated, and can be used for the replication as the pre-update copy. When the replication has completed, the pre-update copy is replaced with the latest version of the resource.

The advantage with this approach over the periodic comparison is that data collisions are detected at a much earlier stage, as soon as a resource is replicated. The downside is that there need to be a copy of every resource that is being replicated, which will take up extra disk space and more bandwidth will be used for replication. Also as discussed in 6.5 at page 30, comparing resources is not always easy.
**Resource Versioning**

This solution is based on the same idea as an attached pre-update copy, but instead of sending an entire copy of a resource, a version number is sent. For every resource in Jenkins, the plugin will map a version number that is incremented anytime the resource is modified.

This solution is the preferred one, as version numbers are independent on the platform Jenkins is installed on, and storing a version number does not use up a lot of disk space.

### 6.8 Collision Resolution

For collision resolution, any of the presented techniques are applicable. The important thing is that each Jenkins reaches the same decision for which version of the resource to use, and which to discard.

Jenkins may choose the version with the most recent timestamp or one Jenkins could be set to have higher priority which will have precedence in case of a data collision. The third way, to ignore a data collision, can be used if an inconsistent state will not do any harm or have undesired consequences.

If more three or more Jenkinses are connected in a HA cluster, it is slightly more complicated than with two Jenkinses only and messages need to be reliably broadcasted and delivered in a way that preserve causal order. Consider this example. Jenkins A creates a job and broadcasts the change, which is first received by B. Before C gets A’s change, B modifies the created job and broadcast that change. Since B modified the job, after it received was received from A, Jenkins C should not see the change from B until A’s change is delivered and the contained job is created. This is referred to as preserving causality. Figure 6-1 shows an incorrect order of delivery, and Figure 6-2 the same scenario but with a correct order of delivery.

![Figure 6-1](image-url) An illustration of the issue with late messages when replication is done to more than one Jenkins. Jenkins A makes a modification that is replicated and first received by B. B makes a new modification that is replicated and received by C before A’s modification.
6.9 Data Looping

When a received replicated change such as a job creation is applied at the target Jenkins, any ItemListener will be notified of this event on the target Jenkins as well. It is important to make the plugin aware of that this notification should be ignored, as the change does not originate from the local Jenkins. Among the presented solutions to this problem, a control table is the one that may be used for Jenkins. The other solutions require modification to the source code of Jenkins. A more detailed version of why these techniques were not applicable to Jenkins is found below.

6.9.1 Database Partitioning

This technique, which is also discussed in chapter 3.3.2, requires that the plugin can intercept and prevent modifications if they are performed on a Jenkins that is not the owner of the partition to which the resources in question belong. This is not possible since plugins only can be notified of changes when they already have happened. If this technique could be used, the plugin would not replicate changes for resources that belong to partition that is owned by another Jenkins, as they obviously were not modified locally.

6.9.2 Tracking the Source of a Change

If there was a way that the plugin could check which Jenkins that originally created the new version of a resource, only changes that were performed by the local Jenkins should be replicated. Unfortunately, no way to accomplish this has been found. Any replicated resource would need to have an extra property with information about which Jenkins that made the last change to it. This property would need to be set by Jenkins before the plugin is notified, which would require modifications to Jenkins.

6.9.3 Control Table

This solution is the one that can be used for Jenkins. The plugin keeps records received replicated changes by storing them in a control table. When the received change is applied and the plugin is notified, the plugin checks in this control table to see if there is an entry for this change there. If there is, the change will not be
replicated. If there is no entry in the table for this change, it means that the change was originally performed locally and should be replicated.

Comparing objects such as jobs in Jenkins proved to be very difficult, see chapter 6.5, and a variant of the solution above was used in the prototype. Before a change is applied, the name of the job in question was stored in a list. When any listener is notified about this change, they will look in this list for this job name, and if it exists in the list, the change will not be replicated. After when the change has been applied, the applier removes the entry from this list.

6.10 Simultaneous Job Starts

When there are two independent Jenkins masters with identical job configurations it is possible for the same job to be running simultaneously on both which is an incorrect behavior. In the normal case when there is only one Jenkins master, there can usually only be one execution of a job ongoing at the same time. If a job is triggered when another build is already running, the job will stay in the queue until the other execution has finished. An active/active solution for Jenkins has to mimic this behavior and only let one Jenkins master at a time to execute a certain job. The extension points QueueTaskDispatcher and QueueDecisionHandler are the ones of interest when solving this issue.

6.10.1 A Naïve Approach

The most simple solution would be to check every Jenkins and make sure that they are not running the job that is about to be started. If no other Jenkins is running the job it can be started, otherwise the job has to be put in the queue, and retried later. This solution will unfortunately not work. If a build for a job is triggered simultaneously on two Jenkins masters, they will also simultaneously ask each other if they are running this job. Since neither of them is running the job (yet), both Jenkins masters will consider the job safe to start. This is a classic problem of concurrency that can be solved with locks and mutual exclusion.

If the source of Jenkins could be modified, this is how the problem of simultaneous job starts could be solved. Before the process of starting a job is initiated, a lock will be acquired. This lock will remain locked until the job either has been started or has been blocked. If during this time the plugin wants to check if this job is running it would have to wait until the lock is released. In the same manner, a job start cannot be initiated if another Jenkins is doing a check if this job is running or not. The check and the startup process for a job are said to be mutual exclusive. The addition of locks would make this approach possible to use, but does unfortunately require changes to the source code of Jenkins so that locks can be acquired and released at the right time. Therefore, it cannot be used.
6.10.2 A better solution

The issue of simultaneous job starts is solvable and an algorithm was developed, that did involve any modifications to Jenkins. However, it cannot be disclosed due to commercial reasons.

6.11 Joining a Jenkins Cluster

A Jenkins master should be able to join and leave a Jenkins cluster without having to be manually configured. Any Jenkins that connects to a cluster need to get the latest versions of all the configurations before it can start serving requests and execute builds. This must be the case both when a completely new Jenkins is started, and when an existing Jenkins has been restarted. A good time to do this synchronization in Jenkins is during the initialization period, where it is initializing the plugins. Jenkins will be able to run any jobs during this time, which is good since their configurations might be outdated.
7 Conclusions

Jenkins is an application used in software development to automate repeated tasks such as building and testing the latest version of a project. As of now Jenkins is a single point of failure, which means that any software building or testing will come to a halt if Jenkins goes down. This thesis investigated how Jenkins, without modifying its source code, can be made highly available in a way that two Jenkins masters can run in parallel while cooperating and sharing the load. The results were obtained by studying available techniques for high availability and the implementation of Jenkins. Also as a part of the investigation, a prototype of a plugin for Jenkins was developed to get inspiration and to test theories.

The result of this thesis shows that the core features of Jenkins can be adapted to work with several Jenkins masters. Any creation, modification or deletion of a job can be replicated to other Jenkins masters. In addition, executed and completed builds can be replicated to other Jenkins masters. Ongoing builds can however not be made visible on other Jenkins masters. The most serious problem with a distributed Jenkins is to keep other plugins for Jenkins working. Any plugin will assume that only one Jenkins exists, and may therefore make erroneous decisions. No obvious solution for this problem was found, not even if the source code of Jenkins could be modified. A solution would most probably need both the source code of Jenkins and the plugins to be modified.

As described in chapter 2 and 3 there are a great number of ways a high availability solution may be designed and also many issues that need to be considered. By studying the source code of Jenkins, it could be concluded which of the available techniques for high availability that are applicable to Jenkins, and which are not.

Two approaches for high availability were considered: parallel database and active/active. Any other configuration meant that one Jenkins would be in standby, which was not an acceptable solution. The solution parallel database was rejected at an early stage, since it meant that Jenkins masters would share the same home directory. Each Jenkins would then have their configuration files modified behind their back by the other Jenkins masters, which would lead to a confused state. Instead, the active/active solution was considered the superior one.

An active/active solution for Jenkins requires that operations performed on one Jenkins are replicated to any other Jenkins in the cluster. It was found that this replication can only be done asynchronously, leading to issues with data collisions and data looping. As presented in chapter 6.7, there is no way data collisions can be prevented from happening. The available techniques for data collision avoidance would all require modifications to the source code of Jenkins. Therefore, one must decide whether data collisions can be tolerated and ignored, or detected and handled.
Data looping is an issue that needs to be handled in Jenkins, and can only be solved by using a control table.

In conclusion, it has been shown that it is possible, despite the many issues, to adapt the core features of Jenkins to behave properly when several Jenkins masters are running in parallel. The end users and thus the load can be spread across the HA cluster. If one Jenkins or server fails, the users can be moved to a healthy Jenkins. Despite the unsolved issues, the results are indeed satisfying. An application designed to run on only one server has been mostly adapted to function as an active/active system without modifying its source code.

8 Future Work

Some parts of Jenkins did not fit in this research, and need to be further investigated. These areas include Jenkins slaves, Jenkins user accounts, and support for existing plugins for Jenkins. Both Jenkins and the plugins would most likely need to be changed so that they are aware of that there is more than one Jenkins.

In the final week of the investigation, a new idea for solving simultaneous job starts was born. When Jenkins is moving jobs between the stages in the build queue, Jenkins have exclusive rights to the queue. Any operation on the queue is performed synchronized with the queue as synchronization object. Therefore, when a job is about to be scheduled or started, the other Jenkinases can be asked if they have the job in question queued or running. By doing this check synchronized with the queue, it should mean that a job cannot be started while this check is being performed. Either a job is in the queue or it is running. This possibility to prevent simultaneous job starts needs to be investigated further.
References


