# Peer to Peer Grid for Software Development

Improving community based software development
using community based grids

ALI SARRAFI


KTH Information and
Communication Technology

Degree project in
Communication Systems
Second level, 30.0 HEC
Stockholm, Sweden

# Peer to Peer Grid for Software Development

## Improving community based software development using community based grids

Ali Sarrafi

Master of Science Thesis

**Examiner:** Prof. Gerald Q. Maguire Jr.

**Supervisor:** Håkan Kjellman, MoSync AB.

Department of Communication Systems (CoS)
School of Information and Communication Technology (ICT)
Kungliga Tekniska Högskolan(KTH)
Kista, Stockholm, Sweden.

# Abstract

Today, the number of software projects having large number of developers distributed all over the world is increasing rapidly. This rapid growth in distributed
software development, increases the need for new tools and environments to facilitate the developers' communication, collaboration and cooperation. Distributed revision control systems, such as Git or Bazaar, are examples of tools that have evolved to improve the quality of development in such projects. In addition, building and testing large scale cross platform software is especially hard for individual developers in an open source development community, due to their lack of powerful and diverse computing resources.

Computational grids are networks of computing resources that are geographically distributed and can be used to run complex tasks very efficiently by exploiting parallelism. However these systems are often configured for cloud computing and use a centralized structure which reduces their scalability and fault tolerance.

Pure peer-to-peer (P2P) systems, on the other hand are networks without a central structure. P2P systems are highly scalable, flexible, dynamically adaptable and fault tolerant. Introducing P2P and grid computing together to the software development process can significantly increase the access to more computing resource by individual developers distributed all over the world.

In this master thesis we evaluated the possibilities of integrating these technologies with software development and the associated test cycle in order to achieve better software quality in *community driven software development*. The main focus of this project was on the mechanisms of data transfer, management, and dependency among peers as well as investigating the performance/overhead ratio of these technologies. For our evaluation we used the MoSync Software Development Kit (SDK), a cross platform mobile software solution, as a case study and developed and evaluated a prototype for the distributed development of this system. Our measurements show that using our prototype the time required for building MoSync SDK's is approximately six times shorter than using a single process. We have also proposed a method for near optimum task distribution over peer to peer grids that are used for build and test.

# Abstrakt

Idag är antalet programvaruprojekt med stort antal utvecklare distribueras över hela världen ökar snabbt. Denna snabba tillväxt i distribuerad mjukvaruutveckling, ökar behovet av nya verktyg och miljöer för att underlätta utvecklarnas kommunikation, samarbete och samarbete. Distribuerat versionshanteringssystem, såsom Git och Bazaar, är exempel påverktyg som har utvecklats för att för bättra kvaliteten påutvecklingen i sådana projekt. Dessutom, bygga och testa storskalig programvara plattformsoberoende är särskilt svrt för enskilda utvecklare i en öppen källkod utvecklingsgemenskap, pågrund av deras brist påkraftfulla och mångsidiga datorresurser.

Datorgridd är nätverk av IT-resurser som är geografiskt fördelade och kan användas för att köra komplexa uppgifter mycket effektivt genom att utnyttja parallellitet. Men dessa system är ofta konfigurerade för molndator och använda en centraliserad struktur vilket minskar deras skalbarhet och feltolerans.

En ren icke-hierarkiskt (P2P-nätverk) system, åandra sidan är nätverk utan en central struktur. P2P-systemen är skalbara, flexibla, dynamiskt anpassningsbar och feltolerant. Introduktion P2P och datorgridd tillsammans med mjukvaruutveckling processen kan avsevä rt öka tillgången till mer datorkraft resurs genom enskilda utvecklare distribueras över hela världen.

I detta examensarbete har vi utvärderat möjligheterna att integrera dessa tekniker med utveckling av programvara och tillhörande testcykel för att uppnåbättre programvara kvalitet i samhä llet drivs mjukvaruutveckling. Tyngdpunkten i detta projekt var påmekanismerna för överföring av data, hantering, och beroendet bland kamrater samt undersö ka prestanda / overhead förhllandet mellan dessa tekniker. För vr utvärdering använde vi MoSync Software Development Kit (SDK), en plattformsoberoende mobil programvara lösning, som en fallstudie och utvecklat och utvärderat en prototyp för distribuerad utveckling av detta system. Våra mätningar visar att med hjälp av vår prototyp den tid som krävs för att bygga MoSync SDK är cirka sex gånger kortare än med en enda process. Vi har ocksåföreslagit en metod för nära optimal uppgift fördelning över peer to peer nät som används för att bygga och testa.

# Acknowledgements

The author would like to thank Professor Gerald Q. Maguire Jr. for his valuable feedback and guidance through out this project and also Mr. Håkan Kjelman for his kindness, supervision, and help during the completion of this project. He would also like to thank MoSync employees Ali Mousavian, Mattias Frånbarg , and Anders Malm for their valuable help and feedback in finishing the prototype. Finally the author would like to thank Mr. Miles Midgley for his valuable help in debugging and testing the prototype.

# Contents

# List of Figures

# List of Tables

# List of Abbreviations

| | |
|---|---|
| API | Application Programming Interface |
| BOINC | Berkeley Open Infrastructure for Network Computing |
| CVS | Concurrent Versions System |
| DHT | Distributed Hash Table |
| EC2 | Elastic Compute Cloud |
| FLOPS | Floating Point Operations Per Second |
| GB | Giga Bytes |
| GNU GPL | GNU General Public License |
| IDE | Integrated Development Environment |
| IP | Internet Protocol |
| JXTA | Juxtapose |
| KB | Kilo Bytes |
| MB | Mega Bytes |
| Mbps | Mega bits per second |
| MD5 | Message-Digest Algorithm 5 |
| MinGW | Minimalist GNU for Windows |
| MIT | Massachusetts Institute of Technology |
| OS | Operating System |
| P2P | Peer to Peer |
| RAM | Random Access Memory |
| RPC | Remote Procedure Call |
| SDK | Software Development Kit |
| SETI@Home | Search for Extra-Terrestrial Intelligence at Home |
| SPMD | Single process multiple data |
| SWT | Software Testing |
| XML | Extensible Markup Language |
| XMLRPC | Extensible Markup Language Remote Procedure Call |
| YETI | York Extendible Testing Infrastructure |

# Chapter 1

# Introduction

Recently the development of large software products in a distributed manner (even globally) has gained a lot of attention from large corporations who are developing complex commercial software [1]. Traditionally globally distributed software development was considered riskier than collocated development, but Bird et. al. in a study on the development and failures of Windows Vista [2] showed that distributed development of software that has many small components, can be more effective and lead to fewer failures[3]. Corporates software developers often use specific centralized management and quality assurance mechanisms to ensure the quality and performance of their products even when doing distributed development [1], while also benefiting from the advantages of distributed development.

In addition, during recent years the diversification and popularity of different computing systems, especially for mobile platforms, has lead to increased attention to cross-platform software solutions, such as the MoSync SDK[4]. Developing such software with many different components, requires extensive build and test mechanisms. Usually, each revision of the software should be build and tested on multiple platforms, thus the time required for building each package, increases significantly with increasing source code size and the number of platforms. In addition, test and verification of such software on multiple platforms requires extensive processing resources in order to test and verify the software's operation on each of the different platforms and in different configurations. The developers of such software want to make sure that the software works on all of their different target platforms, and seamlessly integrates with different configurations without exhibiting any bugs. Additionally, it is not sufficient to test the software only in a fixed development environment, but rather there is a need to test this software on multiple platforms in which some of the platforms are (or act as if they were) mobile. Projects such as Emulab [5] and similar projects provide new test and evaluation environments for software development.

In commercial projects powerful computing resources and servers together with automatic build and test systems [6, 7] or test suites[8, 9], are often used to continuously test and integrate the software. This requires that powerful machines for building and testing the system be available to the development team.

Open source software development on the other hand is normally driven by

communities [10, 11, 12]. Such software is often managed in one of the following ways[10]:

1. Pure Communities or Peer Production;
2. Open source companies leading the development with parallel developments in the community; or
3. Full cooperation between the open source corporate and the development community.

Regardless of the development process, the potentially large size of the volunteer community contributing to the development of an specific software project is advantageous to the core development team. Large communities result in more review and testing of different parts of the code and continuous improvement in the software's quality. Additionally, the existence of a large community can speed up the process of introducing new features and functionality to the software. For example, Debian GNU linux [13] has a community of more than 3000 developers around the world [14] working together to improve the quality of the software.

In open source software development providing centralized expensive powerful resources is often neither technologically appropriate nor economically efficient for such communities, especially those driven by non-profit core teams. Therefore, open source communities seeking to have many individuals contributing to a software development and maintenance project, need to develop solutions to ensure the quality and stability of their products. Additionally it is desirable to attract individual developers who do not have access to very expensive equipment, while making it possible for them to build and test their code faster and easier.

Grid and distributed computing[15, 16] have been proposed in other areas for solving large and processing intensive tasks without using super computers. Grids usually consist of many computing resources distributed over a network (such as the Internet), collaborating with each other to solve an specific problem. Grid based systems have shown promising performance, for instance the BOINC project [17] has the performance of 4,853,552.7 GigaFLOPS [1] as of January 2011[18].

These system usually are used for running specific applications which can be classified as single process multiple data (SPMD) [19] or bag of task applications [20]. SPMDs are applications in which a single process is repeated over different parts of a large set of data independently in different processors or machines. This type of application is suitable for processing a huge amount of data with a simple process, such as searching through different combinations. On the other hand, bag of task applications are application structures which can be split into completely independent parts. These types of applications are more suitable for large modular tasks that can be run independently.

Grid based systems usually exploit a centralized architecture using a single server to assign tasks and to track the contributors. They often do not have support for dynamic environments and expect stability of the computing resources. Peer-to-peer (P2P) computing [21, 22] on the other hand, is a method of sharing resources among different computers without any predefined

[1] 1 GigaFLOPS = 1 Million floating point operations per second

or static client and server architecture, which makes it suitable for dynamic environments. P2P systems were traditionally used for sharing files rather than computing resources. Combining P2P and grid technologies has recenlty been proposed as a means to provide parallel and grid computing[15, 23, 24] together with the and adaptive features of P2P systems.

Test and development of large scale software in a distributed manner has received relatively little attention until the last few years[8, 25, 26, 27], but with the introduction of distributed version control systems [28] and the huge increase in popularity of the distributed revision control systems [29], the need for distributed testing and development has increased.

Although there are some research activities on running regression software tests in computational or P2P grids, there has been little attention paid to the concept of using P2P distributed computing for both software builds and tests. This is especially true when it comes to consideration of the data transfer overhead and the effects of high task dependency. In this area there are no dedicated systems or analysis available. Building software in a distributed manner is different than running unit testing on P2P systems, because building software has a high degree of dependency among different tasks and there is a need for transferring large amounts of data between peers. In addition, automatically dividing tasks into independent parts may encounter some limitations.

In this project we have investigated the possibility of using a P2P grid architecture for our software build and test process, specifically targeting widely distributed open source software development. Our goal is to find the limits and requirements of such systems, especially in terms of balancing the data transfer overhead with the processing speed gain achieved by parallelizing the build and test processes.

The rest of this document is organized as follows. Chapter 2 gives some of the background of the project, including a survey of related work. An overview of the project methodology and designs are given in chapter 3. Chapter 4 consists of the analysis of the designs and approaches used in this project. Finally some concluding remarks and suggestions for future work are presented in chapter 5.

# Chapter 2

# Background

This chapter covers the theoretical background and previous work in the topics related to this thesis project. Section 2.1 briefly describes distributed and grid computing while covering the previous works in these topics. Section 2.2 provides and overview of previous work on the topics of P2P computing and P2P grids. Section 2.3 discusses the concept on distributed software development. Finally, section 2.4 covers distributed software testing and quality improvement.

## 2.1 Distributed and Grid Computing

A distributed system is defined by Kshemkaylani and Singhal as "a collection of independent entities that cooperate to solve a problem that can not be individually solved"[30]. From a computing perspective a system is called distributed if it has the following features[30, 31]:

- There is no globally distributed common physical clock,
- Different processing entities do not share a global memory,
- Processing entities are geographically separated, and
- There is heterogeneity of the components and computational power.

There are several motivations for using distributed computing systems, these include running a naturally distributed application, sharing of resources, accessing remote data and resources, fault tolerance, reduce the cost/performance ratio, and better scalability[30].

In such systems different processing entities communicate with each other through a communication network, rather than an interconnection network, which is their distinction from parallel processing systems[31]. Figure 2.1 shows a generic architecture for distributed computing systems. Each host has a processor and/or memory unit for use in the distributed application. Some sort of distributed system middleware is used to organize the distributed computing operations in each host using the existing communication application programming interfaces (APIs) of the network protocol stack and the operating system. Figure 2.2 shows the relationship between the middleware and the other parts of a host.

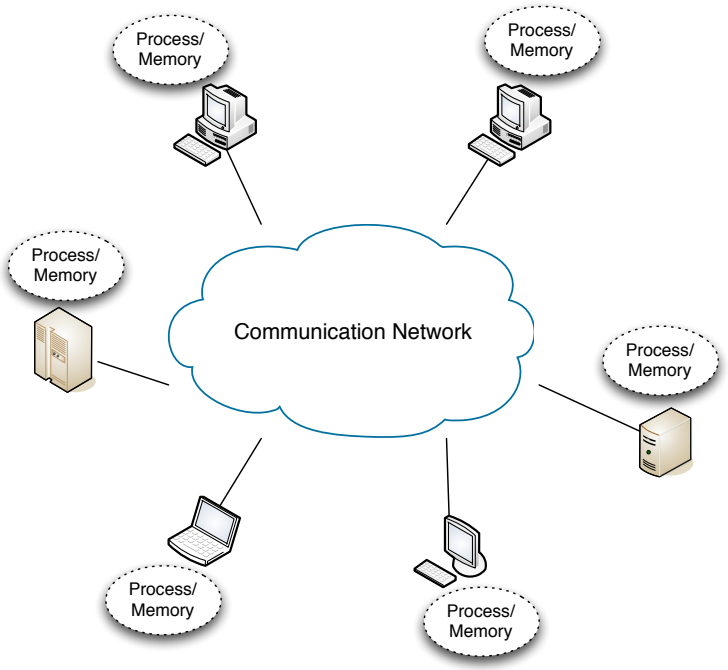


Figure 2.1: A generic architecture for a distributed computing system

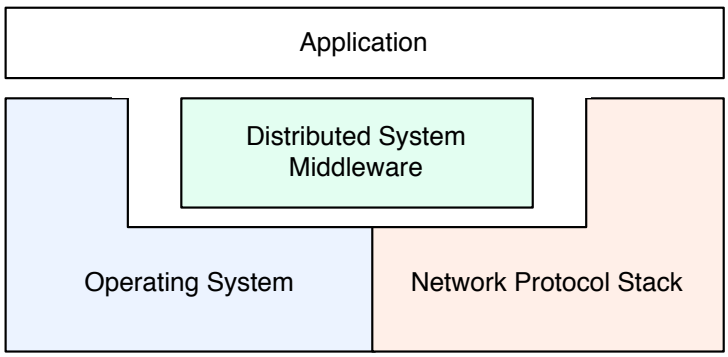


Figure 2.2: A generic architecture for a distributed computing host

Distributed systems often execute independent tasks on different machines when the level of dependency between tasks is very low. When there is a high level of dependency between different processing tasks, then one must consider

this dependency and the effect of communication delays during the design and run-time scheduling of such systems.

Figure 2.3 shows the concept of *dependency delay* due to dependent tasks. If we define a client as the host which sends a request for a process and a slave as a host which runs part of a process, it can be seen in the beginning of the space time diagram that this client starts three different independent parallel tasks that can be run on different hosts. These independent tasks can be run concurrently and without delay and the only limiting factor for each of them is the amount resources available, whereas dependent tasks can only be run after the prerequisites are satisfied and the client has received the results that will become the inputs for the dependent tasks. The time a system must wait before being able to assign new tasks to free resources, i.e. until it receives the results of prerequisite tasks, is defined as *dependency delay*. In Figure 2.3, the dependent task can only execute after the client has received the final input from slave 1. This dependent task may also have needed input from slaves 2 and 3, but these have already been received by the client; hence we can see that the dependency delay is related to the delay to get the last of the needed inputs.

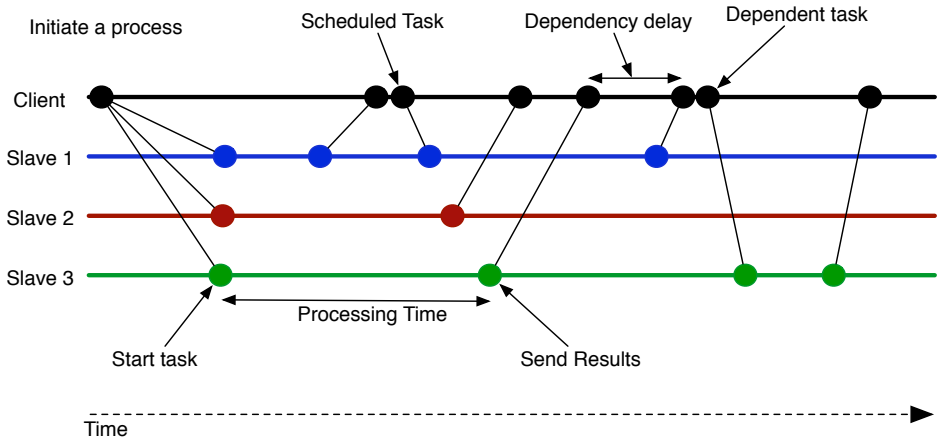


Figure 2.3: Space time diagram of a process with dependency among tasks

A computational grid is a form of distributed computing with a large network of processing entities[16], which provides access to the computational power regardless of the computers' geographical position. A specific type of grid computing called *Volunteer Computing*[32] is a type of grid that has received lots of attraction in scientific and academic projects. A *Volunteer Computing* grid is a network of volunteer computers sharing their computing resources to solve a specific large problem. Projects such as SETI@Home [33] that analyzes radio signals from the space utilizing millions of computational volunteers is one of these computational grids. BOINC [17], which is a multi application volunteer computing grid based on the generic evolution of SETI@Home. The equivalent processing power of these systems, 27000 Giga FLOPS for SETI@Home, and 4,853,552.7 GigaFLOPS for BOINC, shows that volunteer computing is a promising technology for achieving high computing performance at low incremental cost for the entity that wants to run the application.

Most of the currently deployed grid systems utilize a centralized control structure, even though they may have some P2P functionality as well. This centralized control and management approach works like a virtual organization, which results in some barriers for new task submitters who wish to enter the organization and perform computations[34]. P2P based grids are attempts to overcome these barriers and provide a dynamic and flexible system usable by everyone. Figure 2.4 presents a brief comparison between these two major categories of open source grid and volunteer computing systems. Section 2.2 discusses these P2P systems in greater detail.

| Extra Large | |
|---|---|
| Examples | Folding@Home, SETI@Home, BOINC |
| Purpose | Achieve very high performance in terms of FLOPS |
| Architecture | Usually Centralized |
| Application | Usually SPMD Modeled applications |
| Example Applications | Genetics, Space Science, Bio-Chemistry, etc. |

| Small and Medium Communities | |
|---|---|
| Examples | OurGrid, **Jalapeno** |
| Purpose | Multi-Purpose, Research based |
| Architecture | P2P or Centralized |
| Application | Bag of Tasks or SPMD |
| Example Applications | Software Testing, Code Cracking and etc. |

Figure 2.4: A comparison between major grid computing categories

## 2.2 P2P Systems and P2P Grids

P2P systems are networks of computers connected to each other without fixed client-server roles[35]. In P2P systems each node may have different roles and can dynamically change its role depending on the need for this role and its capabilities. Another important feature of P2P systems is their dynamic behavior and potentially high scalability due to their flexible structure and the basic support for adding new nodes and dealing with nodes which depart. There are different varieties of P2P networks ranging from purely decentralized to hierarchical, or even those having a centralized tracker[35].

The traditional application of P2P systems is storage sharing (often characterized by file sharing). These often use an overlay network on top of the internet to interconnect the peers. Resource (typically file) discovery is one of the important aspects of P2P networks, since there is often no central managed directory keeping track of the location of the available resources in the network, therefore techniques such as distributed hash tables (DHT) are used to implement decentralized directories of resources.

The flexibility and dynamic behavior of P2P networks is an attractive feature for use in distributed and grid computing. By using a P2P based architecture, in grids can have a very dynamic and flexible structure with nodes joining and leaving the network at any time. In addition, such networks should not suffer from a single point of failure, thus leading to better fault tolerance[15]. Note that the cost of this increased fault tolerance is replicated copies of data

hence a reduction in efficiency of more than a factor of two in both storage and communication (since each file has to be stored at least twice, the nat least twice as many copies of the file have to be transferred across the network).

P2P grids often use resource discovery mechanisms to search for and find both suitable processing entities and other resources. Resource discovery is a process which can be used by any node to find peers with an available instance of the required resource. Although using P2P systems together with resource discovery increases the flexibility of the grid architecture, there are still problems using such systems, specifically initiation delay and instability. Several attempts have been made to address the issues of resource discovery and search mechanisms in P2P grid systems, with a focus on finding the resources in a P2P network rather than focusing on processing and scheduling.

Therning and Bengtsson proposed Jalapeno, a Java based P2P grid computing system[36]. Jalapeno was developed in Java using the P2P framework and technology provided by the JXTA standard[37]. It uses a hierarchical structure consisting of three types of nodes: manager, worker, and task submitter. Managers in a Jalapeno network act as super peers in the search and discovery mechanisms and also manage the submission of tasks among their peers. The authors claim to achieve a semi-linear speed up in performance when the number of contributing nodes increases up to eight (which may not be a significant number of processing entities for many applications).

Senger et al. proposed P2PComp [38], a framework that uses P2P technology to implement parallel and distributed computing. Nodes in P2PComp have similar functionality and do not have any hierarchical role in the structure. This system was also developed based on the JXTA P2P standard for Java. The aim of this system is to provide a unified framework for running SPMD applications in a flexible P2P environments. The authors claim that P2PComp allows the use of pure P2P philosophy in grids.

Tiburcio and Spohn proposed *Ad Hoc Grid* [39], a self organizing P2P grid architecture developed based on *OurGrid* [40] middleware. *Ad Hoc Grid* adapts the original centralized architecture of *OurGrid* to a more flexible structure by adopting new peer discovery, failure handling, and recovery mechanisms. Their proposed method focuses on running bag of tasks type applications. In their method the peers communicate via multicast messages and form two different multicast groups one local and one for peer discovery. The authors claim to have a dynamic P2P grid architecture with similar performance to *OurGrid*. However, in Ad Hoc Grid nodes do not have any static roles and can switch between being a task executer and a searching peer.

Ma et al. proposed a resource discovery mechanism for P2P based grids[41]. Their model uses a multilevel overlay network, with three different types of peers: super peer-agent, super peer, and ordinary peer. The authors proposed a keyword matching algorithm based on hash tables which can be used together with an ant colony algorithm.

La Andzaro et al. proposed a resource discovery mechanism for decentralized and P2P volunteer computing systems[42]. Their aim is to achieve the simplicity of a centralized system in a scalable decentralized system. Another objective of the authors was to provide constant lookup latency for frequent resources. Their proposed method is claimed to achieve a discovery time of 900ms in a decentralized system with 4096 peers versus the 800ms delay in a centralized search system.

Esteves et al. proposed GridP2P[34], a system for cycle sharing in grids using a P2P structure. The major aim of their proposed method is to provide a system for remote access to idle cycles usable by any ordinary user. The author's claim that GridP2P has a complete set of P2P and grid functions, which helps it to provide efficiency, security, and scalability. They used simulation of up to eight nodes (which again may not be significant for many applications) and claim to have a linear increase in computational performance with an increase in the number of nodes.

## 2.3 Distributed Software Development

Today having distributed teams in different places contributing to a single project, is common in large companies with a global market and multiple development offices[1]. In addition, open source software development has increased the distribution and heterogeneity of software development teams. Projects such as the Debian project with over 3000 developers[14] or The Linux Kernel project with over 6000 developers in 600 different places[43] are examples of open source projects with large development communities. In such projects, there is a need for new development and collaboration tools that support the unique requirements of such distributed teams. For instance, distributed version control systems[44, 45] such as Git [46] or Bazaar [47] were initially developed because of the needs in large open source development efforts.

Distributed version control systems usually store the complete repository on every host [44] and each developer has local access to the complete history of the software source that he/she is contributing to. This method of data management, can achieve high redundancy, hence providing high fault tolerance as each of the developers has a complete copy of the repository - this means that a complete loss would require that all of these copies be destroyed nearly simultaneusly. Such systems can use a semi-P2P architecture since the collaboration between developers may be purely decentralized, but in many cases there is a central publishing site for authoritative code releases.

Using distributed version control and software development has benefits beyond high redundancy and increased fault tolerance. Multiple developers can also work on a specific project and collaborate without publishing their code before it is finalized. Merging and committing new code is more structured with distributed revision management and the probability of breaking a code revision in the main repository is very low and avoidable using the many possible options available in distributed version control systems[44]. An example of preventive actions is Sandboxing, in which developers commit to mirrors of the main repository and their commit will only be integrated into the main branch after it passes all the builds and tests defined by the system.

One of the tools that is used for improving the quality of software development, specifically facilitating build and test is continuous integration[6, 7]. The idea behind continuous integration is to merge and test small parts of the code into the main branch frequently. A continuous integration tool monitors the repository for changes and triggers an automatic build and test after each commit. Using such systems, has proved to improve the quality of software significantly as it performs many tests over small pieces of software[6]. Figure 2.5 shows the basic idea behind continuous integration systems. Using a continuous integration system not only makes merging the source code easier as the merges are done in smaller steps, but it also forces developers to test their code before committing it by sending alerts to the developer or updating the commit's status on the status server. Although continuous integration helps developers to achieve higher quality software, the currently available tools are often centralized. In distributed software development with large communities each separate group of developers may have to operate their own continuous integration server, in addition to the main integration tool.

Distributed revision control systems such as Git[46] or Bazaar[47] provide a semi-P2P environment for developer collaboration and source code management.

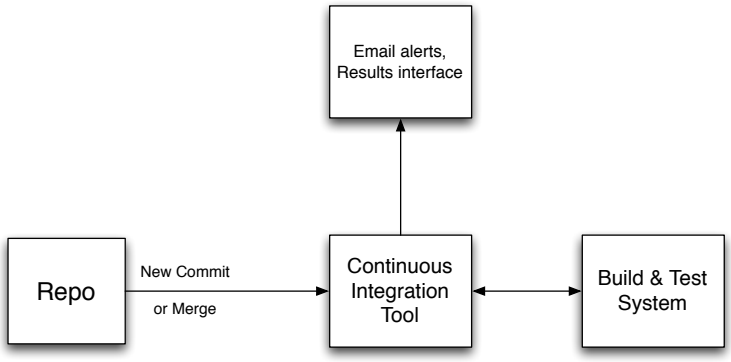


Figure 2.5: Continuous Integration in Software Development

However, both of these systems need a central management system and manual updates. There have been some attempts to adapt pure P2P functionality to revision control and data management in the development environment, thus showing the potential of having P2P functionality in the distributed software development framework.

Mukrejee et al. in a study analyzing the benefits of using P2P technology in software development especially agile software development [48], proposed AKME - a P2P tool that supports distributed agile software development. AKME is designed to be used by small teams and uses a pure P2P architecture. However, the limitation of this system is its scalability as it is explicitly designed for small teams, hence it can not support large distributed teams as well as it supports small teams.

Mukherjee et al. also proposed a purely P2P based version control and collaboration method [49], which controls revisions based on CVS, uses decentralized P2P control, and uses a P2P file sharing and search mechanism on an overlay network to perform file updates and provide peer communication. However, this proposed system only operates on single files and is still under development.

Although there are several distributed revision control systems and some attempts to use pure P2P technology together with revision management system, there are as of yet no scalable systems that can use pure P2P resource discovery for source code management.

## 2.4 Distributed Software Testing

Software Testing (SWT) is defined as the act of verifying the software's quality and functionality against a certain set of requirements and standards[9]. Due to increases in the complexity of new software systems and the increased attention to new software development paradigms such as extreme programming, software testing has become more and more important. The concept of software testing has evolved from bug fixing by individual developers to test automation or agile testing[9]. SWT can be categorized into three different categories:

1. Blackbox Testing
   Blackbox testing, also known as functional testing, occurs when the tester (either the person or the tool) considers the block of code or software as a box with certain functionality. In this type of testing the tester does not consider the internal structure of the code and uses input data and observed output to evaluate the software. The advantage of this technique is that it tests what the program is supposed to do, but it may not be able to achieve exhaustive testing(testing all possible input combinations) in many situations.

2. Structural (whitebox) Testing
   The test conditions in whitebox testing are designed by examining the potential paths through the logic of the source code. This requires the tester to be aware of the code's internal structure. As a result this method of testing can ensure that all paths are examined. The major disadvantage of this technique is that it does not test the functionality of the software. For software that has extensive sanity checks of input values and extensive error handling, much of the testing will be of this code and not necessarily the main paths that will actually be executed for valid data.

3. Hybrid (Graybox) Testing
   Hybrid testing is a combination of the two former techniques, in which the tester and developer are in close collaboration with each other to jointly develop tests of both functionality and completeness.

In addition to these three general types of testing there are several testing techniques that more or less map into one of these categories. Some of these methods are (note that detailed lists and descriptions are available in [9]):

| | |
|---|---|
| Unit Testing: | Testing small part of the code independently. |
| FUZZ Testing: | This task builds MoSync Libraries on Mac OS X. Due to high interdependency of components in this task it cannot be divided into any other subtasks. |
| Exception Testing: | MoSyncWin tasks is used to build MoSync libraries on Microsoft Windows. Due to high interdependency of components in this task it cannot be divided into any other subtasks. |
| Free Form testing: | MoSync IDE is built using this task. This single task generates a Java based IDE for MoSync that can be used on any platform. |

Testing complex software is a large and resource consuming task. There usually exists a huge number of test cases and combinations of these test cases that need to be executed. Distributed or parallel software testing [50, 9] is an approach to increasing the performance of testing, leading to more rapidly getting useful test results. In addition, distributed software testing makes it possible to run multiple different tests at the same time in order to achieve better test coverage. Distributed testing is a topic that has received little attention previously and the few research attempts can be categorized into the following three types:

- Testing on multiprocessor systems,
- Testing on cloud computing resources, or
- Testing on computational grids.

Using multiprocessor computers or local computer clusters is the traditional way of solving complex or processing intensive tasks. There are several examples of attempts to parallelize testing on large and powerful computers. However, this method is neither cost efficient nor feasible for every development team. Therefore other approaches should be considered to provide suitable testing methods for these other types of development teams. We describe three examples of this approach below.

Cloud Computing [51] is a new computing technology in which the computing resources are not directly visible to the end user. Cloud based systems often use virtual machines on powerful servers to provide distributed computing resources. The distributed nature and cost efficiency of such systems makes them a potential target for intensive software testing, hence this method has received major attention from the research community.

Hanawa et al. proposed D-Cloud a testing environment for large scale software testing using cloud computing technology. The authors propose a method specifically designed for dependable parallel and distributed systems. The goal of this system is to address the poor reproducibility of dependable distributed systems, such as high availability servers.

Oriol and Ullah proposed *YETI on the Cloud*[8], a distributed and parallel evolution of the York Extendible Testing Infrastructure (YETI) [52], which is claimed to be a very fast software testing tool. *YETI on the Cloud* uses cloud computing resources from Amazon's Elastic Compute Cloud (Amazon EC2) to achieve high processing power and simulate multiple distributed machines. Their aim is to achieve a solution for distributed and large scale software testing which can use general test cases and operate very fast. However, as of the time of their publication this project is still an ongoing work and there have been no evaluation results presented for this system.

As mentioned earlier grids are also very promising systems for solving large and compute intensive tasks, because of their parallelism. In addition, the transparent grid computing systems can be considered to be cloud computing resources. The features of computational grid systems makes them a suitable for testing large scale software and running multiple parallel tests. At the present time there have been only a few research attempts to adopt software testing to the computational grid. We give some examples of these below.

Duarte et al. proposed GridUnit [27, 53] to use the intrinsic characteristics of grids to speedup the software testing processes. GridUnit is designed based on the JUnit test framework and uses a centralized monitoring and control framework to control the execution of unit tests. They used three different Grid systems to evaluate the performance of their proposed method: Globus [54], ourGrid [55, 56, 40], and Condor [57]. They claim to achieve up to 12 times faster results when more than 45 machines are contributing to the test process.

Almeida et al. proposed an architecture for testing large scale systems using P2P grid technology[26], to achieve better scalability. In order to address the problem of synchronization and dependency between consecutive testers in large scale distributed grid systems, their method uses message passing in a B-tree structure or gossiping messages between consecutive testers.

Li et al. proposed a grid based software unit test framework based on bag of tasks applications[25]. This framework uses a dynamic bag of tasks model instead of the typical static models to achieve adaptive task scheduling. They proposed a swarm intelligence scheduling strategy to improve the efficiency of resource usage and to speed up task completion. They claim to achieve shorter task completion times than random and heuristic testing by approximately 10% and 40% respectively.

# Chapter 3

# Method

Although there have been several attempts to use P2P distributed and grid computing systems for software testing and especially unit testing, there has been little attention paid to using such systems for build and test together. Most of the distributed build systems distribute the build over a predefined set of servers to achieve faster builds. In contrast a P2P distributed system that uses volunteer based computing will have a more challenging task than simply load balancing. The way that source code and revisions are managed can dramatically influence the performance of such a system, because there may be long delays due to the need to transfer large amounts of data in order to be able to run each task. In addition, building and testing often requires some preparation on each platform, which increases the overhead when dividing the build or test task into parts. Additionally, there are situations where dividing a task into multiple tasks would be more expensive than simply running the original task locally because of the data transfer and preparation overheads.

In addition, in cross platform systems such as MoSync SDK, there is a need for building the system and testing it on different platforms. Therefore, in such systems there exists a required minimum number of distributed tasks. This SDK currently has libraries for 77 different mobile platforms and requires at least on three different platforms for a complete system build *regardless* of the required building time due to the requirements for the different cross platform build environments. These requirements make it very hard for the community of developers to easily contribute to such an open source project. In order to attract more developers to their community, open source software projects should facilitate collaboration among many developers who are distributed over the Internet.

The goal of this masters thesis project is to find a suitable structure for distributed build and test to faster developer collaboration in open source software communities. Instead of utilizing powerful centralized servers for build and test, we plan to develop a P2P system that will help developers to share their computing resources. In this way developers can effectively have more computers and devices for running tests on the software, hence the core development team can have greater confidence in the quality and stability of the software.

Since there have been many research attempts on distributed testing (such as [27, 26, 25], [8], and [51]) and also resource discovery mechanisms in P2P

grid and volunteer computing environments (such as [40] and [58, 59]), this project focuses on addressing the issues that arise due to the need to transfer data among peers, hence it will examine the tradeoff between the potential performance gains by adding additional nodes and the potential increase in overhead (especially in the form of delay). Additionally, in this masters thesis we investigated and designed a data management system for a P2P build and test system.

## 3.1 System Requirements

In order to design the P2P based distributed build and test system, we studied the requirements needed for building MoSync SDK packages. Section 3.1.1 describes the features that are required for running our specific type of tasks, while section 3.1.2 describes the structure and specific requirements of MoSync SDK's build system.

### 3.1.1 Task Types

In many grid applications the amount of data that is required to be transferred among nodes is negligible in comparison to the time needed for processing, whereas in build/test systems the data transfer delay is a considerable part of the total processing time. Therefore, in such systems tasks are considered to be data dependent and the processing time may vary depending on the network performance.

As a result, a P2P built and test grid system not only requires a system for managing tasks and distributing them over the network efficiently, but it also requires an efficient way of distributing the data and tasks among the relevant peers. Most of the currently available and general purpose grids do not have an efficient way of managing data and presume that the data transfer delay does not cause a significant decrease in the overall system's performance.

In this project we designed a task management system inspired by bag of tasks applications (as discussed in sections 3.2.1 and 3.3.2), which should also be suitable for build/test applications. We also proposed and analyzed a data management solution for such systems (discussed in section 4.1).

### 3.1.2 Case Study:MoSync SDK

Testing and evaluations of the designs proposed in this master thesis are done on a prototype of a P2P based distributed build and test system, developed specifically for MoSync AB's SDK. MoSync SDK is a cross platform tool providing a single development environment for most of the major mobile handset platforms currently available in the market. Figure 3.1 shows the basic concept behind this platform. The SDK includes libraries for C/C++ which have unique abstracted interfaces and are platform *independent*.

MoSync SDK uses specific components called *runtimes* to provide the interface to the external resources for each MoSync application. These runtimes are *platform specific* and should be run on their respective target platforms, as a result they should also be tested on their specific target platforms.

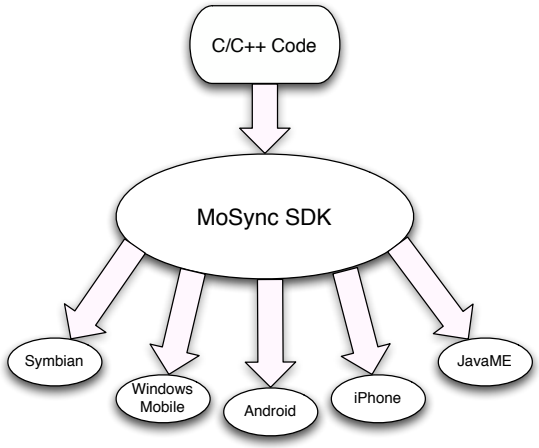


Figure 3.1: The basic concept of MoSync SDK

#### 3.1.2.1 Current Build System

Currently MoSync uses a script based build and test system using a series of Ruby scripts that are invoked hierarchically, these are called MoSync's "workfiles". Each "workfile" at the bottom of the tree is the smallest piece of the build system that can be run independently. However, there may be dependencies between these files. These scripts can be run on either Windows or Mac OS X hosts to build MoSync libraries and tools. The detailed operation and purpose of each component is outside the scope of this document, but is described in the MoSync documentation[60]. The MoSync SDK also includes an integrated development environment which is based on the Eclipse integrated development environment (IDE). This IDE is completely platform independent and can be build on any host using the Apache Ant [61] system.

MoSync runtimes are built using a separate script that can be invoked with a number of different options. The MoSync SDK consists of 78 different runtimes for different platforms (which includes a large number of different phones). Some of these runtimes must be built on their respective platforms, for instance the iPhone iOS and Windows Mobile runtimes should be built on a Mac OS X platform or a Microsoft Windows platform respectively. Figure 3.2 shows the compatibility of different components' build script with different platforms. As it can be seen in this figure, in order to build a complete MoSync SDK package access to at least three different machines (either physical or virtual) is required.

Using the current build it takes more than two hours to do a complete build on a powerful server and requires manual actions on more than one computer to complete the build process, which makes it very hard for individual developers to ensure they have not broken other parts of software by changing a specific part of the whole system.

#### 3.1.2.2 Current Revision Control and Developer Collaboration

Currenlty MoSync uses Git [46] as a hierarchical distributed revision control system for the SDK development. Figure 3.3 shows the structure and

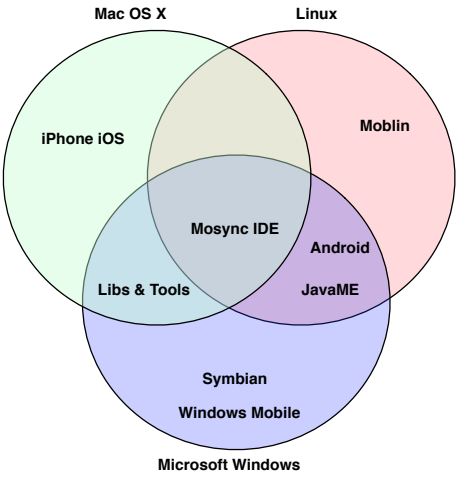


Figure 3.2: Build-compatibility of different MoSync components

interconnections in this revision control system. This structure uses developer sandboxes to achieve distributed independent repositories and uses a master branch which a mirror of the public repository. Each sandbox will be merged with the main repository after passing the build and test requirements. Each sandbox belongs to a single team member, but many developers may commit to an individual sandbox during a specific project.

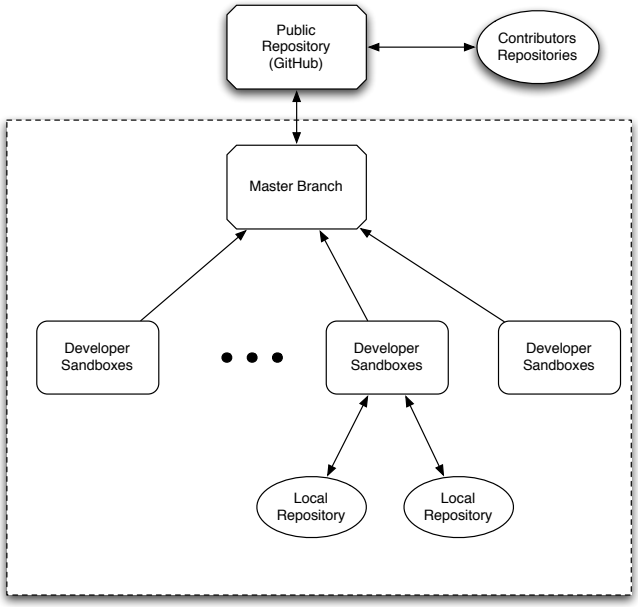


Figure 3.3: Revision control system architecture used by MoSync development team

## 3.2 System Prototype

As the first phase of this project we have developed a prototype based upon the basic functionality required for a P2P grid systems. This prototype can be used as a base for developing the final P2P system. In order to evaluate the proposed structure we used the developed prototype to build all of the MoSync SDK packages. The main goal of this phase was to provide a platform enabling performance and scalability measurements together with providing a basis for expansion to a P2P system.

### 3.2.1 General Architecture

Figure 3.4 shows the network structure for this proposed system prototype. This network consists of a master server which keeps track of the slaves and searches through the slave list to service each request coming from a client. This master server with some extensions can act as super peer (tracker) in the final P2P architecture.

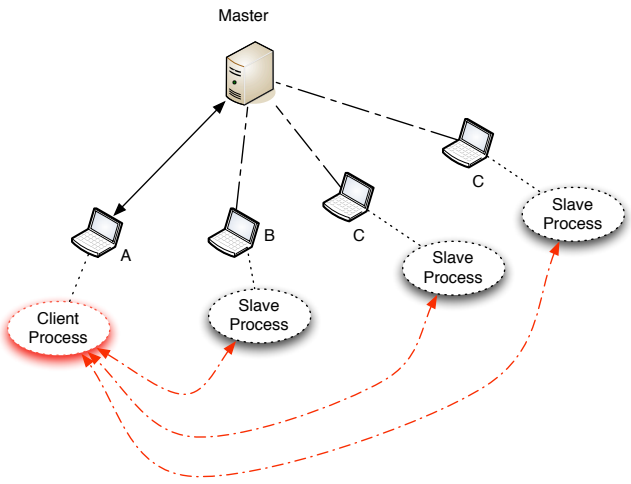


Figure 3.4: The Prototype Architecture

There are several slaves connected to the master server and who independently update their status. Each slave can run an independent build or test script based upon a request by a client. Each slave is assumed to assign all of its (shared) processing power to one client at a time. In order unify the operations that are done by the system, these scripts are registered in the master's list, so they can be run by any of the systems. Each script should consist of the minimum unit of operation that can be performed by a slave. Therefore when generating tasks to process a request the master server distribute the tasks depending on the number of available slaves and their capabilities.

Each host is capable of performing all of the three different roles using its built-in components and run-time configuration. In the current version of the system the role of each host is predetermined using a configuration file. Figure 3.5 shows the main components of the software running on each host and their

connection to the underlying operating system. The base functions are provided by a set of classes that act as a library of basic operations that provide the functionality of the distributed system. These class include everything that is needed to provide an abstract interface for interacting with the distributed system without considering the technology and the underlying layers. Table 3.1 describes these libraries.

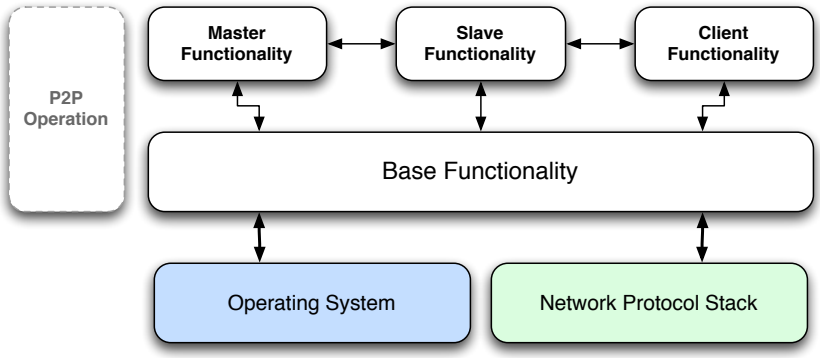


Figure 3.5: Main Components of the System Running on a Host

### 3.2.2 Build Tasks

We used Ruby [63] as the base programming language for our prototype. Ruby is a dynamic programming language, which provides transparent cross platform support for our prototype. In addition, because of the dynamic behavior of the Ruby language interpreter, tasks can be used as independent, self sufficient, dynamic Ruby classes and scripts. We designed a base class in Ruby to provide a unique interface to the system for each task. This base class can be inherited by the task classes.

Figure 3.6 shows the conceptual class diagram for this base class. This class also implements the basic functions that are accessible to the child classes through inheritance and provides access to the system functionality. Table 3.2 includes the list of functions provided by the BuilderBaseClass to its child classes to facilitate implementation of different tasks. By using this class as the parent for a task class the user does not have to care about the details of the underlying system's operation and is only required to implement the specific functionality required to run the task, i.e. building a component or running a specific test.

Table 3.1: Library classes available to different parts of the system

| | |
|---|---|
| Utility | This class provides many system level functions such as compression, decompression, getting host platform information, basic file operations, running external commands while examining their outputs, and examining the underlying operating system for free available TCP ports. |
| FileTransfer | This class provides a simple interface for transferring large files among peers. |
| Logger | This class provides abstract creation and handling of log files for the distributed system. |
| RemoteCall | The RemoteCall class provides an abstract interface for calling remote functions on different hosts. It also provides encoding and decoding of data for the remote functions. This class currently uses XMLRPC over the HTTP protocol to provide remote functionality to the system. Using this class the underlying technology can be changed easily without changing the user functions. |
| ClientHandler | ClientHandler is a child class of RemoteCall and provides an interface to the remote functions available on the client component, such as sending sources, fetching results, and updating status of a task. |
| SlaveHandler | This class is a child class of RemoteCall and provides an interface to the remote functions available on the slave component, such as getting a task script, running a task, and stopping a process. |
| MasterHandler | MasterHandler is a child class of RemoteCall and provides an interface to the remote functions available on the master component of the host, such as query for slaves, generating task scripts, adding a slave, and updating status of a slave. |
| TaskInfo | This class consists the information about a task its requirements, compatible platforms, its subtasks, the associated script, assigned slave info if any, and other compatible tasks. This class is transferred over XMLRPC [62] as the request and response packet. |
| SlaveInfo | The SlaveInfo class contains information about the slaves and is used in both task class and master component to track and store information about the slaves. It provide information such as slave's IP address and RPC port, its platform, script generated for it, unique input filenames and output filenames to be used by it in a particular process. |

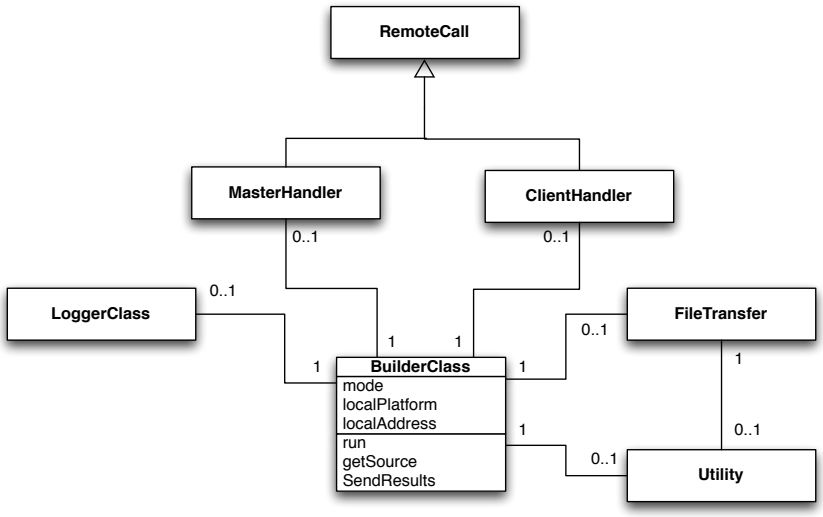


Figure 3.6: General class and object relationship for builder

Table 3.2: Functions available to the build/test tasks using inheritance

| | |
|---|---|
| getSource | Downloads the source from the client and returns the local path for it. |
| sendResults | Sends the results back to the client. |
| packLogsAndExit | Stops the process, makes a package of log files, sends the log files to the client where an error happens. |
| sendError | Sends realtime error messages to the client. |
| cleanUp | Removes all temporary files and variables. |
| log | Provides logging functionality which puts ordered entries into the log file. |
| sh | Provides the ability to run shell commands in a limited manner, handles errors and logging of the command. |
| prepareEnvironment | Prepares the environment for the script by creating temporary directories, adding needed environmental variables, etc. |

### 3.2.3 Main System Components

The software that will be installed on each host contains three major components: the master component, the slave component, and the client component. Based upon the situation, each host may use any of these components. The active components define the role of the host in our overlay network.

#### 3.2.3.1 Master Component

The master component of the system is build around a class named MasterServer. This class uses many other classes from the common library and provides all of the functionality required for searching through the slaves connected to it, generating unique scripts and filenames for each task, pushing common files to the slaves, and providing the slave list to the client. It also instantiates some specific classes which are only used by this class client and/or slave. Figure 3.7 shows the master class relationship with other classes and Table 3.3 describes the private classes that are used by the master class.

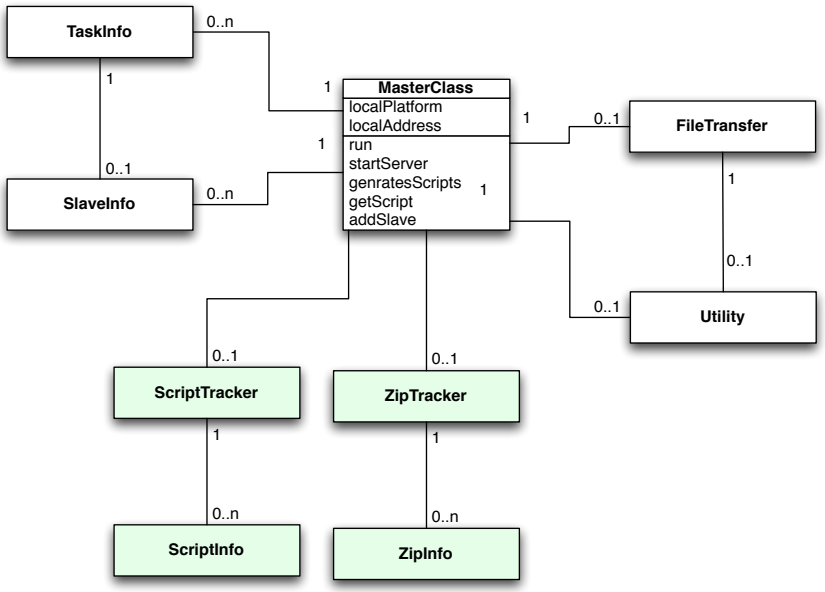


Figure 3.7: General class and object relationship for master

#### 3.2.3.2 Slave Component

A class named SlaveServer is the central part of the slave component in our prototype. This class instantiates and uses multiple classes to perform its assigned operations. Figure 3.8 shows SlaveServer's relationships with other classes of the system. The SlaveServer class also uses ZipInfo and ZipTracker classes to track its local resources and also to update the local zip files (compressed files that contain resources that are not changed frequently), when there is a change in the file on master host.

Table 3.3: Extra classes used by the MasterServer Class

| | |
|---|---|
| ScriptTracker | This class is used for registering, tracking and generating scripts for the registered tasks. |
| ScriptInfo | ScriotInfo is used by ScriptTracker class to store information about each specific script. It includes information about the script name, the task related to it, script platform, and parameters needed to be passed to the script. |
| ZipTracker | The system uses a separate table for common resources used by tasks. In order to decrease the data transfer delay at runtime, master server pushes these files as zip files to each slave that connects to it. This class tracks the resource files in form of zip files and sends them to the slaves if needed. |
| ZipInfo | This class is used by ZipTracker to store the information of each resource file. This information includes MD5 hash of the file, fileName, fileSize and its location. |

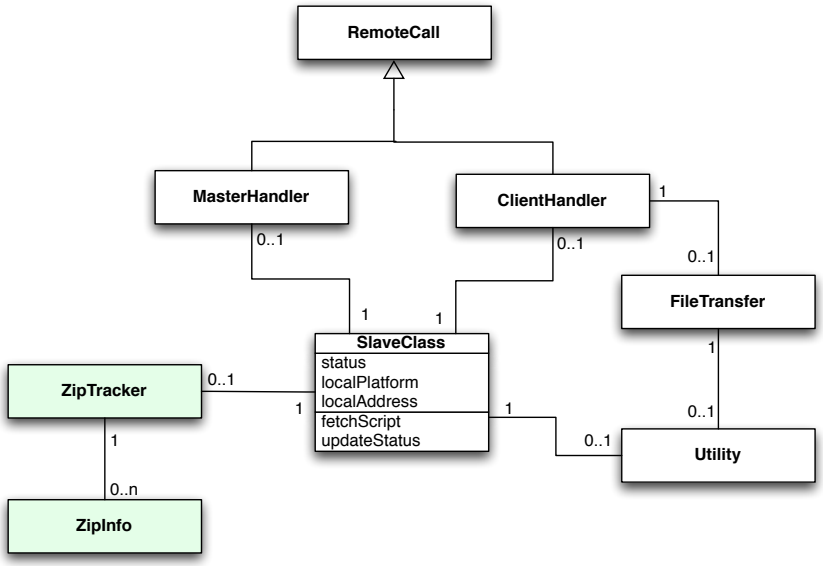


Figure 3.8: General class and object relationship for slave

#### 3.2.3.3 Client Component

Each host also includes a client part which uses a ClientBuilder class as its main component. This class also uses multiple classes from the common functionality library, together with some client specific classes to support the functionality required of the client. The extra classes used by the client are listed in Table 3.4.

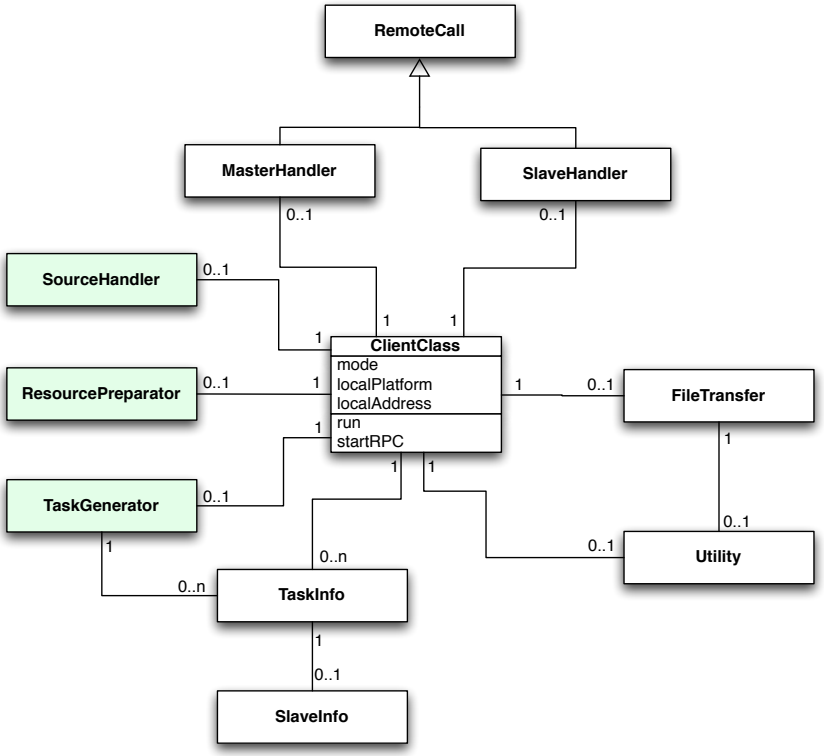


Figure 3.9: General class and object relationship for clients

Table 3.4: Extra classes used by the ClientBuilder Class

| | |
|---|---|
| SourceHandler | This class handles the packing and preparation of the source for the task list. It compresses the source directory into different zip files based on the requirements of different tasks. |
| ResourcePreparator | The ResourcePreparator class handles the preparation of resources for dependent tasks. |
| TaskGenerator | This class generates a list of tasks to be sent to the master. |

### 3.2.4 General Operation

In this section we discuss the general functionality of the prototype. Section 3.2.4.1 describes the messages that are required to be transferred among nodes in order to run a task. Section 3.2.4.2 shows the structure and operation of a client node. Section 3.2.4.3 describes the steps in running a task from the slave point of view. Finally, section 3.2.4.4 discusses the operation of the master and its role in the network.

#### 3.2.4.1 Communication Messages

In order to run a specific task, different nodes with different roles may need to send multiple messages to each other. Figure 3.10 shows a sequence diagram for the messages transferred among different nodes in the system in order to run a specific task. The client starts by sending a build/test request to the server. This request may include multiple subtasks depending on the main task of the client. The master sever sends a list of slaves together with a task list to the client. After receiving this information the client sends (build or test) task requests to each of the slaves in this list.

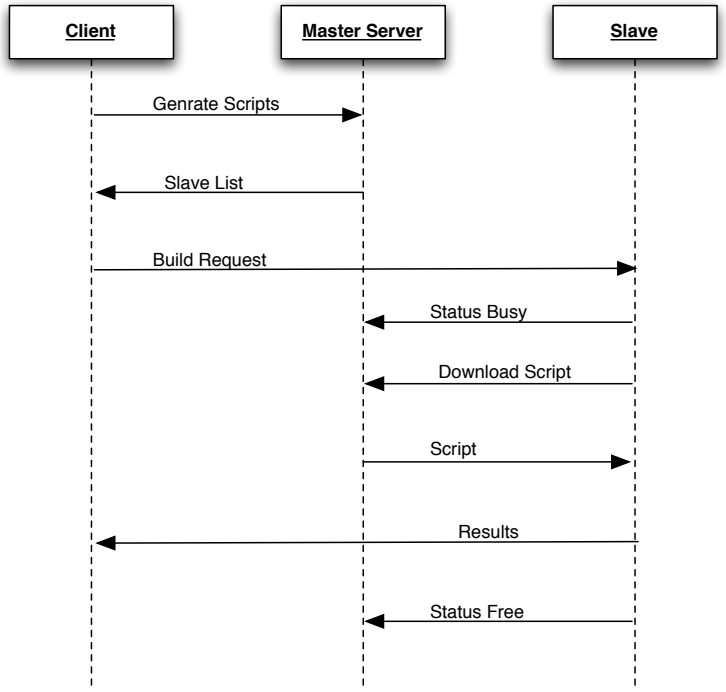


Figure 3.10: Sequence of events for build/test task

After receiving the task request, a slave calls the master server to receive the specific (and unique) Ruby script generated to run the specific task assigned to it. The master server sends the script to the slave in a single message. Due to the small size of these scripts they are sent directly packed in the XMLRPC [62] messages and no extra file transfer connection is needed to transfer them.

The slave then runs the script, which in turn may download the relevant source file from the client via a separate connection. After completion of the task the slave sends any available result file together with the log files to the client. Results and log files are transferred via a separate temporary TCP connection between the client and the slave. This TCP connection from the control connection. These TCP connections are closed after finishing the file transfer and a new temporary connection is initiated for each new data transfer. After finishing its processing and sending the results back to the client, the slave

sends a status message to the master to indicates that it is available for future task requests.

#### 3.2.4.2 Client Operation

A client generates a set of subtasks based on the original main task. The original main task can be building or testing a specific software system (in our case MoSync SDK). These tasks are sent to the master server (a host which is configured to use its master functionality). This master server may in turn divide each of these subtasks into yet smaller subtasks based upon slave availability and the multi-platform nature of the task, then it attaches information about the assigned slaves to the task list and returns this list to the client. If there are not enough available free slaves to finish all of the required tasks, then the master server sends the task list with slave information attached to some of the tasks. The client sends requests to the available slaves in the list and schedules another request to be sent to the master to complete the remaining tasks. The scheduling of tasks, communication with slaves, and processing of the intermediate results is done in the clients in order to reduce the load on the master server. Figure 3.11 shows the operation of a client to complete a requested task. Clients use as many threads as possible to achieve parallelism in sending files and tasks to the slaves.

Each build or test request operation can be complete or partial. Partial operations are handled similarly to the complete operations, with the only difference being in source preparation, and the type of requests sent to the server. For instance a client may start a process that builds or tests a single component of the software, rather than the complete package.

#### 3.2.4.3 Slave Operation

Slaves always consider their assigned tasks to be self sufficient, thus the script should include the information about the client that has generated the request itself and the script does not need to get any additional information from the slave. The main operation of the slave is to provide libraries to the scripts and run them in a controlled environment. Figure 3.12 shows how a slave generally operates. The operation starts with the slave registering itself with the master server as an available slave, then the slave starts a timer which updates its status on the master server every 60 seconds. After receiving a task request, this particular slave downloads the generated script, changes its status to busy, and runs the script as a separate process. The status of the task script is locally monitored to determine if it has finished its operation, then the slave removes any temporary files and variables associated with the specific task and waits for a new task. In addition to task requests, slaves respond to stop requests as well. When a client wants to terminate a previously requested task due to a local error or user request it can send a stop request to the slave, which results in killing the script's process and removing all of its temporary and status variables.

#### 3.2.4.4 Master Operation

The master component, as shown in Figure 3.13, has the roles of tracking and search through slaves, script, resource files, and task types. When receiving a

general build/test request from the client, the master server examines the list of tasks and divides them into subtasks as needed. After dividing the received tasks into subtasks and reorganizing the task list, the master searches for suitable slaves in its list of available slaves, assigns appoperiate slaves to their respective tasks, generates unique scripts for each of them, and sends a new list with this information back to the client. In situations when the master finds a task in the list, which is not compatible with any available slave that is either free or busy, then an error message will be generated and sent back to the client. The master also keeps track of every slave that is connected to it, based upon the status updates that the slaves send. When a master receives an update message it searches through the slave list for the specific slave and if the slave is not in the list, then an error message will be sent back to that particular slave so that the slave can register itself again with the master by sending its complete information.

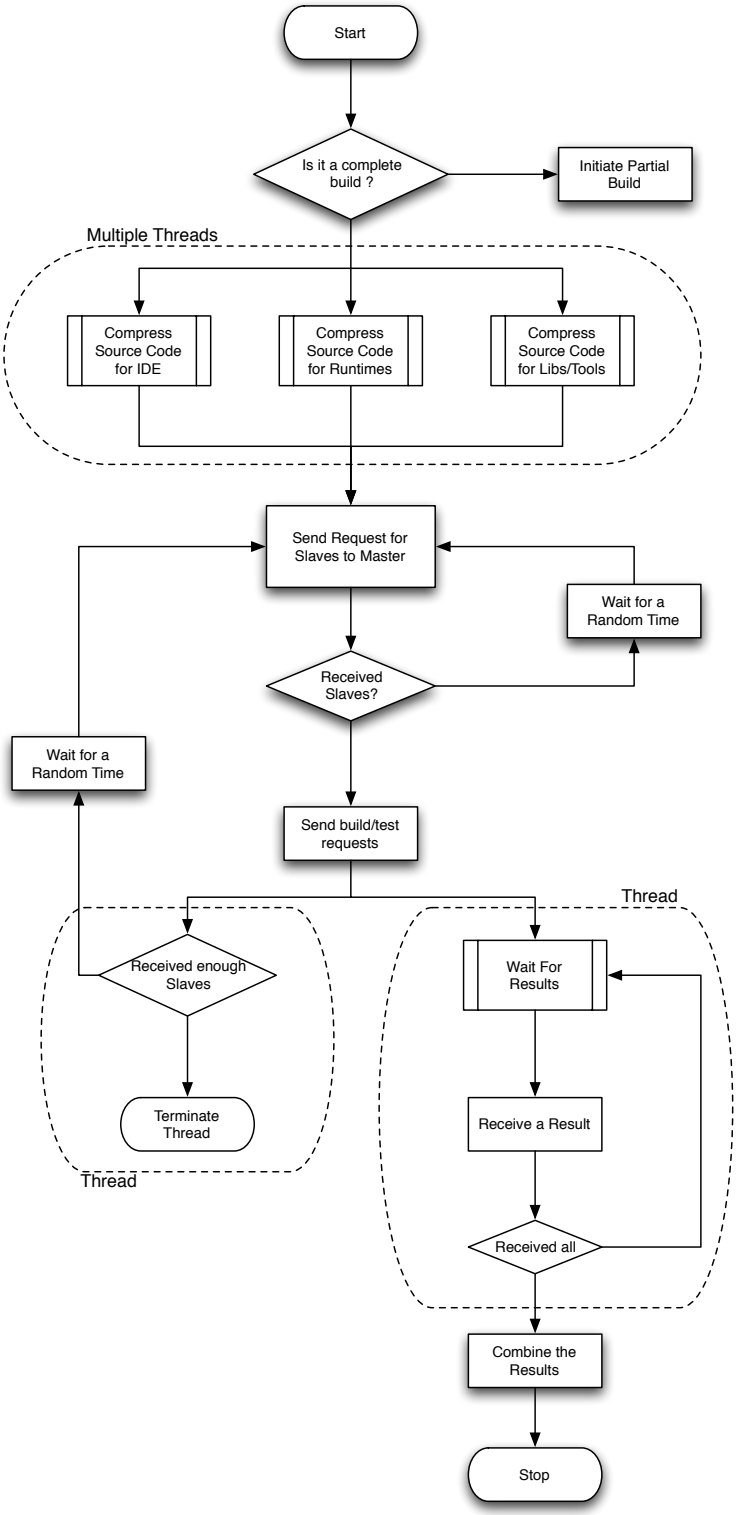


Figure 3.11: General operation of a client

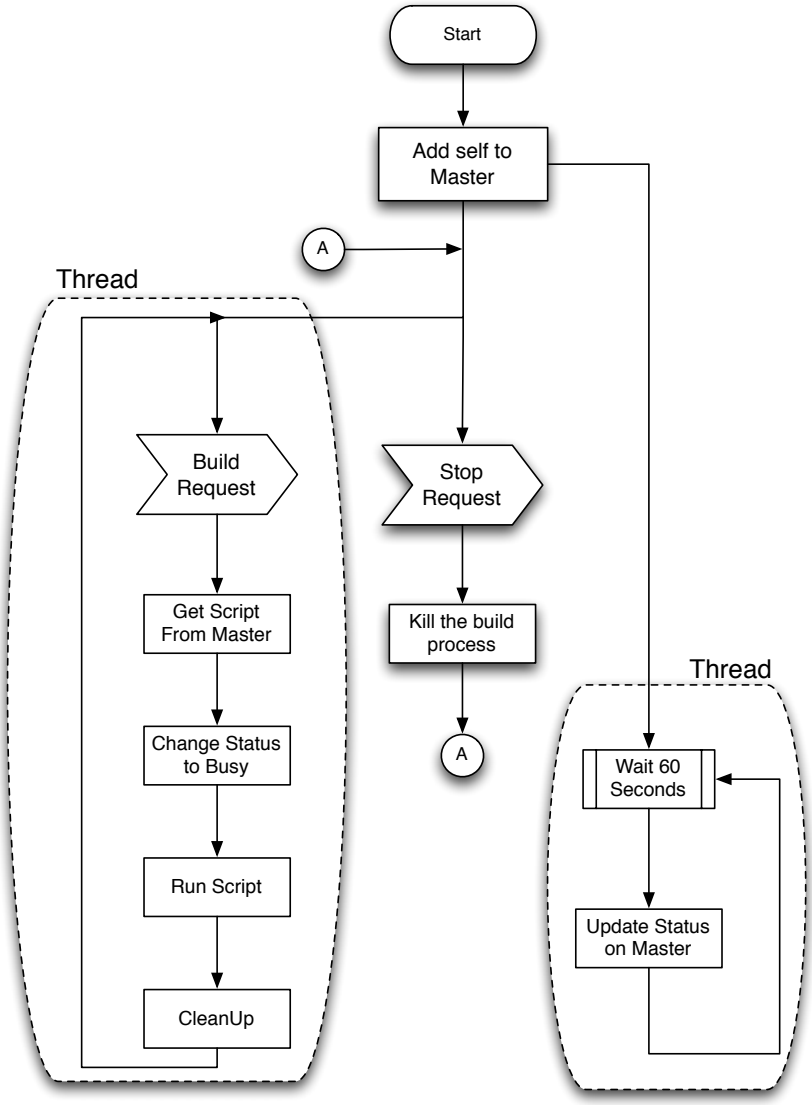


Figure 3.12: General operation of a salve

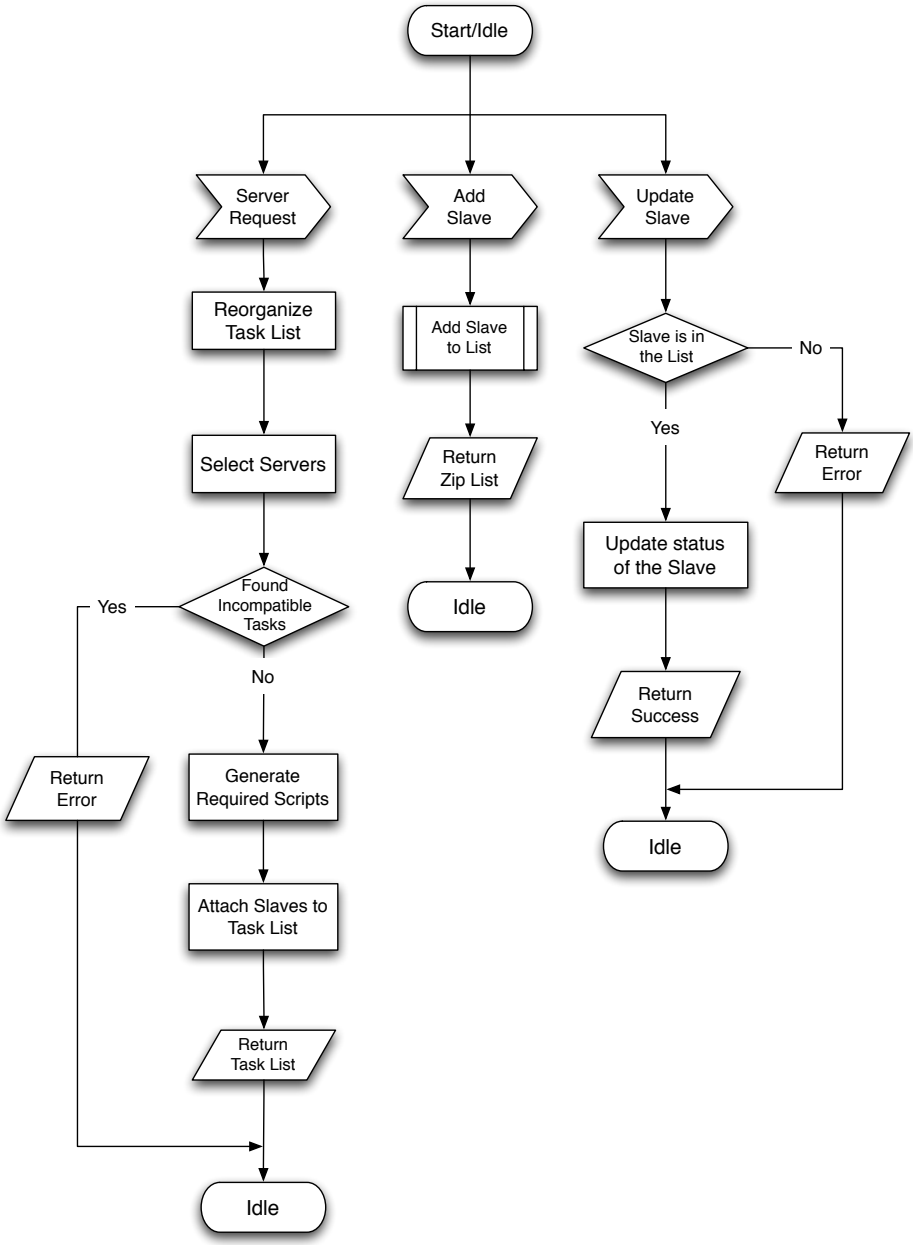


Figure 3.13: General operation of a master

## 3.3 Expansion to a P2P Architecture

Our second step in this project was to design an expansion mechanism for the prototype to have a pure P2P architecture that can be usable by the developer communities. In this new structure, there might be many masters connected to each other working as trackers tracking dynamically available slaves (i.e. user machines). There have been several proposals and a great deal of research regarding P2P search in grids and resource discovery (i.e. [41, 42, 34, 59, 64, 65, 66, 67]). Therefore, our focus has been on methods of distributing tasks over the slaves discovered by a P2P search algorithm.

### 3.3.1 System Architecture

As mentioned in section 3.2.1 the prototype components can be used as the basis for developing the final P2P system. Figure 3.14 shows the proposed hierarchical architecture for the final P2P build/test grid system. Using a hierarchical structure reduces the number of changes required in the prototype system to achieve full P2P functionality. The first step in this transition is that the master functionality will be expanded to perform P2P search and discovery operations (as shown earlier in Figure 3.5 ). Client and slave operations will remain the same as in the prototype, although there may be a need for minor changes.

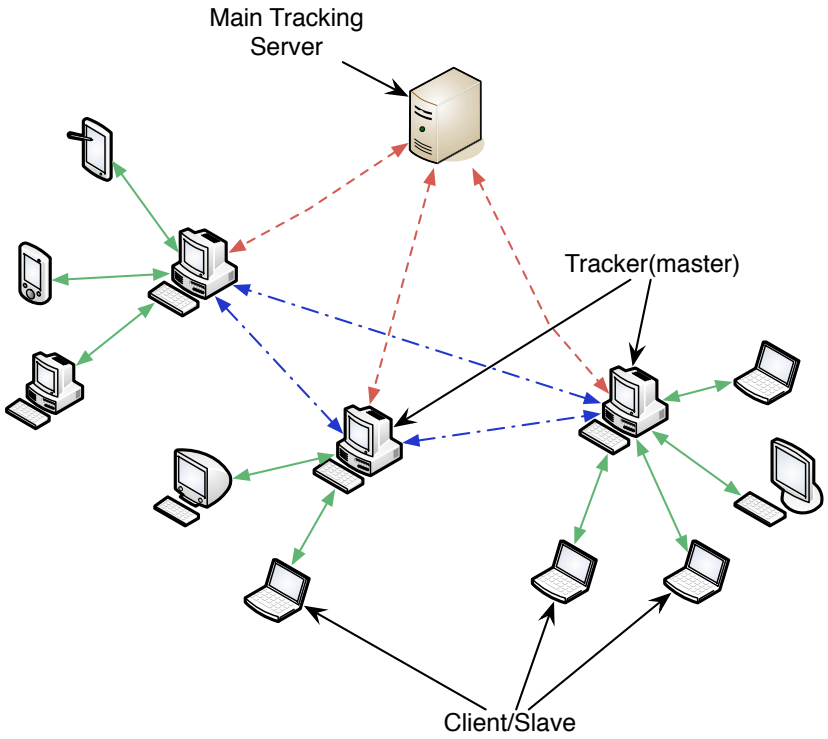


Figure 3.14: Final system architecture and host interconnection

In a real P2P system with nodes scattered over the Internet, communication delay may be significant and high level of control messaging to be transferred among peers can cause additional delay. In order to reduce the control messaging overhead in the final P2P design we refined the messages that are required to be exchanged in order to run a task. Figure 3.15 shows the sequence of messages transferred among peers in order to run a specific task on a specific slave. This figure does not include the search mechanism that is handled by the trackers because as mentioned before our focus in this project is on distributing the tasks over the slaves found by the P2P search mechanism. Additionally, this search can be take place in the background and hence is not in the set of dependencies for processing a given Task request.

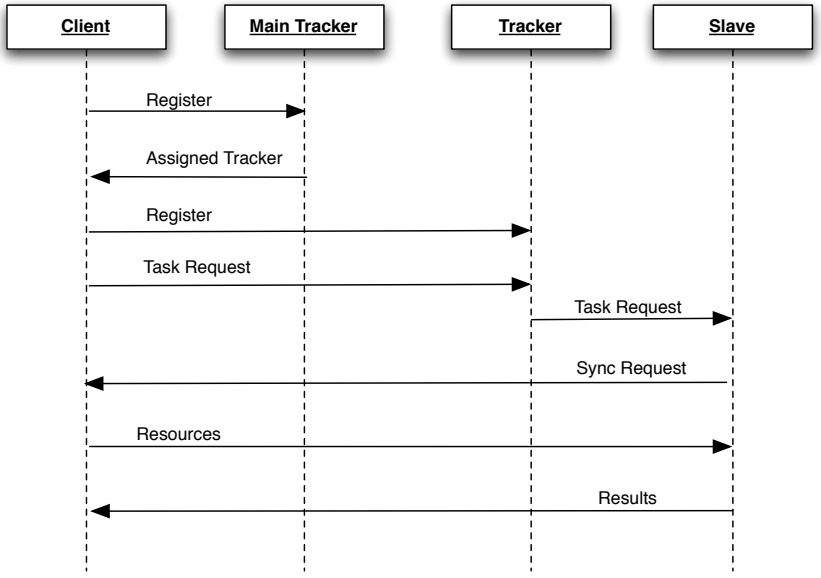


Figure 3.15: Minimum number of control messages required to run a specific task

### 3.3.2 Task Distribution

As will be shown in section 4.2.3, the optimal way to exploit a distributed build grid is to distribute the available tasks over the available slaves proportionally to the processing resources. This method is feasible when the client has a complete list of available slaves in advance. In real P2P systems the information about the available slaves is not known in advance and may change dynamically. P2P networks may be very dynamic and nodes may freely join or leave the network.

In order to overcome this problem and to achieve a balance between the optimal task assignment and P2P search, we have designed a hierarchical task distribution algorithm. This algorithm uses partial search results to initiate task distribution and exploits XML based self dividing tasks. The tasks used in our design are self-sufficient tasks that can be rescheduled and further divided by slaves in the network.

```
<task name="TaskName" id="TaskID">
        <requirements>
          <requirement name="platform" value= "value"/>
          <requirement name="libraries" value= "required libraries"/>
          <requirement name="source" value= "revision" branch="branchName"/>
        </requirements>
        <compatible>task1,task2,task3</compatible>
        <prerequisites>
          <prerequisite name="Task Name"/>
        </prerequisites>
        <subTasks>
          <subTask name="subtaskName" id="subtaskID">
            <option name="optionName" value="OptionValue"/>
          </subTask>
        </subTasks>
        <phases>
          <phase order="0" taskList="task1,task2" />
          <phase order="1" taskList="task3,task4" />
        </phases>
        <initiator address="clientAddress" port="clientPort"/>
</task>
```

Figure 3.16: Task Definition in XML format

We designed XML based task definitions as shown in figure 3.16. These tasks are considered to be self sufficient; meaning they have complete information about the task, the subtasks, the client initiating the process, source information, and requirements for running the task. Using XML defined tasks improves the extendibility of the system and makes it possible for custom tags to be used in different situations. In addition, such structure is easily dividable by just extracting the definition of a subtask and create new xml structure for it. New task chunks inherit their requirements , client information and compatibility information from the parent task. Table 3.5 shows description of the default tags in the proposed XML based task definition.

Using the self-sufficient tasks the system can benefit from better division of tasks and achieve a sub-optimal task division. Figure 3.17 shows the concept of task distribution in our proposed system. The client starts by sending a search request to its tracker, then it selects $k$ slaves from the initial search results and sends a subtask of its main task to each of those selected slaves. Parameter $k$ defines the starting point of the process and can be configured on the client by the user. Each slave in turn can run a new search query and divide its assigned task into chunks based upon the number of secondary slaves it can find. The intermediate slaves can either pick a chunk to run and forward the rest to other slaves or just forward the request based upon their available processing capacity. The process continues until either the slave cannot find any other slaves or the task is not dividable anymore. This process leads to forming a tree overlay for task distribution. Each of the slaves for completing their tasks directly communicate with the client for source updates and sending the results back.

Table 3.5: Description of XML tags used for describing the tasks in the proposed system

| Tag Name | Description |
|---|---|
| task | This tag is the root of each XML definition and covers the whole task. |
| requirements | This tag bound the list of requirements for each task. |
| requirement | This tag defines the actual requirements of the system. |
| compatible | This tag defines other tasks that can be run with this tag on the same machine and do not conflict with each other. |
| prerequisites and prerequisite | These tasks cover the prerequisite tasks that are required for completion of this task. |
| subtasks and subtask | These two tags cover the subtasks that can be derived from the task defined in this XML. |
| phases and phase | List of the steps needs for completion of the specific task. Used to prioritize the subtasks. |
| initiator | Detailed information of the client initiated the process for downloading the sources and sending the results back. |

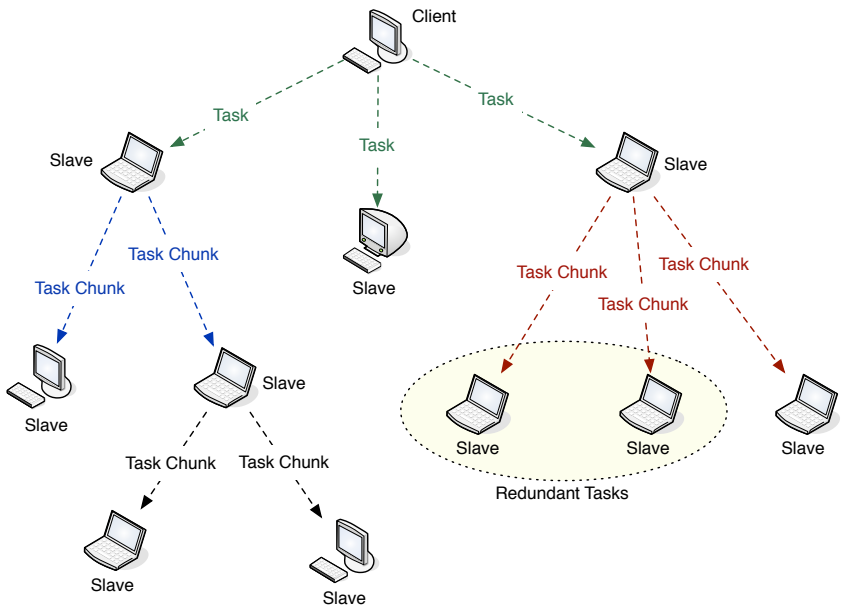


Figure 3.17: Distribution of tasks over the network using self contained dividable tasks

P2P networks are often very dynamic and any node may leave the network at any time. Therefore, forming a tree structure with unreliable hosts as base nodes can cause significant decrease in the overall performance. To prevent

such problems the system can use reliability factors to rank the nodes based on their availability and connectivity. Each node logs its time on the tracker. The tracker calculates the reliability rank of each node and sends this reliability rank along with the search results to the client. In addition, the subtasks can be send to redundant slaves to achieve greater reliability and achieve better expected performance without needing to ignore low ranked slaves. The general queuing theory that supports this approach is described in [68].

# Chapter 4

# Analysis

In the previous chapter we proposed a prototype and a new method of using grids for software build and test. In this chapter we present an analysis of these proposed methods and algorithms in the master thesis together with experimental measurements in order to evaluate the proposed methods. Section 4.1 discusses and evaluates our proposed data management method for use with P2P build and test grids. Section 4.2 then presents our measurements using the developed prototype and compares them to the ideal situations.

## 4.1 Data Transfer and Management

In a distributed build and/or test system transferring the source code from the original host to the hosts performing the tasks causes a delay that cannot be neglected in the performance analysis. In order to completely build the software, the complete source tree may be required on each host. This requirement may lead to significant data transfer overhead (depending on the size of source code). By using a distributed revision control system, we can significantly reduce the amount of data that needs to be transferred among peers by only sending only the differences between the local repositories.

We performed an analysis of seven open source projects, listed in Table 4.1, in order to calculate the amount of data transfer that may be required in different situations to perform a build or test operation. As the information in table 4.1 indicates, transferring the complete source tree among peers can lead to significant file transfer overhead. This overhead may cause a significant decrease in overall performance of the system especially when the network connecting peers has low performance. Therefore, a data management system is needed to decrease the amount of data that actually must be transferred over the network hence decreasing the delay and increasing the overall performance of the system.

All of the projects that we have considered use Git [76] as their revision control system. Table 4.2 shows the commit statistics for these seven projects. As this table indicates the maximum size of a potential change in the source code is significantly smaller than the source tree itself. These numbers also show that there can be a significant decrease in the required data transfer overhead when using a revision control management system as the base for these data transfers. Figure 4.1 shows maximum commit size of each project as a percentage of its

Table 4.1: Seven open source project used as samples for the analysis

| Project Name | Source Size (MB) | Description |
|---|---|---|
| Active Merchant [69] | 9.2 | An extension to the e-commerce system Shopify. Has been in production since 2006 and was released under sponsorship of the Shopify Project. |
| jQuery [70] | 19.9 | A cross browser JavaScript Library for simplifying the client side development. Released under MIT License and GNU GPL and has been active since 2006. |
| Ruby on Rails [71] | 55.0 | An open source web application framework for the Ruby programming language. Released under MIT License and has been active since 2004. |
| Xorg [72] | 66.0 | A set of packages providing access to the X window system and published by the free desktop foundation. Released under X11 License and has been active since 2004. |
| Perl [73] | 172.2 | A high level general purpose dynamic programming language. Released under GNU GPL and has been active since 1987. |
| MoSync SDK [74] | 317.0 | A cross platform mobile development SDK using C++ and Eclipse. Has been active since 2004 and was released under GNU GPL license. |
| Linux Kernel [75] | 1638.4 | The operating system kernel for Linux operating system. Initially released in 1991 under GNU GPL and maintained by a community of developers. |

source tree size. It can be seen from this figure in normal situations peers need to transfer at most 20 percent of a source tree when using the features of revision control system. Therefore, exploiting the features of this revision control software is recommended to increase the efficiency of a P2P build and test system.

Although using a revision control system can increase the efficiency of such P2P build and/or test systems, there are limitations in the type and architecture of the revision control system that can be used. Centralized revision control systems, such as SVN [77] do not fulfill the requirements of P2P build and/or test systems because any change must be committed to the central server before being made available to and transferred using the source code control system. Therefore distributed revision control systems are recommended because peers can pull changes directly from each other without interfering with the main repository. It should also be mentioned that there may be exceptional situations where the changes in the local source code are comparable to the source tree in size, but on average the changes that were merged into the main repository of

these projects are significantly smaller than the source code itself (see table 4.2 and figure 4.1).

Table 4.2: Commit Statistics for the Projects under analysis

| Project | Max Size (MB) | Avg. Size (KB) | Max. Files | Avg. Files |
|---|---|---|---|---|
| Linux Kernel | 99.13 | 79.233 | 4208 | 2 |
| Active Merchant | 0.087 | 196.96 | 19 | 3 |
| MoSync SDK | 41.0 | 35.631 | 737 | 6 |
| jQuery Project | 3.75 | 174.57 | 26 | 2 |
| Perl | 7.86 | 42.721 | 123 | 3 |
| Ruby on Rails | 4.1 | 26.725 | 337 | 2 |
| Xorg | 7.58 | 97.554 | 345 | 3 |

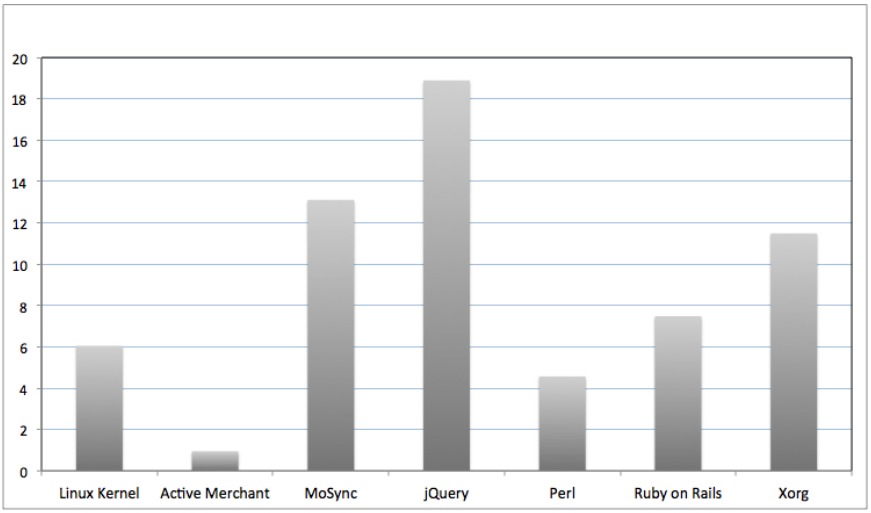


Figure 4.1: Maximum commit size divided by the size of source tree for all of the seven projects shown as percentage.

We have performed an experiment to compare the performance of using direct file transfer and using revision control systems. In order to perform this experiment, we downloaded MoSync SDK's source tree from a single server using both the Git revision control system and a direct file transfer. Both of these methods were used to download the source tree from the same server over the internet and through a 14Mbps connection. The direct transfer method downloaded a compressed version of the source tree (comprising 89.9MB) directly from the server using the HTTP protocol, while the Git connection was used in two different modes: cloning a complete source tree and downloading the differences using Git commands. The differences download measurement was done by reseting the local Git repository to an older revision and branch, then updating it to the latest revision and switching the branch. Table 4.3 shows the results of our measurements. This table shows that if every slave has a copy of the source tree the data transfer can be up to approximately 24 times more efficient. It also shows the inefficiency of using Git to transfer the complete

source tree, but if the data management system uses the revision control system it will only transfer the complete source tree once, as all subsequent transfers will only be files that have been changed.

Table 4.3: Data transfer measurement results

| Method | Average Time (seonds) |
|---|---|
| Direct File Transfer | 67.45 |
| Git Cloning the complete repository | 147.59 |
| Git pulling the differences from 50 commits back | 2.73 |
| Git pulling the differences from 5 commits back | 1.09 |

There are also limitations and risks in using revision control systems as the base of data transfer. In many projects (especially in community based open source software development) developers may cause merge and commit conflicts which should often be handled manually. In such situations pulling the source code from another machine via the revision control system may cause problems in the operation of the system. Therefore, extra actions should be considered when using these systems as the base for data transfer. To overcome this problem we propose using dummy branches together with backup direct transfer. Figure 4.2 shows an overview of the proposed way to exploit the benefits of revision control systems. Using this method will sacrifice performance in some situations, but is more fault tolerant than using the revision control systems alone.

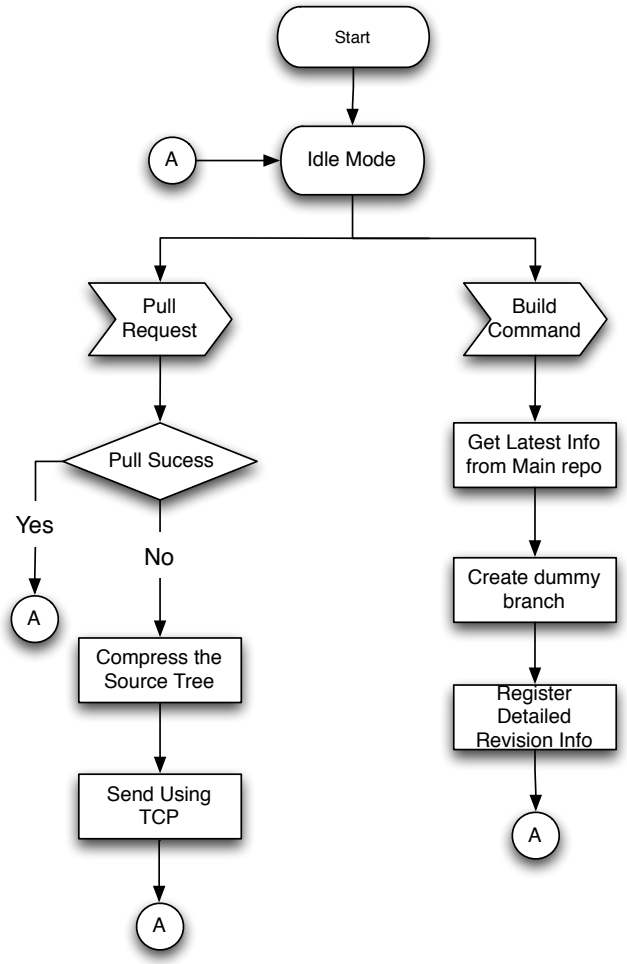


Figure 4.2: Summary of the data transfer algorithm combining revision control and plain TCP connections

## 4.2 Performance and Scalability

As mentioned before the data transfer overhead can cause significant additional delay and as a result lower the performance of P2P build systems. In addition, peers need to run some prerequisite tasks before running their assigned task. These prerequisite tasks can also cause additional delay in processing. All of these limitations may produce a practical limit for dividing tasks into subtasks, i.e., a point after which the performance gain would be less than the overhead penalty.

In this section we present our evaluation and analysis of the performance and scalability of our proposed prototype using both experimental measurements and theoretical analysis. First we discuss the experiment setup we used for our measurements and then discuss the expected performance versus the actual measured performance.

### 4.2.1 Test Setup and Configuration

We have deployed the prototype on a set of virtual and physical machines, which will be used for our measurements and investigations. Figure 4.3 summarizes the configuration of these machines and the configurations used in our test setup. In order to model the machines in a P2P network, we used virtual machines with limited resources as slaves. We use two powerful server that have six and eight core processors, each of these servers run four virtual machines. These servers run Ubuntu Linux Desktop Edition 64bit [78] as their operating system. Each virtual machine has one GB of RAM assigned to it. The Linux virtual machines have four virtual processors each and the Microsoft Windows virtual machines run on two virtual processors. In addition, we have two dual core Mac mini machines acting as Mac OS X slave servers. The Linux slaves run XUbuntu Linux 32bit and the Windows slaves run Microsoft Windows XP Service pack 3 with the MinGW environment configured on them. All of our tests are done using direct source transfer by compressing the source tree into a single zip file and transferring it over the network.

Based upon our analysis of the MoSync source code we divided the main task of building MoSync SDK into eight different general tasks as shown in Table 4.4. Figure 4.4 shows the hierarchy of tasks used in our design. MoSyncWin and MoSyncMac tasks are high priority tasks which are assigned to the available slaves. After assigning slaves to these tasks, then Runtimes and Eclipse tasks are divided into subtasks to occupy all of the available slaves. MacPacking and WinPacking tasks depend upon finishing the other tasks therefore scheduled in the next phase.

### 4.2.2 Basic Task Times

Since the nature of tasks running on our system is very complicated and vary from component to component, providing an exact theoretical formula for calculating the performance is not feasible. Therefore, we measured the time needed for performing the most basic tasks of building MoSync components. We use these timings for further analysis and measurements. Table 4.5 contains the results of these measurements. According to Amdahl's law the shortest time that can be achieved by parallelization is equal to the time required for processing the

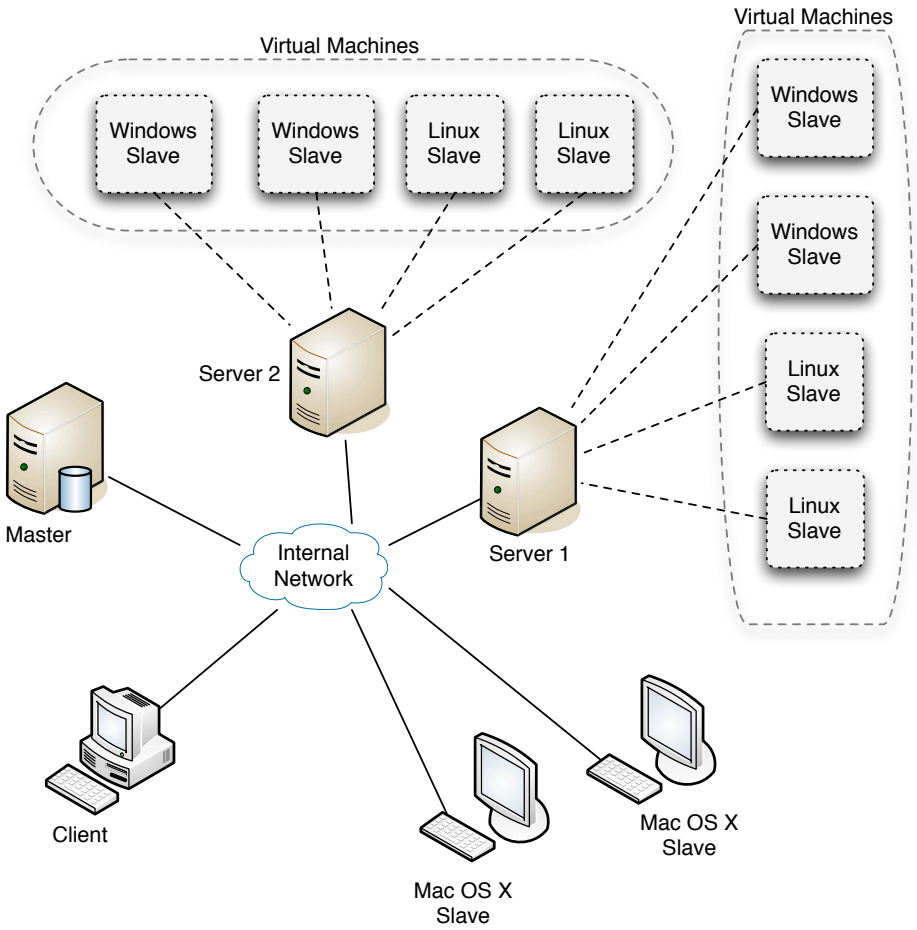


Figure 4.3: The setup that was used for measurements and evaluations

longest unbreakable task in the whole process. The results in Table 4.5 indicate that the shortest achievable time when building complete MoSync packages is greater than equal to the time required for building Windows Libraries plus the time required for running the windows packaging task(1426 seconds).

### 4.2.3 Task Scheduling and Scalability

Division of tasks among available nodes is one of the important factors in the performance of grids. Additionally, how a grid is able to scale with increase in the number of nodes is an important performance benchmark in most of the literature(such as [25, 36, 16], and [20]).

Usually in ordinary grid systems the tasks are divided into small chunks with equal size, then a new chunk is assigned to a new node that joins the system. This method results in a linear increase in theoretical performance. In this project we consider a decrease in build or test time as an increase in performance. Therefore, the ideal increase in performance versus the number of

Table 4.4: Designed build and test tasks

| Task Name | Description |
|---|---|
| Runtimes | This task is a general task that has the potential to build all of the required runtimes. This task is divided into multiple tasks depending on the situation. Since some runtimes can only be built on Mac OS X, Linux, or Windows; in order to build the complete set of runtimes this task should be divided into at least three subtasks. |
| MoSyncMac | This task builds MoSync Libraries on Mac OS X. Due to high interdependency of components in this task it cannot be divided into any other subtasks. |
| MoSyncWin | MoSyncWin tasks is used to build MoSync libraries on Microsoft Windows. Due to high interdependency of components in this task it cannot be divided into any other subtasks. |
| Eclipse | MoSync IDE is built using this task. This single task generates a Java based IDE for MoSync that can be used on any platform. |
| MacPacking | This task packages the results of other tasks that it depends on, into an installation package suitable for Mac OS X. This is a dependent task that requires previous tasks to finish. |
| WinPacking | This task packages the results of other tasks that it depends on, into an installation package suitable for Microsoft Windows. This is a dependent task that requires previous tasks to finish. |

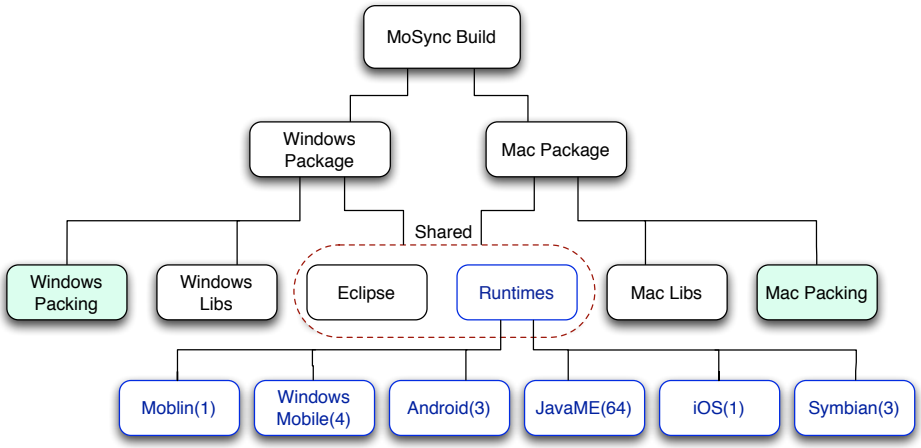


Figure 4.4: Task breakdown for the case of building MoSync SDK

processing nodes, would be as follows:

Table 4.5: Initial measurement results

| Task Description | No. of Runs | Task Completion Time (seconds) |
|---|---|---|
| Building two MoSync SDK packages(for Microsoft Windows and Mac OS X) sequentially and using the original build system that does not have any parallel mechanism. | 4 | 10597 |
| Building a single JavaME runtime on Linux without taking into account the required preparation time. | 20 | 12 |
| Building a single Android runtime on Linux without taking into account the required preparation time. | 20 | 21 |
| Building Moblin runtime on Linux without taking into account the required preparation time. | 20 | 52 |
| Building a single Windows Mobile runtime on Microsoft Windows without taking into account the required preparation time. | 20 | 78 |
| Building a single Symbian runtime on Microsoft Windows without taking into account the required preparation time. | 20 | 95 |
| Building iPhone runtime on Mac OS X without taking into account the required preparation time. | 20 | 28 |
| Building prerequisites for building a runtime on Microsoft Windows. | 20 | 134 |
| Building prerequisites for building a runtime on Linux. | 20 | 42 |
| Building prerequisites for building a runtime on Mac OS X. | 20 | 34 |
| Windows Libraries | 20 | 1183 |
| Mac OS X Libraries | 20 | 752 |
| Windows Packaging | 20 | 243 |
| Mac OS X Packaging | 20 | 158 |

$$T_{process} = \begin{cases} T_{single} - N_p * T_{chunk} & \text{if } N_p < N_{chunks} \\ T_{chunk} & \text{if } N_p > N_{chunks} \end{cases} \quad (4.1)$$

where $T_{process}$ is the actual time required for running the build or test package on the achieved distributed system, $T_{single}$ is the time required for running all of the sub tasks(chunks) on a single local machine, $N_p$ is the number of nodes(processes) running the the tasks, $N_{chunks}$ is the number of possible task chunks, and $T_{chunk}$ is the time required for running a single task chunk over a single processor.

According to equation 4.1 the effects of an ideal increase in performance are as shown as blue points (and a blue curve) in Figure 4.5. Figure 4.5 also shows

the measured performance of our prototype. We used the Runtimes tasks for both our theoretical calculations and experimental measurements because this set of tasks can be easily split into uniformly small tasks. The measurements shown in Figure 4.4 is based on running the experiment for 100 times (to reach the confidence interval of 0.01) and using the average. Since each of our nodes in the setup had more than one processors the decrease in processing time is approximately linear when the number of processes increases, but when a new remote node is added to the system a small increase in the processing time is visible. This increase occurs because of the network overhead and the extra delay caused by running prerequisite tasks on the remote machine. As figure 4.5 shows, due to the additional delays the gap between the ideal system and the actual system increases with increase in number of processing nodes. An analysis over these measurement results shows that the delay in processing caused by the inter process delay is negligible in comparison to the delay caused by the network. Therefore, we can consider the delay to come from the data transfer over the network.

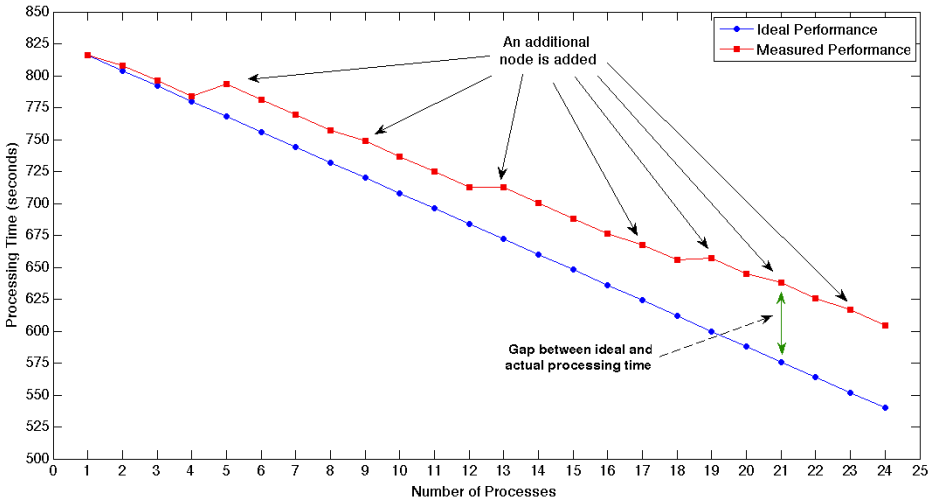


Figure 4.5: Processing time versus number of processes with linear task division

Another way of scheduling and dividing the tasks is to fully benefit from the number of available processors to balance the number of tasks running on each node by dividing the number of subtasks by the number of available nodes. Using this method the processing time is significantly lower than when using linear task assignment. As a result the ideal processing time versus the number of processors is described in equation 4.2.

$$T_{process} = \begin{cases} \left(\frac{N_{chunks}}{N_p}\right) * T_{chunk} & \text{if } w \text{ is 0 and } N_p < N_{chunks} \\ round\left(\frac{N_{chunks}}{N_p}\right) * T_{chunk} + T_{chunk} & \text{if } w \text{ is not 0 and } N_p < N_{chunks} \\ T_{chunk} & \text{if } N_p > N_{chunks} \end{cases} \tag{4.2}$$

where $w$ equals to $mod(N_{chunks}, N_p)$.

Equation 4.2 indicates that ideally there should be a decrease in processing time by a factor of N as shown in figure 4.6. Figure 4.6 also includes the results from our measurements too. This figure shows that the real system follows the scalability of an ideal system with a small gap. Unfortunately, the performance gap increases as the number of nodes increases, thus indicating that there are diminishing returns for increasing the number of processors. This expansion is due to the network and processing overhead. When reaching a certain number of processors(in our case 16) the ideal processing time does not decrease significantly with an increase in the number of processors. This is the point in which the optimum number of processors are being used and increasing the number of processors does not affect the performance of the system. In the prototype the performance starts declining when the numbers of nodes exceeds this point and adding new processors causes the actual performance of the system to decrease (as can be seen in the curve with an increase in processing time when the number of processors exceeds 16).

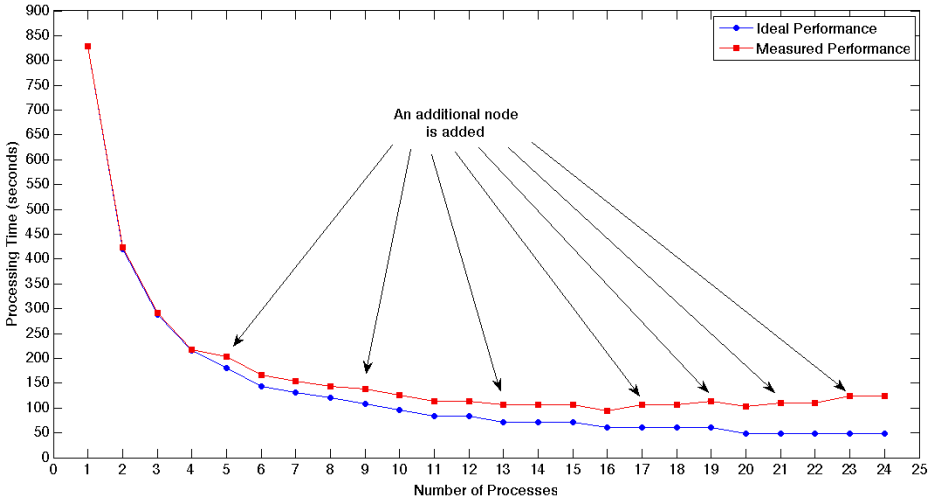


Figure 4.6: Processing time versus number of processes with optimal task divisions

The actual building and testing of software is a more diverse task than the subset of tasks represent by Runtimes. The complete set of tasks consists of different types of sub-tasks either dependent or independent sub-tasks. In order to evaluate the performance improvement introduced by our prototype when building two complete MoSync SDK packages, we measured the time required for building these packages with different numbers of simultaneous jobs. Table 4.6 shows the results of our measurements when building two MoSync packages over our prototype. The combination of tasks are chosen in a way that can be easily detected by the system without any manual intervention.

Table 4.6: Package Build Measurement Results

| No. of Jobs | No. of Nodes | Description |
|---|---|---|
| 1 | 2 | Building each package on a single machine (one Windows and one Mac OS X) sequentially. |
| 2 | 2 | Building each package on a single machine (one Windows and one Mac OS X) in parallel. |
| 4 | 4 | Building on four nodes using full power of each machine. Tasks were divided as MoSyncWin, MosyncMac, Eclipse, Runtimes, Windows Packaging, Mac Packaging. The packaging tasks depend on finishing all of the others (packages share components). |
| 6 | 6 | Building on six nodes using full power of each machine. Tasks were divided as MoSyncWin, MosyncMac, Eclipse, Three Runtimes Tasks, Windows Packaging, Mac Packaging. The packaging tasks depend on finishing all of the others (packages share components). |
| 8 | 8 | Building on eight nodes using full power of each machine. Tasks were divided as MoSyncWin, MosyncMac, Two Eclipse Tasks, Four Runtimes Tasks, Windows Packaging, Mac Packaging. The packaging tasks depend on finishing all of the others (packages share components). |

Figure 4.7 shows the results of our measurements of building complete packages. This figure shows the performance achieved by using our distributed prototype builder is six times more than the performance achieved by the single machine sequential builder. Figure 4.8 shows the achieved processing time with maximum possible parallelism in comparison to the ideal performance when using the same degree of parallelism. This figure supports the previous expectations regarding the performance according to the Amdahl's law. The actual achieved processing time is 305 longer than the ideal processing time. This time is mostly because of the data transfer delay among different nodes.

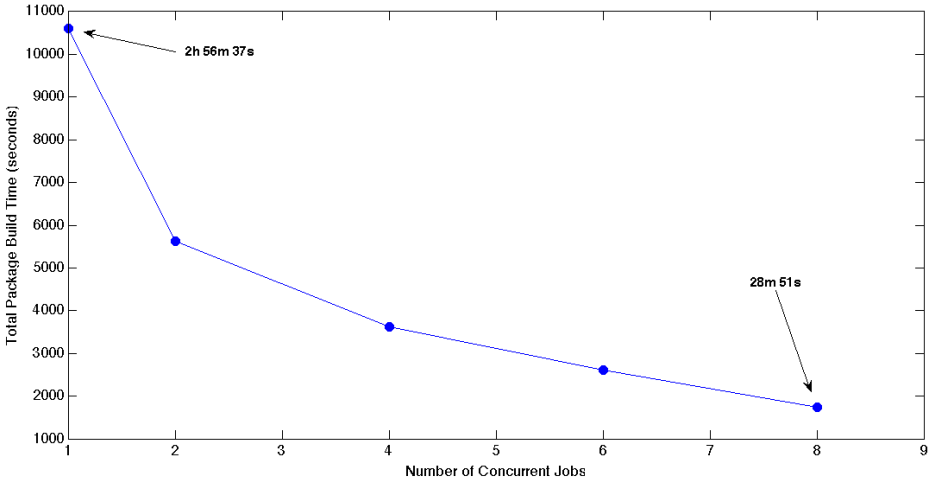


Figure 4.7: System performance building two complete packages

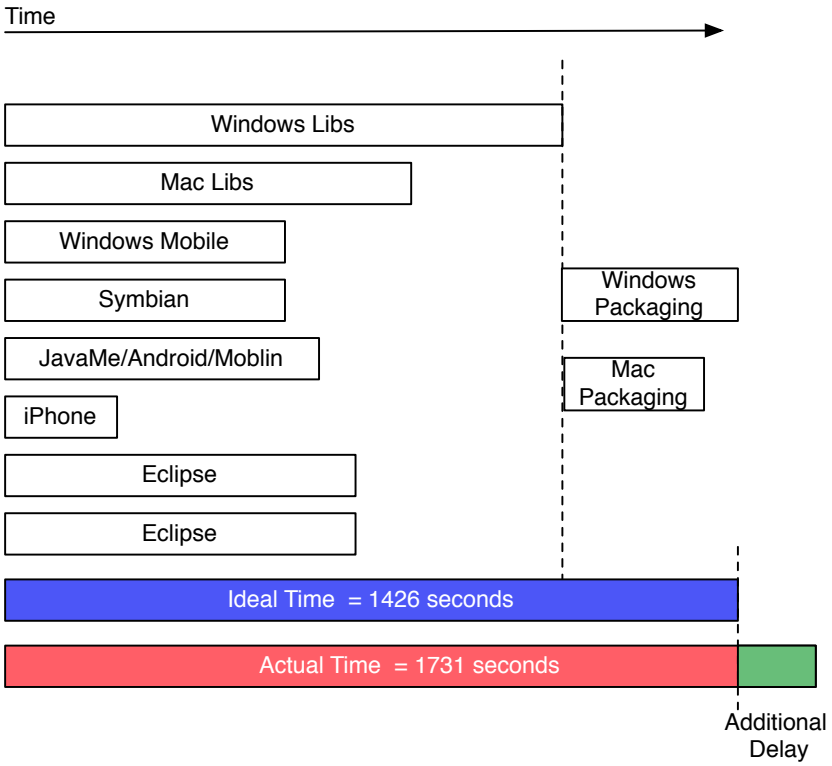


Figure 4.8: The achieved time with maximum possible parallelism

# Chapter 5

# Conclusions and Future Work

## 5.1 Conclusion

In this thesis project we explored the possibilities of using P2P grids for building and testing software. The goal was to investigate the limits and benefits of running more complex tasks in comparison to running computationally intensive tasks using P2P grids. Our main focus was to evaluate the role of the data transfer delays in the performance of such grids and also the effects of partitioning of tasks on the performance of such grids. We have developed a prototype for running such tasks using a hierarchical structure. Our prototype is designed in such a way that can be expanded to a real P2P architecture without major modifications. This prototype improved the performance of building MoSync SDK by approximately six times over a test setup (that was designed to model a network of multiple nodes). This performance improvement illustrates the possibility of running such tasks over grids.

One of the reasons for the limited increase in performance is that, in contrast to ordinary grid tasks, build and test tasks have high data transfer requirements and network delay can cause an additional decrease in the performance of the grid based system. Therefore, we propose a data management system that exploits the functionality provided by revision control systems to improve the efficiency of transferring data over the network. However, there are limitations in using revision control systems as the base for data management of the P2P build and test grid. It is recommended that distributed revision control systems be used for this purpose rather than centralized systems. Additionally, there should be a mechanism for detecting and ignoring source code conflicts during the transfer. Therefore, we designed a system that uses dummy branches over the Git revision control system. This method uses direct transfer when there are unresolvable conflicts.

Tasks running on P2P build and test systems can only be divided into smaller tasks up to a certain degree. This limitation requires additional considerations when deciding upon task distribution. In order to overcome the limitations dictated by the nature of the tasks and to exploit the full power of the distributed system, we proposed a method of task distribution that evenly divides the

tasks (that are possible to divide) over the set of nodes that are available for running them. Our measurements show that this method of dividing the tasks can increase the achieved performance by a factor of approximately six in the developed prototype (see Figures 4.5 and 4.6). Although dividing the tasks evenly over the available nodes leads to a significant increase in performance, in a real P2P environment it is not feasible to have the list of available nodes in advance without extra penalties. In order to overcome this problem we designed a tree based hierarchical task distribution method that can reach a near-optimum performance in a real world scenario.

P2P grids can help in increasing the number of contributions among open source software communities by providing the opportunity for every developer to build and test their codes without having access to multiple machines locally. Developing cross platform applications may need building and testing the software on multiple platforms. Therefore, using a P2P system can also help such projects by providing the possibility of building and testing the software over multiple platforms with different configurations, thus enabling contributes from developers who only have one of the platforms.

## 5.2 Future Work

The future work includes the following:

- Evaluate the system's queueing and scheduling capabilities when responding to requests from multiple clients.
- Develop and evaluate a test application over the middleware prototype.
- Integrate the proposed data management system and P2P searching mechanisms into the prototype.
- Analyze the effect of adding a P2P search mechanism in terms of resulting delay and the overall system performance.
- Evaluate performance of the new system and compare it to the prototype.
- Exploit available source code analysis tools to achieve automatic modularization and task generation.
- Analyze the security requirements for P2P build and test grids.
- Add automatic feature detection to detect the capabilities of the hosts.
- Add a mechanism for detection and usage of idle processing power of the slaves.

# References


[1] M. A. Cusumano, "Managing software development in globally distributed teams," *Commun. ACM*, vol. 51, pp. 15–17, February 2008. [Online]. Available: http://doi.acm.org/10.1145/1314215.1314218

[2] "Microsoft Windows Vista," http://windows.microsoft.com/en-US/windows-vista/products/home.

[3] C. Bird, N. Nagappan, P. Devanbu, H. Gall, and B. Murphy, "Does distributed development affect software quality? An empirical case study of Windows Vista," in *Proceedings of the 31st International Conference on Software Engineering*, ser. ICSE '09. Washington, DC, USA: IEEE Computer Society, 2009, pp. 518–528. [Online]. Available: http://dx.doi.org/10.1109/ICSE.2009.5070550

[4] MoSync, "MoSync cross platform SDK," http://www.mosync.com.

[5] D. Johnson, T. Stack, R. Fish, D. M. Flickinger, L. Stoller, R. Ricci, and J. Lepreau, "Mobile emulab: A robotic wireless and sensor network testbed," in *INFOCOM 2006. 25th IEEE International Conference on Computer Communications. Proceedings*, april 2006, pp. 1 –12.

[6] P. Duvall, S. Matyas, and A. Glover, *Continuous integration: improving software quality and reducing risk*, 1st ed. Addison-Wesley Professional, 2007.

[7] J. Holck and N. Jorgensen, "Continuous integration and quality assurance: A case study of two open source projects," *Australian Journal of Information Systems*, vol. 11, no. 12, pp. 45–53, 2004.

[8] M. Oriol and F. Ullah, "YETI on the Cloud," in *Software Testing, Verification, and Validation Workshops (ICSTW), 2010 Third International Conference on*, 2010, pp. 434 –437.

[9] W. E. Lewis, *Software Testing and Continuous Quality Improvement*, 3rd ed. CRC Press, 2009.

[10] P. de Laat, "Governance of open source software: state of the art," *Journal of Management and Governance*, vol. 11, pp. 165–177, 2007. [Online]. Available: http://dx.doi.org/10.1007/s10997-007-9022-9

[11] E. Capra, C. Francalanci, F. Merlo, and C. Rossi Lamastra, "A survey on firms' participation in open source community projects," in *Open*

*Source Ecosystems: Diverse Communities Interacting*, ser. IFIP Advances in Information and Communication Technology, C. Boldyreff, K. Crowston, B. Lundell, and A. Wasserman, Eds. Springer Boston, 2009, vol. 299, pp. 225–236.

[12] G. von Krogh and E. von Hippel, "The promise of research on open source software," *Management Science*, vol. 52, no. 7, pp. 975–983, July 2006.

[13] (2010) Debian GNU Linux. http://www.debian.org. [Online]. Available: http://www.debian.org

[14] Debian Community, "About Debian," http://www.debian.org/News/2010/20101215, Last Checked: 28 Jan 2011.

[15] G. Fox, D. Gannon, S. hoon Ko, S. Lee, S. Pallickara, M. Pierce, X. Qiu, X. Rao, A. Uyar, M. Wang, and W. Wu, "Peer-to-peer grids," in *In Grid Computing Making the Global Infrastructure a Reality*. John Wiley and Sons Ltd, 2010.

[16] F. Berman, G. Fox, and A. J. G. Hey, *Grid Computing: Making the Global Infrastructure a Reality*. New York, NY, USA: John Wiley & Sons, Inc., 2003.

[17] D. P. Anderson, "BOINC: A System for Public-Resource Computing and Storage," in *Proceedings of the 5th IEEE/ACM International Workshop on Grid Computing*, ser. GRID '04. Washington, DC, USA: IEEE Computer Society, 2004, pp. 4–10. [Online]. Available: http://dx.doi.org/10.1109/GRID.2004.14

[18] "BOINC Status - Boinc Combined," http://boincstats.com/stats/, last Checked: 25-01-2011.

[19] F. Darema, "The SPMD Model: Past, Present and Future," in *Recent Advances in Parallel Virtual Machine and Message Passing Interface*, ser. Lecture Notes in Computer Science, Y. Cotronis and J. Dongarra, Eds. Springer Berlin / Heidelberg, 2001, vol. 2131, pp. 1–1.

[20] W. Cirne, F. Brasileiro, J. Sauvé, N. Andrade, D. Paranhos, E. Santos-neto, R. Medeiros, and F. C. Gr, "Grid computing for bag of tasks applications," in *In Proc. of the 3rd IFIP Conference on E-Commerce, E-Business and EGovernment*, 2003.

[21] D. S. Milojicic, V. Kalogeraki, R. Lukose, K. Nagaraja, J. Pruyne, B. Richard, S. Rollins, and Z. Xu, "Peer-to-peer computing," HP Labs, Tech. Rep., 2002.

[22] A. Iamnitchi, P. Trunfio, J. Ledlie, and F. Schintke, "Peer-to-peer computing," in *Euro-Par 2010 - Parallel Processing*, ser. Lecture Notes in Computer Science, P. D'Ambra, M. Guarracino, and D. Talia, Eds. Springer Berlin / Heidelberg, 2010, vol. 6271, pp. 444–445.

[23] J. A. Pouwelse, P. Garbacki, J. Wang, A. Bakker, J. Yang, A. Iosup, D. H. J. Epema, M. Reinders, M. R. V. Steen, and H. J. Sips, "Tribler: A social-based peer-to-peer system," in *In The 5th International Workshop on Peer-to-Peer Systems (IPTPS'06*, 2006.

[24] J. Cao and F. Liu, "P2PGrid: Integrating P2P Networks into the Grid Environment," in *Grid and Cooperative Computing - GCC 2005*, ser. Lecture Notes in Computer Science, H. Zhuge and G. Fox, Eds. Springer Berlin / Heidelberg, 2005, vol. 3795, pp. 871–883.

[25] Y. Li, T. Dong, X. Zhang, Y. duan Song, and X. Yuan, "Large-scale software unit testing on the grid," in *Granular Computing, 2006 IEEE International Conference on*, May 2006, pp. 596 – 599.

[26] E. de Almeida, G. Sunyé, and P. Valduriez, "Testing architectures for large scale systems," in *High Performance Computing for Computational Science - VECPAR 2008*, ser. Lecture Notes in Computer Science, J. Palma, P. Amestoy, M. Daydé, M. Mattoso, and J. Lopes, Eds. Springer Berlin / Heidelberg, 2008, vol. 5336, pp. 555–566.

[27] R. N. Duarte, W. Cirne, F. Brasileiro, P. Duarte, and D. L. Machado, "GridUnit: Using the Computational Grid to Speed up Software Testing," in *19th Brazilian Symposium on Software Engineering*, 2005.

[28] Y. Jiang, G. Xue, and J. You, "Distributed hash table based peer-to-peer version control system for collaboration," in *Computer Supported Cooperative Work in Design III*, ser. Lecture Notes in Computer Science, W. Shen, J. Luo, Z. Lin, J.-P. Barthès, and Q. Hao, Eds. Springer Berlin / Heidelberg, 2007, vol. 4402, pp. 489–498.

[29] B. de Alwis and J. Sillito, "Why are software projects moving from centralized to decentralized version control systems?" in *Cooperative and Human Aspects on Software Engineering, 2009. CHASE '09. ICSE Workshop on*, May 2009, pp. 36 –39.

[30] A. D. Kshemkalyani and M. Singhal, *Distributed Computing Principles, Algorithms, and Systems.* Cambridge, UK: Cambridge University Press, 2008.

[31] D. P. Vidyarthi, B. K. Sarkar, A. K. Tripathi, and L. T. Yang, *Scheduling in Distributed Computing Systems: Analysis, Design and Models.* New York, NY, USA: Springer, 2009.

[32] L. F. G. Sarmenta, "Volunteer computing," Ph.D. dissertation, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, Cambridge, Massachusetts, USA, June 2001.

[33] D. P. Anderson, J. Cobb, E. Korpela, M. Lebofsky, and D. Werthimer, "SETI@home: an experiment in public-resource computing," *Commun. ACM*, vol. 45, pp. 56–61, November 2002. [Online]. Available: http://doi.acm.org/10.1145/581571.581573

[34] S. Esteves, L. Veiga, and P. Ferreira, "GridP2P: Resource usage in Grids and Peer-to-Peer systems," in *Parallel Distributed Processing, Workshops and Phd Forum (IPDPSW), 2010 IEEE International Symposium on*, 2010, pp. 1 –8.

[35] R. Rodrigues and P. Druschel, "Peer-to-peer systems," *Commun. ACM*, vol. 53, pp. 72–82, October 2010. [Online]. Available: http://doi.acm.org/10.1145/1831407.1831427

[36] N. Therning and L. Bengtsson, "Jalapeno: secentralized grid computing using peer-to-peer technology," in *Proceedings of the 2nd conference on Computing frontiers*. New York, NY, USA: ACM, 2005, pp. 59–65. [Online]. Available: http://doi.acm.org/10.1145/1062261.1062274

[37] L. Gong, "JXTA: a network programming environment," *Internet Computing, IEEE*, vol. 5, no. 3, pp. 88 –95, 2001.

[38] L. Senger, M. de Souza, and D. Foltran, "Towards a peer-to-peer framework for parallel and distributed computing," in *Computer Architecture and High Performance Computing (SBAC-PAD), 2010 22nd International Symposium on*, 2010, pp. 127 –134.

[39] P. Tiburcio and M. Spohn, "Ad hoc grid: An adaptive and self-organizing peer-to-peer computing grid," in *Computer and Information Technology (CIT), 2010 IEEE 10th International Conference on*, 29 2010-july 1 2010, pp. 225 –232.

[40] N. Andrade, L. Costa, G. Germóglio, and W. Cirne, "Peer-to-peer grid computing with the ourgrid community," in *in 23rd Brazilian Symposium on Computer Networks (SBRC 2005) - 4th Special Tools Session*, 2005.

[41] S. Ma, X. Sun, and Z. Guo, "A resource discovery mechanism integrating p2p and grid," in *Computer Science and Information Technology (ICCSIT), 2010 3rd IEEE International Conference on*, vol. 7, 2010, pp. 336 –339.

[42] D. Lá andzaro, J. Marquè ands, and X. Vilajosana, "Flexible resource discovery for decentralized p2p and volunteer computing systems," in *Enabling Technologies: Infrastructures for Collaborative Enterprises (WETICE), 2010 19th IEEE International Workshop on*, 2010, pp. 235 –240.

[43] J. Corbet, G. Kroah-Hartman, and A. McPherson, "Annual report on linux kernel development - linux kernel development," December 2010.

[44] B. O'Sullivan, "Making sense of revision-control systems," *Communications of the ACM*, vol. 52, no. 9, pp. 56–62, September 2009.

[45] N. B. Ruparelia, "The history of version control," *SIGSOFT Software Engineering Notes*, vol. 35, pp. 5–9, January 2010. [Online]. Available: http://doi.acm.org/10.1145/1668862.1668876

[46] S. Chacon, *Pro Git*, D. Parkes, Ed. Apress, 2009.

[47] S. Fomin, "The cathedral or the Bazaar: Version-control centralized or distributed?" in *Software Engineering Conference in Russia (CEE-SECR), 2009 5th Central and Eastern European*, 2009, pp. 259 –265.

[48] P. Mukherjee, A. Kovacevic, and A. Schürr, "Analysis of the benefits the peer-to-peer paradigm brings to distributed agile software development," in *In Proceedings of the the Software Engineering Conference (SE)*, Feb 2008, pp. 72 – 76.

[49] P. Mukherjee, C. Leng, W. Terpstra, and A. Schurr, "Peer-to-peer based version control," in *Parallel and Distributed Systems, 2008. ICPADS '08. 14th IEEE International Conference on*, 2008, pp. 829 –834.

[50] W. Perry, *Effective methods for software testing, third edition*. New York, NY, USA: John Wiley & Sons, Inc., 2006.

[51] T. Hanawa, T. Banzai, H. Koizumi, R. Kanbayashi, T. Imada, and M. Sato, "Large-scale software testing environment using cloud computing technology for dependable parallel and distributed systems," in *Software Testing, Verification, and Validation Workshops (ICSTW), 2010 Third International Conference on*, 2010, pp. 428 –433.

[52] M. Oriol and S. Tassis, "Testing .NET code with YETI," in *Engineering of Complex Computer Systems (ICECCS), 2010 15th IEEE International Conference on*, 2010, pp. 264 –265.

[53] A. Duarte, W. Cirne, F. Brasileiro, and P. Machado, "GridUnit: software testing on the grid," in *ICSE '06: Proceedings of the 28th international conference on Software engineering*. New York, NY, USA: ACM, 2006, pp. 779–782.

[54] I. Foster, "Globus toolkit version 4: Software for service-oriented systems," *Journal of Computer Science and Technology*, vol. 21, pp. 513–520, 2006.

[55] M. Mowbray, "How web community organisation can help grid computing," *International Journal of Web Based Communities*, vol. 3, no. 1, pp. 44–54, 2007.

[56] W. Cirne, F. Brasileiro, N. Andrade, L. Costa, A. Andrade, R. Novaes, and M. Mowbray, "Labs of the world, unite!!!" *Journal of Grid Computing*, vol. 4, no. 3, pp. 225–246, 2006.

[57] D. Thain, T. Tannenbaum, and M. Liyny, "Distributed computing in practice: the condor experience," *Concurrency and Computation: Practice and Experience*, vol. 17, no. 2-4, pp. 323–356, 2005.

[58] D. Talia and P. Trunfio, "A P2P Grid Services-Based Protocol: Design and Evaluation," in *Euro-Par 2004 Parallel Processing*, ser. Lecture Notes in Computer Science, M. Danelutto, M. Vanneschi, and D. Laforenza, Eds. Springer Berlin / Heidelberg, 2004, vol. 3149, pp. 1022–1031.

[59] S. Basu, S. Banerjee, P. Sharma, and S.-J. Lee, "Nodewiz: peer-to-peer resource discovery for grids," in *Cluster Computing and the Grid, 2005. CCGrid 2005. IEEE International Symposium on*, vol. 1, May 2005, pp. 213 – 220 Vol. 1.

[60] "MoSync SDK Documentation," http://www.mosync.com/content/programming-mosync, Last Checked: February 5, 2011.

[61] "The Apache Ant Project," http://ant.apache.org/.

[62] "Xmlrpc specifications," http://www.xmlrpc.com/spec.

[63] "The ruby programming language," http://www.ruby-lang.org/en/.

[64] Z. Xiong, Y. Yang, X. Zhang, M. Zeng, and L. Liu, "A grid resource discovery model using p2p technology," in *Intelligent Information Hiding and Multimedia Signal Processing, 2008. IIHMSP '08 International Conference on*, 2008, pp. 1553 –1556.

[65] E. Meshkova, J. Riihijrvi, M. Petrova, and P. Mähönen, "A survey on resource discovery mechanisms, peer-to-peer and service discovery frameworks," *Computer Networks*, vol. 52, no. 11, pp. 2097 – 2128, 2008. [Online]. Available: http://www.sciencedirect.com/science/article/B6VRG-4S4JYXJ-3/2/c6c11b48f6e01f253797c416292a93b0

[66] A. Di Stefano, G. Morana, and D. Zito, "Qos aware services discovery in a p2p grid environment," in *Pervasive Computing and Applications, 2007. ICPCA 2007. 2nd International Conference on*, 2007, pp. 546 –551.

[67] P. Trunfio, D. Talia, H. Papadakis, P. Fragopoulou, M. Mordacchini, M. Pennanen, K. Popov, V. Vlassov, and S. Haridi, "Peer-to-peer resource discovery in grids: Models and systems," *Future Generation Computer Systems, Elsevier*, vol. 23, no. 7, pp. 864 – 878, 2007. [Online]. Available: http://www.sciencedirect.com/science/article/B6V06-4MMFJ0H-1/2/c45730642fb5e1dd5dbeb364dcad5b19

[68] A. Weinrig and S. Shenker, "Greed is not enough: adaptive load sharing in large heterogeneous systems," in *INFOCOM '88. Networks: Evolution or Revolution, Proceedings. Seventh Annual Joint Conference of the IEEE Computer and Communcations Societies, IEEE*, mar 1988, pp. 986 –994.

[69] "The Active Merchant Project Repository Page," https://github.com/Shopify/.

[70] "jQuery Java Script Library Git Repository," https://github.com/jquery.

[71] "Ruby on Rails Git Repository," https://github.com/rails.

[72] "Xorg Git Repository," http://cgit.freedesktop.org/xorg/.

[73] "Perl Git Repository," http://perl5.git.perl.org/perl.git.

[74] "MoSync SDK Git Repository," https://github.com/MoSync.

[75] "The Linux Kernel Git Repository," http://git.kernel.org/?p=linux/kernel/git/torvalds/linux-2.6.git;a=summary.

[76] "Git - Fast Version Control System," http://git-scm.com/.

[77] "Apache subversion," http://subversion.apache.org/.

[78] "Ubuntu desktop edition," http://www.ubuntu.com/desktop.

TRITA-ICT-EX-2011:164

www.kth.se