# Component Placement and Location in a Dynamic Composition System

by

Behzad Sajed Khosrowshahi

A thesis submitted to
The Faculty of Graduate Studies of
The University of Manitoba
in partial fulfillment of the requirements
of the degree of

Master of Science

Department of Computer Science
The University of Manitoba
Winnipeg, Manitoba, Canada
April 2013

Thesis advisor Author

**Dr. Peter Graham** **Behzad Sajed Khosrowshahi**

**Component Placement and Location in a Dynamic Composition System**

## Abstract

*Using Software-as-a-Service (SaaS), software resides on servers not on user computers. Service Oriented Architecture (SOA) provides the ability to divide an application into parts known as services. This allows enhanced support for distribution, code re-use and code sharing. Combining these ideas, applications can be dynamically composed from components stored at convenient locations in a wide-area network. This benefits users since software installation and upgrades are unnecessary and is also suited to personal devices that may have limited resources (e.g. disk space) to support conventional installed software. I have designed, prototyped, and evaluated component-placement and location algorithms for a system that combines ideas from SaaS and SOA to support on-demand composition of applications that run on user devices from storage sites in the network. These algorithms support mobility and are scalable and reliable. I have implemented a Java prototype and a simulation system that I used to assess my systems behaviour.*

# Contents

Abstract . . . . . . . . . . ii
Table of Contents . . . . . . . . . . v
List of Figures . . . . . . . . . . vi
List of Tables . . . . . . . . . . viii
Acknowledgments . . . . . . . . . . ix
Dedication . . . . . . . . . . x

**1 Introduction** **1**

**2 Related Work** **5**
2.1 Monolithic vs. Non-Monolithic Software . . . . . . . . . . 5
2.1.1 Software-as-a-Service (SaaS) . . . . . . . . . . 5
2.1.2 Service Oriented Architecture (SOA) . . . . . . . . . . 6
2.1.3 Other Concepts Closely Related to SaaS and SOA . . . . . . . 7
2.1.4 Component-based Software . . . . . . . . . . 7
2.2 Composition . . . . . . . . . . 8
2.2.1 Service Composition . . . . . . . . . . 8
2.3 Distributed Systems . . . . . . . . . . 9
2.4 Placement Algorithms . . . . . . . . . . 10
2.5 Location Algorithms . . . . . . . . . . 13
2.5.1 Service Lookup . . . . . . . . . . 13
LDAP . . . . . . . . . . 13
MDS . . . . . . . . . . 14
DNS . . . . . . . . . . 14
INS . . . . . . . . . . 16
2.6 Network Modelling and Simulation . . . . . . . . . . 16
2.6.1 Network Simulators . . . . . . . . . . 16
Network Generator Tools . . . . . . . . . . 17
Generation Methods . . . . . . . . . . 18
2.6.2 Simulation Systems and Software . . . . . . . . . . 20

**3 Concise Problem Statement 22**

**4 Solution Strategy and Prototype Implementation 24**
4.1 Introduction . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 24
4.2 Components . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 25
4.3 System Entities . . . . . . . . . . . . . . . . . . . . . . . . . . . 27
4.3.1 Clients . . . . . . . . . . . . . . . . . . . . . . . . . . . . 27
Searching for a component and downloading it . . . . . . . . . 27
Uploading and registering a new component . . . . . . . . . . 28
4.3.2 Cache Location Servers . . . . . . . . . . . . . . . . . . . . 28
4.3.3 Master Location Servers . . . . . . . . . . . . . . . . . . . 29
4.3.4 Component Storage Server . . . . . . . . . . . . . . . . . . 30
4.4 Use of Hash Function in My Prototype . . . . . . . . . . . . . . . 30
4.5 Redundancy . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 32
4.6 Location Awareness . . . . . . . . . . . . . . . . . . . . . . . . . 35
4.7 The Component Download Process . . . . . . . . . . . . . . . . . 36

**5 Assessment 41**
5.1 Introduction . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 41
5.2 Simulation . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 41
BRITE Configuration . . . . . . . . . . . . . . . . . . . . . . 42
BRITE Output . . . . . . . . . . . . . . . . . . . . . . . . . . 44
Use of BRITE in my Thesis . . . . . . . . . . . . . . . . . . . 48
5.2.1 Use of SSJ in my Thesis . . . . . . . . . . . . . . . . . . . 48
5.3 Results from the Simulation Work . . . . . . . . . . . . . . . . . 52
5.3.1 Scenarios . . . . . . . . . . . . . . . . . . . . . . . . . . . 52
Simulation Constants . . . . . . . . . . . . . . . . . . . . . . 52
Simulation Default Values . . . . . . . . . . . . . . . . . . . . 55
5.3.2 Scenario 1 - Increasing the Number of Components . . . . . . 57
Goal . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 57
Expectation . . . . . . . . . . . . . . . . . . . . . . . . . . . 57
Result . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 57
5.3.3 Scenario 2 - Increasing the Number of Clients . . . . . . . . . 58
Goal . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 58
Expectation . . . . . . . . . . . . . . . . . . . . . . . . . . . 59
Result . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 59
5.3.4 Scenario 3 - Increasing the Number of Operations . . . . . . . 60
Goal . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 60
Expectation . . . . . . . . . . . . . . . . . . . . . . . . . . . 61
Result . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 61
5.3.5 Scenario 4 - Increasing the Number of Component Update Events 62
Goal . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 62

Expectation . . . . . . . . . . . . . . . . . . . . . . . . . 63
Result . . . . . . . . . . . . . . . . . . . . . . . . . . . 63
5.3.6 Scenario 5 - Increasing the Probability of Cache Location Server Failures . . . . . . . . . . . . . . . . . . . . . . . . . 65
Goal . . . . . . . . . . . . . . . . . . . . . . . . . . . . 65
Expectation . . . . . . . . . . . . . . . . . . . . . . . . . 65
Result . . . . . . . . . . . . . . . . . . . . . . . . . . . 65
5.3.7 Scenario 6 - Increasing the Probability of Master Location Server Failures . . . . . . . . . . . . . . . . . . . . . . . . . 68
Goal . . . . . . . . . . . . . . . . . . . . . . . . . . . . 68
Expectation . . . . . . . . . . . . . . . . . . . . . . . . . 68
Result . . . . . . . . . . . . . . . . . . . . . . . . . . . 68
5.3.8 Scenario 7 - Increasing the Probability of Component Storage Server Failures . . . . . . . . . . . . . . . . . . . . . . 71
Goal . . . . . . . . . . . . . . . . . . . . . . . . . . . . 71
Expectation . . . . . . . . . . . . . . . . . . . . . . . . . 71
Result . . . . . . . . . . . . . . . . . . . . . . . . . . . 72
5.3.9 Scenario 8 - Increasing User Movement . . . . . . . . . . . . 75
Goal . . . . . . . . . . . . . . . . . . . . . . . . . . . . 75
Expectation . . . . . . . . . . . . . . . . . . . . . . . . . 75
Result . . . . . . . . . . . . . . . . . . . . . . . . . . . 76
5.3.10 Scenario 9 - Varying the Number of Clients in Dense Population Areas . . . . . . . . . . . . . . . . . . . . . . . . . . 77
Goal . . . . . . . . . . . . . . . . . . . . . . . . . . . . 77
Expectation . . . . . . . . . . . . . . . . . . . . . . . . . 78
Result . . . . . . . . . . . . . . . . . . . . . . . . . . . 78
5.3.11 Scenario 10 - Varying Component Popularity . . . . . . . . . 79
Goal . . . . . . . . . . . . . . . . . . . . . . . . . . . . 79
Expectation . . . . . . . . . . . . . . . . . . . . . . . . . 79
Result . . . . . . . . . . . . . . . . . . . . . . . . . . . 80
5.3.12 Scenario 11 - Varying the Number of Component Storage Servers 81
Goal . . . . . . . . . . . . . . . . . . . . . . . . . . . . 81
Expectation . . . . . . . . . . . . . . . . . . . . . . . . . 82
Result . . . . . . . . . . . . . . . . . . . . . . . . . . . 82
5.3.13 Scenario 12 - Varying the Number of Cache Location Servers . 85
Goal . . . . . . . . . . . . . . . . . . . . . . . . . . . . 85
Expectation . . . . . . . . . . . . . . . . . . . . . . . . . 85
Result . . . . . . . . . . . . . . . . . . . . . . . . . . . 86

**6 Conclusion** **89**
6.1 Summary and Contributions . . . . . . . . . . . . . . . . . . 89
6.2 Future Work . . . . . . . . . . . . . . . . . . . . . . . . . . 90

**Bibliography** **95**

# List of Figures

2.1 Example of a Distributed System . . . . . . . . . . . . . . . . . . . . 10
2.2 Replication in Distributed System [WPS+00] . . . . . . . . . . . . . . 12
2.3 DNS structure for www.cs.umanitoba.ca . . . . . . . . . . . . . . . . . 15

4.1 Example of the Solution Strategy [SKG09] . . . . . . . . . . . . . . . 25
4.2 Code of Time_100.java . . . . . . . . . . . . . . . . . . . . . . . . . 26
4.3 Code of getServerID Method . . . . . . . . . . . . . . . . . . . . . . 32
4.4 Example of Grouping Five Master Location Servers . . . . . . . . . . 34
4.5 Code of getServerIDSequence method . . . . . . . . . . . . . . . . . 35
4.6 Code of calcDistance Method . . . . . . . . . . . . . . . . . . . . . . 36
4.7 Flow chart of how a client downloads a new component . . . . . . . . 39
4.8 Communication between different entities . . . . . . . . . . . . . . . 40

5.1 Structure of BRITE models adapted from [MLMB01] . . . . . . . . . 43
5.2 Example of a BRITE output . . . . . . . . . . . . . . . . . . . . . . . 47
5.3 Scenario 1 - Effect of Increasing the Number of Components on Average Download Time . . . . . . . . . . . . . . . . . . . . . . . . . . . 58
5.4 Scenario 2 - Effect of Increasing the Number of Clients on Average Download Time . . . . . . . . . . . . . . . . . . . . . . . . . . . 60
5.5 Scenario 3 - Effect of Increasing the Number of Operations on Average Download Time . . . . . . . . . . . . . . . . . . . . . . . . . . . 62
5.6 Scenario 4 - Effect of Increasing the Number of Component Update Events on Average Download Time . . . . . . . . . . . . . . . . . . . 64
5.7 Scenario 5 - Effect of Increasing the Probability of Cache Location Server Failures on Average Download Time when 80% of Clients were Placed in Dense Population Areas . . . . . . . . . . . . . . . . . . . . 66
5.8 Scenario 5 - Effect of Increasing the Probability of Cache Location Server Failures on Average Download Time when all Clients were Placed Uniform Randomly . . . . . . . . . . . . . . . . . . . . . . . . 67

5.9 Scenario 6 - Effect of Increasing the Probability of Master Location Server Failures on Average Download Time when 80% of Clients were Placed in Dense Population Areas . . . . . . . . . . . . . . . . . . . . 69
5.10 Scenario 6 - Effect of Increasing the Probability of Master Location Server Failures on Average Download Time when all Clients were Placed Uniform Randomly . . . . . . . . . . . . . . . . . . . . . . . . . 70
5.11 Scenario 7 - Effect of Increasing the Probability of Component Storage Server Failures on Average Download Time when 80% of the Clients were Placed in Dense Population Areas . . . . . . . . . . . . . . . . . 72
5.12 Scenario 7 - Effect of Increasing the Probability of Component Storage Server Failures on Average Download Time when All Clients were Placed Uniform Randomly . . . . . . . . . . . . . . . . . . . . . . . . . 74
5.13 Scenario 7- Effect of Increasing of Number of Component Storage Server Failure on Number of Completed and Outstanding Search Operations . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 75
5.14 Scenario 8 - Effect of Increasing User Movement on Average Download Time . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 77
5.15 Scenario 9 - Effect of Varying Number of Clients in Dense Population Areas on Average Download Time . . . . . . . . . . . . . . . . . . . . 79
5.16 Scenario 10 - Effect of Varying Component Popularity on Average Download Time . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 81
5.17 Scenario 11 - Effect of Varying the Number of Component Storage Servers on the Average Download Time with 3 Components . . . . . 83
5.18 Scenario 11 - Effect of Varying the Number of Component Storage Servers on the Average Download Time with 100 Components . . . . 84
5.19 Scenario 12 - Effect of Varying the Number of Cache Location Servers on Average Download Time with 3 Components . . . . . . . . . . . . 87
5.20 Scenario 12 - Effect of Varying the Number of Cache Location Servers on Average Download Time with 100 Components . . . . . . . . . . . 88

# List of Tables

5.1 Flat Topology (AS Only or Router Only) Parameters [MLMB01] . . . 45
5.2 Top-down Hierarchical Topology Parameters [MLMB01] . . . . . . . . 46
5.3 Nodes Information in BRITE's Output File [MLMB01] . . . . . . . . 46
5.4 Edges Information in BRITE's Output File [MLMB01] . . . . . . . . 46
5.5 Simulation Configuration Values . . . . . . . . . . . . . . . . . . . . . 56
5.6 Scenario 1 - Effect of the Increasing the Number of Components on Average Download Time . . . . . . . . . . . . . . . . . . . . . . . . . 57
5.7 Scenario 2 - Effect of Increasing the Number of Clients on Average Download Time . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 59
5.8 Scenario 3 - Effect of Increasing the Number of Operations on Average Download Time . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 61
5.9 Scenario 4 - Effect of Increasing the Number of Component Update Events on Average Download Time . . . . . . . . . . . . . . . . . . . 64
5.10 Scenario 5 - Effect of Increasing the Probability of Cache Location Server Failures on Average Download Time when 80% of Clients Placed in Dense Population Areas . . . . . . . . . . . . . . . . . . . . . . . . 66
5.11 Scenario 5 - Effect of Increasing the Probability of Cache Location Server Failures on Average Download Time when all Clients Placed Uniform Randomly . . . . . . . . . . . . . . . . . . . . . . . . . . . . 67
5.12 Scenario 6 - Effect of Increasing the Probability of Master Location Server Failures on Average Download Time when 80% of Clients were Placed in Dense Population Areas . . . . . . . . . . . . . . . . . . . 69
5.13 Scenario 6 - Effect of Increasing the Probability of Master Location Server Failures on Average Download Time when all Clients were Placed Uniform Randomly . . . . . . . . . . . . . . . . . . . . . . . . 70
5.14 Scenario 7 - Effect of Increasing the Probability of Component Storage Server Failures on Average Download Time when 80% of the Clients were Placed in Dense Population Areas . . . . . . . . . . . . . . . . 72

5.15 Scenario 7 - Effect of Increasing the Probability of Component Storage Server Failures on Average Download Time when All Clients were Placed Uniform Randomly . . . . . . . . . . . . . . . . . . . . . . . . . 73
5.16 Scenario 8 - Effect of Increasing User Movement on Average Download Time . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 76
5.17 Scenario 9 - Effect of Varying Number of Clients in Dense Population Areas on Average Download Time . . . . . . . . . . . . . . . . . . . . 78
5.18 Scenario 10 - Effect of Varying Component Popularity on Average Download Time . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 80
5.19 Scenario 11 - Effect of Varying the Number of Component Storage Servers on the Average Download Time with 3 Components . . . . . 82
5.20 Scenario 11 - Effect of Varying the Number of Component Storage Servers on the Average Download Time with 100 Components . . . . 83
5.21 Scenario 12 - Effect of Varying the Number of Cache Location Servers on Average Download Time with 3 Components . . . . . . . . . . . . 86
5.22 Scenario 12 - Effect of Varying the Number of Cache Location Servers on Average Download Time with 100 Components . . . . . . . . . . . 87

# Acknowledgments

It is with immense gratitude that I acknowledge the inspiration, motivation, and support of my advisor, Professor Peter Graham, throughout this project. In addition to many aspects of computer science, Professor Graham has taught me how to exercise patience while being persistent and how to respect and embrace diversity of our world.

I dedicate this thesis to my loving parents, Mahnaz and Hamid without whose encouragement and push for tenacity this dissertation work would have been impossible.

In addition, I would like to thank my uncle, Dr. Abbas Farazdel, for many stimulating discussions about creative applications of computer science in everyday life and for his continuous support and encouragement since I was in middle school.

*To Father, Mother and my lovely Sister*

# Chapter 1

# Introduction

Service Oriented Architectures (SOAs) [Car08, HKW$^+$08] allow an application to be composed of many distributed **components** while software-as-a-service (SaaS) [Cus10, Nit09] system design allows applications to exist "in the network" rather than locally on computers or personal devices. Using SOA principles, SaaS users should be able to access their applications as components stored separately and remotely. This will allow user devices such as laptops, smartphones, etc to operate directly off the network with only core operating system code stored on the device and applications loaded in parts, as needed, from the network and "composed"/assembled on the fly to meet specific user requirements. To be useful, these components (i.e. parts of an application)[1]need to be efficiently and reliably accessible from anywhere at anytime.

Component placement is the problem of deciding where to place copies/replicas of components so they will be readily and quickly available when needed. Component location is the problem of finding a needed replica of a component for efficient access

[1]For example Word might be decomposed into 10's or 100's of components based on function (e.g. a formula editor might be one component) and users not load components they are not using.

based on the current location of an accessing device.

There are different benefits of using such a “non-monolithic” model of applications. First of all, for devices with limited resources (memory, CPU) (e.g. a smartphone) it is not a good idea to install the whole application on them. Instead, users can just startup the core of the application and when they need another part of the application, just download it and use it and then, later, when they do not need that part, it can be deleted from the device. Another benefit of this model is cost efficiency. For example, if a user needs Adobe Photoshop to edit a single photo, he does not need to buy the whole application and install it on his local computer even though he needs to use the application for only a short period of time. The non-monolithic approach gives the ability to users to use and pay for the application based on their needs. Therefore, users can save money and it is possible for vendors to sell the use of their products to a larger number of clients. This will also tend to reduce copyright violations. Another significant benefit of using a non-monolithic application model is that applications do not need to be installed and maintained on user devices as automated composition of the needed application components precludes this. The result is a simpler and thus more attractive environment for the user and probably also enhanced security since old, insecure software versions will no longer exist. (Security patching will be done centrally and reliably by the software provider). I will show that such a system can be both efficient and effective.

In this thesis I describe a system (focusing on component placement and location) that I have designed, implemented and assessed that provides the basis for dynamic composition of software from components found on the Internet in component storage servers. My solution strategy is focused on the use of component replication (in multiple storage servers) to ensure that the needed components are reliably accessible to clients as quickly as possible from anywhere at anytime. It is these replicas that need to be placed and quickly located. To do this, I created different types of servers in my system. The first type is the cache location servers to which clients send their requests to access components. They store the local information of the components' locations. There are master location servers which store the primary information about the location of all components. The component storage servers actually store replicas of components. Cache location servers and component storage servers are accessed by clients. Clients upload new components they may have created (i.e. software providers) and send their requests to download needed components (i.e. software users). My algorithms for component placement and lookup are location sensitive to ensure efficiency based on locality of access.

To assess any system and prototype, we need to use a prototype system or simulation software. I have chosen to use both. By prototyping, I have demonstrated algorithm correctness, and by applying simulation I was able to assess large-scale system behaviour. In my prototype I built all the aforementioned parts of the system using Java and tested the prototype by distributing some simple Java components (e.g. a component to retrieve the current time) across Linux machines in the Computer Science Department. Components were placed by my system on certain machines and then downloaded to others upon request. The first step of my simulation was creating a simulated network topology. I used BRITE to create a network topology that reflects the large-scale characteristics of the Internet. I chose SSJ to use to develop my simulation system. I then designed 12 different scenarios to assess the large-scale behaviour of my system under various conditions. I report the results of these simulations in this thesis.

The rest of this document is organized as follows. Chapter 2 presents the work related to my thesis including distributed systems generally, placement and location algorithms, software composition and a brief overview of network modeling and simulation. Chapter 3 concisely describes the problem to be solved. I then describe my solution strategy in Chapter 4. In Chapter 5, I describe the assessment of system, focusing on the 12 simulation scenarios mentioned earlier. Finally, Chapter 6 provides my conclusions and proposes some areas for future work.

# Chapter 2

# Related Work

## 2.1 Monolithic vs. Non-Monolithic Software

The traditional "monolithic" software model where an application is installed in its entirety on each user machine introduces a number of problems including, high resource demands (on the client machine), high cost, licensing issues and difficulty in keeping software versions current. To address some of these issues, a number of alternative models have been proposed and implemented. Those of direct relevance to this thesis will now be briefly reviewed.

### 2.1.1 Software-as-a-Service (SaaS)

In recent years, many companies, including IBM, Google, Dell, and HP, have pushed the SaaS concept and are delivering SaaS services in the form of cloud computing and other hosted delivery services [KKLL09].

The general idea behind SaaS is that customers can access applications and data using, for example, a web browser on a notebook or other personal device without investing in and owning more extensive hardware and software infrastructure. The SaaS provider company (or *vendor*) is responsible for delivery, management, maintenance, and security of the applications, data, and the infrastructure on which the applications run. The delivery is usually based on a Service Level Agreement (SLA) with the client company (or *tenant*). SaaS has become popular because it is a way to save on the cost of owning and maintaining applications without, in most cases, compromising functionality. A simple example of SaaS is web-based email systems delivered over the Internet such as Gmail and Yahoo mail or document management applications like Google Apps [Ciu09].

### 2.1.2 Service Oriented Architecture (SOA)

Hau et al. [HEHB08] identify two building blocks underlying SOA. The first is architecture and the second is services. A service is any functional process. Thus, in SOA, an application is divided into different component services according to some architectural (business-based) model. The component services are independent from each other but support connections allowing them to be combined. Thus, multiple services possibly from different providers can be composed to build applications.

Hau et al. [HEHB08] also identify several benefits of SOA. The first is greater agility allowing a vendor company to adapt services to new requirements faster. To change a service in an SOA application is easier and faster than traditional applications because one may only need to change just one part of the loosely-coupled application. Another benefit is decreased complexity because, in an SOA-organized architecture, each component is implemented separately. Another benefit is reusability. Each component can be reused in different applications. Finally, interoperability between service components is enhanced since each service component has well defined inputs and outputs intended for composition and programmers need be concerned only about the middleware that connects the components.

### 2.1.3 Other Concepts Closely Related to SaaS and SOA

Cloud Computing [KKLL09] is a form of distributed computing in which real-time scalable computing resources are provided to clients over the Internet (from the "cloud") as a service, rather than clients having their own local hardware and software resources. As such, Cloud Computing can be thought of as one possible implementation strategy for SaaS.

An Application Service Provider (ASP) [CS01] is a third-party entity that hosts software-based services on a wide-area network (e.g. the Internet). SaaS-delivered applications differ from ASP applications in multiple ways. ASP applications are traditionally single-tenant, are hosted, managed, monitored by a third party and are not customized, while SaaS applications are commonly multi-tenant, hosted by vendors and users can customize the SaaS applications. Otherwise, SaaS and ASP are very similar.

### 2.1.4 Component-based Software

Component-based software serves as a basis for SOA but can be used in different environments and at various scales from the small (e.g. shared libraries in

Unix [Lev99]) through to the large (e.g. distributed object systems such as Corba [TB01]). Each component-based application is comprised of different components. For example in a hypothetical component-based implementation of Microsoft PowerPoint, there might be components for animations, editing slide shows, etc. In component-based software, first a core component of the software is loaded onto a device and later, when a user needs another feature of the application, the necessary components are added ("composed" with the core). When components are no longer needed, they may also be removed to save device resources (e.g. memory).

## 2.2 Composition

### 2.2.1 Service Composition

Wu et al. [WDLW09] describe the main task of service composition as being to allow existing components to connect to each other. There are two parts to service composition: *composition synthesis* and *orchestration*. Composition synthesis refers to the procedure that components use to reply to service requests to enable their assembly. Orchestration refers to runtime coordination between components in terms of both data and control flow.

Ibrahim and Le Mouel [ILM09] describe a typical instance of service composition middleware that is used to connect multiple services. Such services can be static or dynamic. Static services run when the main software starts up, but dynamic services can be added to the main software after startup. Using dynamic services, the main software can be leaner and hence run faster (and/or on less capable devices) but the user can still invoke dynamic services, as needed, that will be dynamically located and attached prior to invocation and which may be later terminated and detached.

## 2.3 Distributed Systems

A distributed system [CD88] is a type of software architecture that includes different parts running across various networked machines. The whole software, for the most part, operates as if it were a single monolithic piece of code running on a single computer.

There are two primary benefits of a distributed system over a centralized system. The first is scalability, which means that growing the system is very easy and fast. The other benefit is redundancy. If a machine fails in the distributed system there may be other machines that have the needed capabilities and can provide them to the user. Therefore, if the distributed system is designed properly, the user may not need to be aware that one machine is down.

The machines in a distributed system can have different operating systems (e.g., Unix, Windows, etc.) and also can connect to each other using different network protocols (e.g., TCP/IP, Bluetooth, etc.). Figure 2.1 shows an example of a distributed system.

The most common type of distributed system is client/server [DC]. Modern client/server systems normally have three tiers. The first tier is the “Presentation” tier. Using the presentation tier, a user creates requests and sends them to the second tier, which is the “Business Logic” tier. The business logic tier processes users requests and connects to the “Data/Resource” tier to retrieve data and then creates and sends results back to the user for display using the presentation tier.

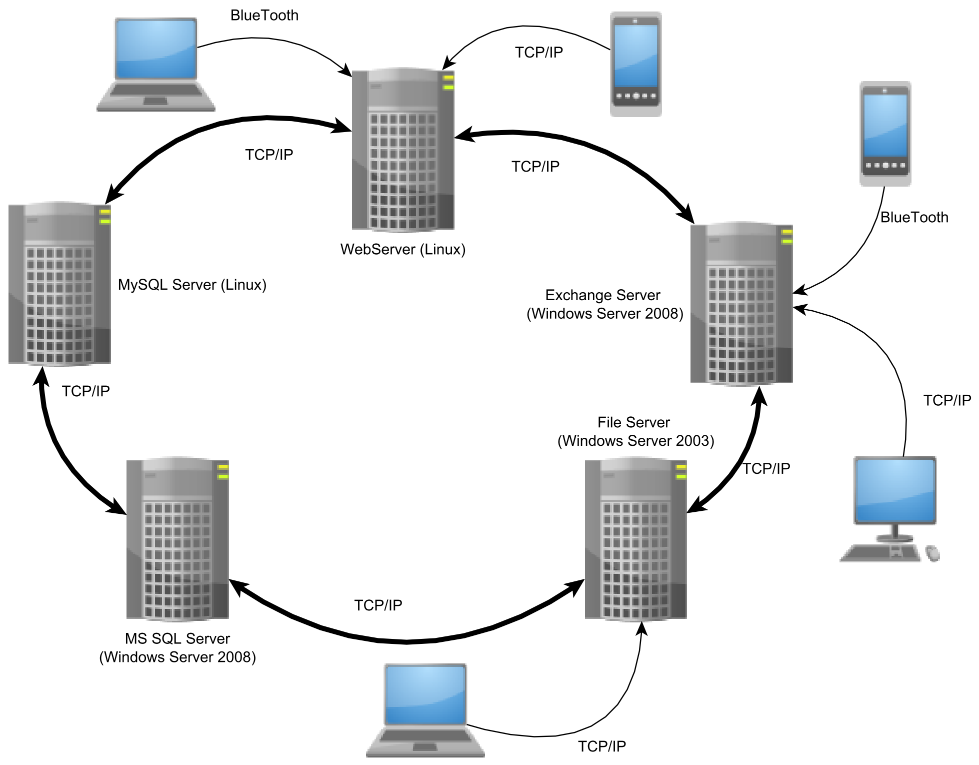


Figure 2.1: Example of a Distributed System

## 2.4 Placement Algorithms

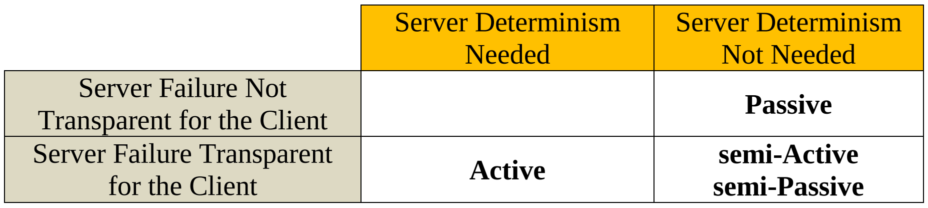
In addition to scalability and redundancy, another potential benefit of distributed systems is providing high availability, which means the system can minimize service interruption. High availability should not be confused with fault tolerance. The latter means the system has no service interruption. Fault tolerance relies on hardware redundancy to replace the function of the failed components and it typically costs significantly more than high availability.

High availability is one of the key goals in distributed systems design. As the name suggests, high availability helps ensure that a software system will be available for use a high percentage of the time. One way to achieve high availability is through replication of the needed information (data and application). If access to the needed information fails, another replicated copy can be accessed and the service will continue with minor delay. Assuming that replication is to be done a key issue is where to *place* the replicas.

Oki and Liskov [OL88] advocate a replication scheme based on a primary-backup algorithm. The application runs on the "primary copy" of data while informing all the replicas what it is doing. In the case that the primary copy fails, one of the backup copies will take over and becomes the new primary and the service will continue.

Wiesmann et al. [WPS$^+$00] compare four different models of replication for both databases and distributed systems. The first is "Active Replication", which is a non-centralized replication technique where the client does not connect to a specific server. The client sends its requests to all replicas. Advantages of this technique include simplicity of implementation (just need to copy code on all servers) and failure transparency for the client (if the system has a failed replica, client does not need to know about it).A disadvantage is the overhead associated with the multiple request messages that are sent to all replicas. Another technique is "Passive Replication". In this technique the client sends its request to a specific server and then updates other replicas for backup (for example, it downloads a component and then uploads this component to other replicas). This technique uses little processing time as all servers do not need to process the client's request. The third model is "Semi-Active

Replication". This technique is very close to active replication, but there is a defined leader server and multiple follower servers. The leader server informs the followers of new update using "View Synchronous Broadcast" [WPS$^+$00][1]. The last model is "Semi-Passive Replication". This model is similar to the passive replication technique but can be implemented asynchronously without needing any views[2]. A client does not need to wait for the response of a failed replica using time out. Figure 2.2 shows the key differences between these models.



| | Server Determinism Needed | Server Determinism Not Needed |
|---|---|---|
| Server Failure Not Transparent for the Client | | **Passive** |
| Server Failure Transparent for the Client | **Active** | **semi-Active** **semi-Passive** |

Figure 2.2: Replication in Distributed System [WPS$^+$00]

Guerraoui and Schiper [GS96] present a comprehensive survey of replication techniques to provide fault-tolerance. They elaborate on the two main classes of replication methods, namely, primary-backup replication and active replication. They define an abstract general model to discuss a variety of aspects of using replication in achieving fault-tolerance.

[1]View Synchronous Broadcast is a communication way to monitor modification of members

[2]Each view defines the composition of the group at some time t, i.e. the members of the group that are perceived as being correct at time t. [WPS$^+$00]

## 2.5 Location Algorithms

As entities of a distributed system (data, servers, etc.) may be located at different places in a network, one of the most important aspects of a distributed system is its location algorithm to find required entities.

### 2.5.1 Service Lookup

Repositories are commonly used to store code and data for access when they are needed. There are many different repository/directory systems with corresponding lookup methods used in various existing distributed systems. A small sample of such systems is now briefly reviewed.

#### LDAP

The Lightweight Directory Access Protocol (LDAP) [KV04] is an application protocol that supports queries on a directory to find an "entity". Each directory entry in LDAP consists of a name, a type and a value. The name of the entry is unique and sometimes is referred to as a "distinguishing" name. The type of an entry indicates the type of value that the entry can accept. The value of an entry is the data that is associated with the entry. Thus, LDAP can store different types of data in a single directory.

The architecture of LDAP is client/server. There is an LDAP server that provides lookup services and stores the data. Clients send requests that consist of an operation (such as "query") and a value, using a protocol agreed to by the clients and the server.

LDAP directories can be categorized into different types based on their location.

If a directory is in a local network, it is a local directory; otherwise it is a global directory. LDAP directories can also be centralized, having all data stored in one directory, or they can be distributed.

### MDS

Another lookup system is the Monitoring and Discovery System (MDS) used in the Globus grid computing toolkit [SPM$^+$06]. With MDS a user has the ability to discover a service and then monitor the repository that stores the service in a "virtual organization" (VO) consisting of users and resources that are related by a common policy set. There are two functions provided by MDS. The first is an indexing function that involves data from different service repositories and supports querying the data. If this function fails, clients cannot find new services in repositories but clients can still use any cached lookup data. Clients can also ask about the current status of the resources directly. The other function provided by MDS is a trigger function that takes an action based on some condition in the repository/directory allowing clients to react asynchronously (i.e. without the need for constant checking) to changes in repository data.

### DNS

The Domain Name System (DNS) [MD95, Moc87a, Moc87b] is a wide-area distributed system that looks up IP addresses (e.g. 130.179.28.33) given domain names (e.g. gold.cs.umanitoba.ca). Each server in DNS contains databases that store domain names and their associated IP addresses. The namespace is hierarchical. The root of the hierarchy is divided into different main "zones"[3] (e.g., .com, .edu, .net). This division continues to multiple levels of sub-domains. For example in the domain "www.cs.umanitoba.ca" "cs" is a sub-domain of "umanitoba" and "umanitoba" is a child of "ca" which is a child of the root of the DNS. Figure 2.3 shows this as an example of the DNS hierarchical naming structure.

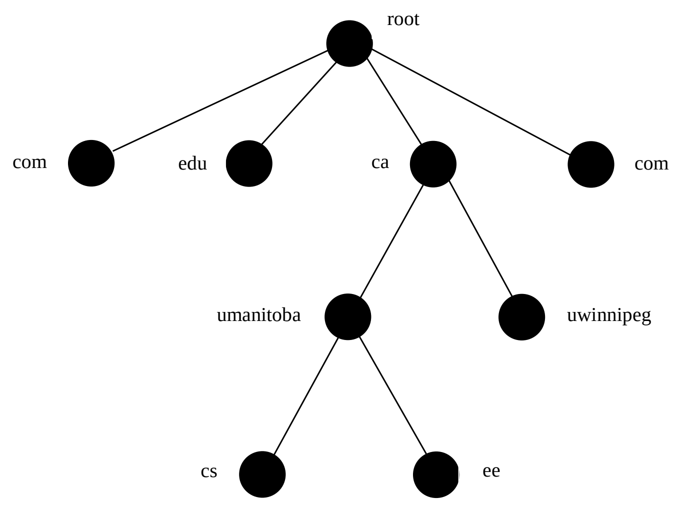


Figure 2.3: DNS structure for www.cs.umanitoba.ca

The databases of domain name to IP address mappings are collectively distributed across three types of servers in DNS. A master server contains information for one or more zones. A slave server replicates domain name information from its master to provide enhanced reliability. Cache servers distributed throughout the Internet save domain name information locally (typically from multiple zones) for client machines after they request and receive the information from a master (or a slave) server. DNS provides a very efficient and reliable wide area lookup capability.

[3]A DNS zone corresponds to a specific part of the global DNS namespace.

**INS**

The Intentional Naming System (INS) [AWSBL00] provides a unique way to name and look up information. In the INS, the names of entries are “expressive” which means each name is a combination of different descriptive properties instead of just a text string. A request in INS is made using a description of what is wanted in terms of one or more properties, not using a specific name as in DNS. For example a client might ask for information about a component that can “do animation in Microsoft PowerPoint”.

## 2.6 Network Modelling and Simulation

Assessment of distributed systems almost always includes some form of simulation based on an abstracted model of the network being used (commonly the Internet). This section briefly reviews some common tools needed to do such simulations.

### 2.6.1 Network Simulators

To assess any distributed system, network simulation at global scale is commonly required as large-scale deployments are impractical. A key characteristic of any simulation is accuracy within the constraints of the associated model. For distributed systems, a key aspect of this is the structure of the underlying network. Many tools exist in support of network simulation. A subset are now discussed.

### Network Generator Tools

There are two ways to test a system (e.g., routing protocol, network application, etc.) in a network environment. First, we can test in an actual new/existing network. In this way the testing process is limited by scalability (e.g., number of nodes, etc.) and network properties (e.g., bandwidth, node location, etc.). This method is not sufficient to assess all possible aspects of a system's behaviour. If we want to create a new network just for testing, it will be expensive and for any modification of the network or its property parameters, the network structure must be changed and this will add to the cost. The second way to test a system in a network environment is to use a network topology generator tool and then simulation. The advantages of using network generator tools and simulation are to support scalability assessment, and efficiency. There is much flexibility using such tools. For example, changing network properties in a tool is trivial since by changing a network parameter you can quickly and easily create a new network topology.

There are different topology models that are commonly used for modelling the Internet and other real-world network structures with the most popular ones being Waxman [Wax88] and the hierarchical domain model [CDZ97]. Each of these models uses a different structure to generate the topology [MLMB01]. The Waxman model is based on placing routing nodes at random locations on a plane (representing the earth's surface) and then adding edges between these nodes to reflect connectivity of nodes. The hierarchical domain model reflects common Internet structure. It first creates Autonomous Systems (ASs) that reflect wide-area network infrastructure owned by a single organization (and connected to other ASs at what are sometimes referred to as "peering points") and then adds routing nodes to the ASs.

There are different network generator tools available with different features. [MLMB01] says that a generator tool is universal if it has certain specific features. The first is *Representativeness*, which means the generated topology should be an accurate reflection of the characteristics of the real topology of interest. *Inclusiveness* says that the tool has the strength of many generation models. *Flexibility* means that changing the topology characteristics should be feasible and easy. *Efficiency* implies that running the tool and generating a topology should use the CPU and the memory efficiently. *Extensibility* means the user should be able to change a model's features or add new models to the tool. *Interoperability* indicates that the tool should generate output that can be used in another tool. *Robustness* requires detection features to find errors in the topology. Lastly, *User-friendliness*, implies that using the tool should be easy and intuitive for the user. All these features give researchers the ability to use the existing tool or just modify it for their new requirements without developing and implementing a new model. BRITE (the Boston university Representative Internet Topology gEnerator) [MLMB01] is a universal generator tool.

### Generation Methods

In any specific network generator, there are two key factors, the placement method that is used to distribute nodes, and the connection function which gives the probability of a new node connecting to existing node(s).

There are three common generation strategies [MMB00]. The first is the *random* method. In this method, nodes are placed in a plane uniform randomly. The probability that two nodes connect to each other is based on Waxman's probability function shown in Equation (2.1) [MMB00] where $\alpha > 0$ , $0 < \beta \leq 1$, $d$ is the distance between two nodes and $L$ is the longest distance between any two nodes. If $\alpha$ is a large number, $\beta$ is close to 0 and $d$ is a small value, which means if two nodes are very close to each other, then the probability of these two nodes connecting to each other is increased.

$$P(u,v) = \alpha e^{\frac{-d}{\beta L}} \tag{2.1}$$

The second method is the *regular method*. In this method nodes are not at random locations but are located on the vertices of a regular grid. The connection between two nodes is determined by the grid connectivity, which means each node is connected to two or four (in 2D) adjacent nodes.

The third method is the *hierarchical method*. This method has two types of connectivity graphs: higher-level and lower-level. In GT-ITM [CDZ97] the nodes are placed randomly in the lower-level graph and then these lower-level graphs are connected to each other by a higher level structure. The connection between nodes in the lower level graphs are random but the connection at the higher level is structured (i.e. grid). Another type of hierarchical method, is the *Transit-Stub* method [CDZ97]. In this method, first a *Transit domain* graph is randomly generated and then nodes are placed randomly inside each domain which is called a *Stub domain*. A topology generated by the Transit-Stub method is the closest to the Internet topology.

Both Waxman and Transit-Stub methods create their nodes and then add the edges between them. Therefore, the network cannot be grown [MMB00]. Barabasi and Albert [AR99] believe generated network topologies should be able to grow reflecting the fact that real networks do so. There are two aspects to growing networks. First is *incremental growth*, which means a network is open to grow and nodes can be added to it based on a timeline reflecting activities such as a new smartphone or a laptop connecting to the Internet. The other is *Preferential connectivity* that reflects the increased probability that a new node will be connected to a popular node (rather than to other nodes).

### 2.6.2 Simulation Systems and Software

There are different types of simulation systems. In some of them, the user must implement the whole simulation from scratch and use a programming language such as C++. The benefit of this type is the performance of simulation, which is customized to the problem at hand and thus the simulations use less resources and run faster. The downside is the need to re-implement all common simulation functions (e.g. queue management, random number generation, etc.). This could be a time consuming and complex process. Another option is to use point and click simulation software (e.g. NetSim [CIS]) to create the simulations. The benefit of this type of simulation software is it is very user friendly. The user just needs to drag and drop a pre-defined simulation component and the simulation component is added to the simulation. The user does not need to know much about programming and can have all graphic results very easily and quickly. Unfortunately, this type of simulation is very restrictive. User cannot modify existing functionality or add new functionality to the simulation. Also, the speed of running the simulation is commonly slower than the first type [PE05].

There is another type between these two where the simulation is based on a program but a pre-defined simulation architecture and components are provided. One example of this type of simulation system is SSJ (Stochastic Simulation in Java) [PE05] from the Université de Montréal. SSJ is written in Java and uses Java as its main programming language and it contains most common simulation features in its library. The user just needs to add new components to the software and then specify the simulation parameters. Also, the user can modify existing functionality or add new features. This approach provides a compromise between powerful but complex “simulation from scratch” and simple but restrictive “drag and drop” simulation systems. Another benefit of using SSJ is that it is based on Java so it runs on many different machines and is operating system independent. As SSJ uses Java, it runs simulation faster than those created with ”drag and drop” system. SSJ also supports different types of simulation including event view, process view, continuous simulations or any combination of these types.

# Chapter 3

# Concise Problem Statement

To create a system where applications can be composed from components dynamically, it must be possible to find and access components quickly regardless of user location. Such components will be stored in component storage servers located throughout the Internet. Deciding which component storage servers to place components in to ensure efficient access is the first aspect of the problem. Then the problem of how to dynamically locate needed components quickly to compose an application must also be solved. There are certain challenges that must be dealt with to address these two sub-problems. Any solution must support user mobility and be location sensitive, (i.e., it must be possible to find the required components from *nearby* component storage servers). The solution must also be scalable with respect to the number of users and the number of components. The solution also needs to provide fast response and be reliable because users need to be sure that they will always have access to the required components if such a system is to be accepted as an alternative to applications stored locally on devices or the use of conventional

SaaS approaches.

# Chapter 4

# Solution Strategy and Prototype Implementation

## 4.1 Introduction

Figure 4.1 shows high-level abstraction of the solution strategy described in this thesis. A client with any device (e.g. in this example a smartphone) requests a component. First it connects to a nearby cache location server. Then, the cache location server queries in its database or, if necessary, asks one or more master location servers to find a component storage server's ID that has the component. The cache location server sends back the component storage server ID to the client. After that, the client and the component storage server have a direct connection and the client downloads its requested component. Component lookups are location-aware so that the nearest available component replicas are used. New components uploaded by users are automatically replicated and distributed by the system to ensure later efficient access.

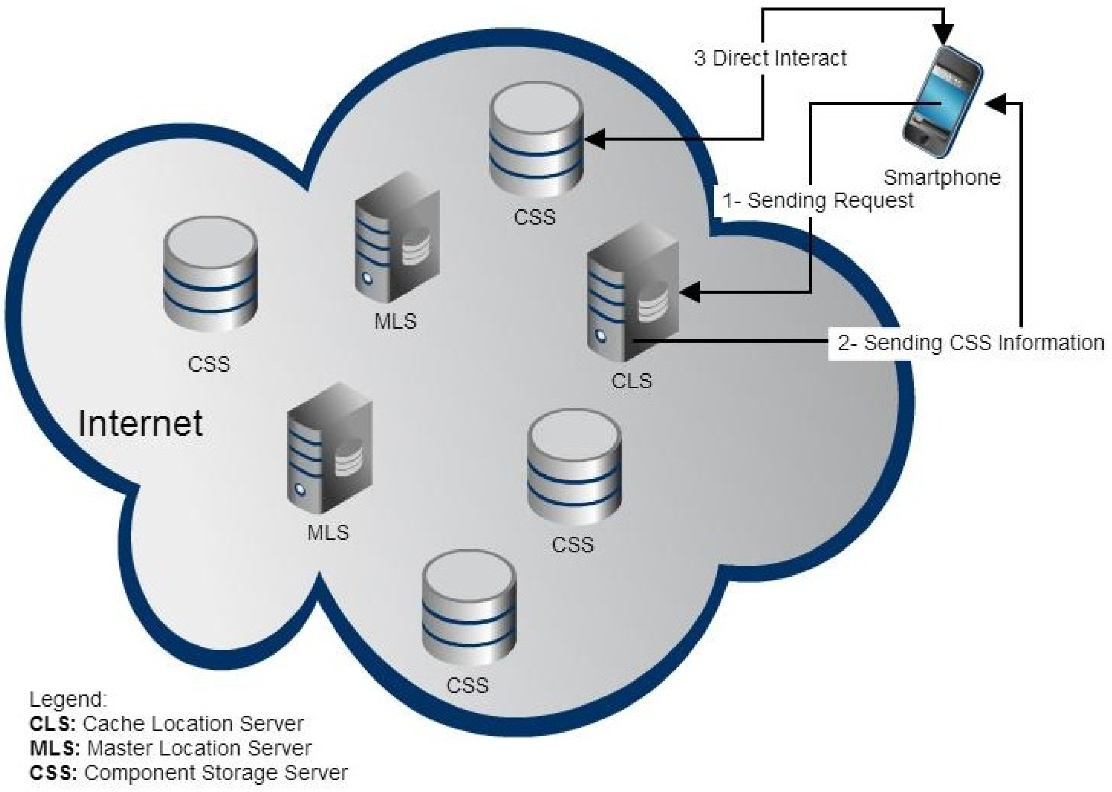

3 Direct Interact
Smartphone
1- Sending Request
2- Sending CSS Information
CSS
MLS
CLS
Internet
Legend:
CLS: Cache Location Server
MLS: Master Location Server
CSS: Component Storage Server

Figure 4.1: Example of the Solution Strategy [SKG09]

## 4.2 Components

A component is an identifiable part of an application that provides a particular function or group of related functions for the whole application (e.g. the animation part of the Microsoft PowerPoint). In a distributed computing system a component is a reusable part of the application that can be combined with other components and which may be stored at different locations in a network.

In my prototype implementation, which is done in Java, each component has two key attributes, which are the name of the component *Component_Name* and the version *Component_Version*. Every component has at least one method that represents the functionality of the component. For example *Time_100* is a component that prints the current time and has a version[1] number of 1.0.0. You can see the source code in Figure 4.2. The name of a component class will have the form *(Component-Name)_(Version without dots).java*. For example, in Figure 4.2, *ComponentName* = Time and *Version* = 1.0.0, so the class name will be Time_100.java

```java
public class Time_100 {
        private static final String VERSION = "1.0.0";
        private static final String COMPONENT_NAME = "Time";
        private static final String DATE_FORMAT_NOW ="yyyy-MM-dd HH:mm:ss";
        public static String getCurrentVersion(){
                return VERSION;
        }
        public static String getComponentName(){
                return COMPONENT_NAME;
        }
        public static String getCurrentTime(){
                String result = "";
                Calendar cal = Calendar.getInstance();
                SimpleDateFormat sdf = new SimpleDateFormat(DATE_FORMAT_NOW);
                result = sdf.format(cal.getTime());
                System.out.println("Current time is " + result);
                return result;
        }
}
```

Figure 4.2: Code of Time_100.java

This component includes the getCurrentTime() method which prints the current time of the system and can be dynamically loaded into a Java Virtual Machine (JVM) for use via Java's ClassLoader mechanism [CLA].

[1] There is a method in each component (getComponentVersion()) which returns the component's version.

## 4.3 System Entities

In my research there are four entities of interest, which are explained in the following subsections.

### 4.3.1 Clients

Client machines/nodes are where users generate requests to access/download components. Each client has two major responsibilities:

1. Searching for a component and downloading it from the closest component storage server when accessing a component created elsewhere.

2. Uploading and registering a new component to the closest component storage server when providing a new component for the use of others.

#### Searching for a component and downloading it

Each component has its own name and version. When a user needs a component, the component name and the component version are provided. The client node will then connect to the closest operational cache location server based on Euclidean distance. Then the client sends its request to the cache location server along with its latitude and longitude information. It then waits for the response of the closest component storage server that has the requested component. After downloading the component, the client checks if the component storage server which it downloaded from was reasonably close or not. If not, then the client uploads the component to its closest component storage server.

#### Uploading and registering a new component

The second task for the client is registering a new component to a component storage server when a user creates one. Again, the component storage server chosen is the closest component storage server to the client as described. After the client uploads its component, the client notifies its closest cache location server that the new component storage server has the component.

Also, if a client connects to a component storage server to download the requested component and component storage server sends the response that it does not have the component (the component could have been removed), then the client sends a message to the cache location server that this component storage server does not have the component. Then the cache location server update its database (removing the record of that component storage server) and searches again for another component storage server.

### 4.3.2 Cache Location Servers

The client always sends its component requests to the nearest cache location server. Cache location servers provide nearby copies of information on locally relevant components to provide quick lookup and access. Each cache location server has an empty database initially. The responsibility of the cache location server is to store the component name and its assigned component storage server ID for the next client request (as shown in Figure 4.8).

When there is more than one component storage server ID in the cache location server database, the cache location server will return the "best" component storage server based on shortest ping time. Cache location servers build up their databases using successful previous results from master location servers. Also cache location server records the component storage server information when a client has uploaded a new component to the client's nearby component storage server.

### 4.3.3 Master Location Servers

Each master location server has its own database that for each component stores component name, component version, and the ID of the component storage servers containing the component. Master location servers are the definitive source for location information. When a cache location server contacts the master location server for a client's requested component, it checks its own database and returns component storage server ID that contains the requested component. If more than one component storage server has the component, the master location server will return a list of IDs of the closest component storage servers based on the latitude and longitude of both the client and the component storage server. The number of items in the list is defined in the configuration file and the default value is 2 providing for support of basic redundancy. The IDs of the closest component storage servers in each master are returned to the cache location server and the cache location server will decide which to give to the requesting client based on the ping time.

Another responsibility of the master location server is to register new components' component storage servers IDs in all master servers that replicate that information. When a client uploads a new component to its closest component storage server based on latitude and longitude, the component storage server after downloading the component, sends a notification to the primary master location server and the master location servers in the master location server sequence list. The notification contains the componentName, componentVersion and the ID of the component storage server. This information will be used by cache location servers which do not have any information about requested components from their clients.

### 4.3.4 Component Storage Server

A node that stores components is a component storage server. A component storage server connects to a client to upload or download components. When a client uploads a component into a component storage server, the component storage server sends a notification to the primary master location server and the master location servers in the master location server sequence list asking them to update their location information to include the new component. Then the component storage server sends a copy of the new component to its closest component storage server. So there is redundant storage of each component. The selected alternate storage server also contacts the necessary location servers to have them update their information.

## 4.4 Use of Hash Function in My Prototype

Each component based on its name and version gets a key, which is used to register the component into master location servers that, as the name implies, store the locations of available components and their replicas. This key is generated by using a hash function.

For getting the master location server ID, first all letters of the component_Name are converted to uppercase letters (Component names are assumed to be case-insensitive and thus, for example a user may search for "Time" and another user searches for "time", but both are assumed to be searching for the same Time component). The ASCII code values of all uppercase letters are then simply summed to produce the hash code value. While this technique will likely not distribute all component names uniformly over the available hash space, it is sufficient for this application and is very simple and therefore efficient to compute. For example, if a component name is Time, then its hash value would be computed as follows:

Step 1- Retrieve Component Name: Component Name = Time

Step 2- Convert all letters to uppercase: = TIME

Step 3- Get ASCII code values for all Component_Name letters: TIME ⇒ T (84) I (73) M (77) E (69)

Step 4- Summing of ASCII code values: For TIME is 84+73+77+69 = 303

The second phase in computing a component's key is to get a numeric value of the version. For doing this, the value is the version number without dots. For example, if the component_version is 1.2.0 then the numeric value is 120.

After getting the component_name and the component_version **values**, I multiply them together. The primary master location server ID is thus computed as *(component_name value* × *component_version value) mod (number of master location servers)* which is done in the getServerID method. The source code of the getServerID method is shown in Figure 4.3. If the result is zero, the primary master location server ID is arbitrarily set to the number of master location servers in the configuration file. This primary master location server records information about the component's component storage server and then replicates this information to other master location servers for reliability.

```
public int getServerID(String componentName,String version) throws IOException{
        Config cfg = new Config();
        int numberOfServer = cfg.getNumberofMasterServers();
        int result = 0;
        int ID = getStringASCIIValue(componentName)  * getVersion(version);

        if (ID % numberOfServer != 0){
                result = ID % numberOfServer;
        }else{
                result = numberOfServer;
        }
        return result;
}
```

Figure 4.3: Code of getServerID Method

The hash function returns the same master location server ID for all components with the same name, and version number. When different components are hashed to the same master location server, they are identified by the server using their name and version number so there is no problem.

## 4.5 Redundancy

The hash function returns the primary master location server ID for each component and version. But for reliability we need to have more than one master location server that stores the component's information. The solution for this problem is redundancy.

The redundancy factor (total number of master location servers storing information for a given component) is set in the configuration file. Based on the number of master location servers and the redundancy factor and having the primary master location server's ID which comes from the hash function, *getServerIDSequence(int* *primaryNumber)* will return a sequence of master location server IDs that store information for a component of interest. This sequence is called the master location servers sequence. The number of master location server groups with a desired redundancy factor is *int(number of master location servers / redundancy factor)+1.*

For example, if we have 5 master location servers and the redundancy factor is 2 then there are 3 master location server groups:

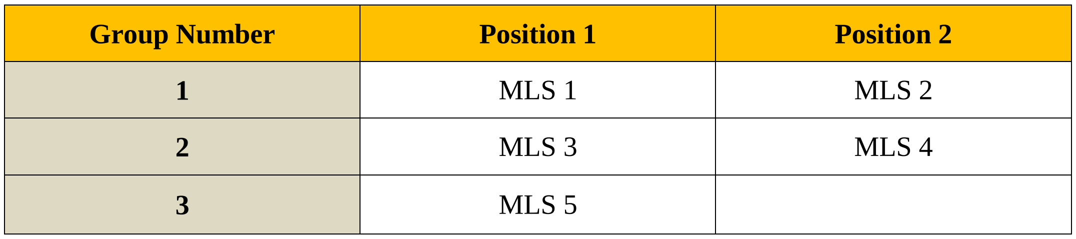
Group 1 includes (Master Location Server1, Master Location Server2),

Group 2 includes (Master Location Server3, Master Location Server4),

Group 3 includes (Master Location Server5)

Only master location servers with the same sequence number within their own group can be used as redundant servers. For example, Master Location Server1, Master Location Server3 or Master Location Server3, Master Location Server5 or Master Location Server2, Master Location Server4 can be selected as master location servers in this example provided that the primary master location server is Master Location Server1 or Master Location Server3 or Master Location Server2, respectively.

On the other hand, if getServerID selects Master Location Server 4 as the primary master location server, what will be chosen by the algorithm as the next master location server? Since Master Location Server 4 is placed as the second position in its own group (Group 2), and there is no such position in Group 3, the algorithm will select the second position from the previous group which is Group 1 and it will be Master Location Server 2. Figure 4.4 shows the grouping of five master location servers.



| **Group Number** | **Position 1** | **Position 2** |
|---|---|---|
| **1** | MLS 1 | MLS 2 |
| **2** | MLS 3 | MLS 4 |
| **3** | MLS 5 | |

Figure 4.4: Example of Grouping Five Master Location Servers

Based on the required number of selected master location servers which is set in the configuration file, the selecting master location server ID loop in the getServerID-Sequence method continues to get enough master location servers to create its master location servers IDs sets. Figure 4.5 shows the source code of the getServerIDSe-quence method. This sequence of master location server IDs would be again identical for components with the same name and version number and given redundancy factor in configuration file.

```java
public int[] getServerIDSequence(int primaryNumber) throws IOException{

        Config cfg = new Config();
        int numberOfServer = cfg.getNumberofMasterServers();
        int redundancyFactor = cfg.getRedundancyFactor();
        int numberOfSelectedServer = cfg.getNumberofSelectedMasterServers();
        int numberOfGroups = getNumberOfGroups(numberOfServer, redundancyFactor);
        int[] result = new int[numberOfGroups];
        result [0] = primaryNumber;
        int value = 0;
        int leftCounter = 1;
        int rightCounter =1;
        int totalCounter = 1;
        int internalCounter = 1;
        int i = 1;
        while (totalCounter <numberOfSelectedServer && i<numberOfServer){
                if (i%2 != 0){ value = primaryNumber + (rightCounter *
                    redundancyFactor);
                        if (value <= numberOfServer){
                                result[internalCounter] = value;
                                rightCounter++;
                                internalCounter++;
                                totalCounter++;
                        }
                }else{ value = primaryNumber - (leftCounter * redundancyFactor);
                        if (value > 0){
                                result[internalCounter] = value;
                                leftCounter++;
                                internalCounter++;
                                totalCounter++;
                        }
                }
                i++;
        }
        return result;
}
```

Figure 4.5: Code of getServerIDSequence method

## 4.6 Location Awareness

I have used two methods to decide on the closest component storage server to each client. The first is based on latitude and longitude of the client and component storage servers that store the component. Figure 4.6 shows the source code of the calcDistance method.

The master location server performs this distance calculation and sends the ID of the closest component storage servers to a requesting cache location server. The

```
//It comes from http://www.movable-type.co.uk/scripts/latlong.html
//http://www.csgnetwork.com/gpscoordconv.html
//http://support.microsoft.com/kb/213449
static double calcDistance(double lat1, double long1, double lat2, double
    long2){
        double distance = 0;
        double radius = 6371;
        double deltaLat, deltaLong;
        double a, c;
        deltaLat = Math.toRadians((lat2 - lat1));
        deltaLong = Math.toRadians(long2 - long1);
        a = Math.pow(Math.sin(deltaLat/2), 2) + (Math.cos(Math.toRadians(lat1
            ))*Math.cos(Math.toRadians(lat2))*Math.pow(Math.sin(deltaLong/2),
             2));
        c = 2 * (Math.atan2(Math.sqrt(a), Math.sqrt(1-a)));
        distance = radius * c;
        return distance;
}
```

Figure 4.6: Code of calcDistance Method

default number of closest component storage servers to be returned is predefined in the configuration file.

The second method used to find the closest component storage server in cache location servers is the average ping time. First the master location server sends the list of closest component storage servers based on latitude and longitude of the client to the cache location server. Then the cache location servers ping each component storage server and the component storage server with the shortest average ping time is sent to the client. By this method, cache location servers can also be assured that the chosen component storage server is currently online and can be accessed as quickly as currently possible.

## 4.7 The Component Download Process

The flow chart in Figure 4.7 shows the algorithm for the entire process for downloading a new component from the closest component storage server. The procedure a client uses to find a component is:

- Step1: The client sends the name and the version of required components to its nearby cache location server.
- Step 2: The cache location server receives the component name and version of the requested component.
- Step Q[2]1: The cache location server checks to see if the component is found in its database?
- Step Q1N[3]-1: The cache location server runs the hash function on the name and the version of the requested component to find the series of master location server IDs and sends the request component name and component version to these master location servers.
- Step Q1N-2: The master location server searches its database for the requested component and returns a sequence of component storage servers (the number of component storage servers is defined in configuration file) based on the closeness to the clients location.
- Step Q1N-3: The master location server sends information of these component storage servers to the cache location server.
- Step Q1N-4: The cache location server records this information to its database.

[2]The 'Q' means this step poses a 'Question'.
[3]The 'N' means the answer to the preceding question was 'No'.

- Step Q1Y-1: The cache location server runs ping method to find the closest currently online component storage server to the clients between its lookup results.
- Step Q1Y-2: The cache location server sends the IP address of the component storage server to the client.
- Step Q1Y-3: The client sends its request to the component storage server to download the component.
- Step Q1Y-4: The component storage server sends the requested component to the client.
- Q2: Was the responding component storage server the one close to the client?
- Q2N-1: The client sends a copy of the component to its closest component storage server and notifies the cache location server that the new component storage server has this component.
- QN-2: The cache location server records the new uploaded component's information and the component storage server's ID that the client has uploaded.
- QN-3: The component storage server copies the component to its closest component storage server for reliability and notifies the appropriate master location servers of the update.

The diagram in Figure 4.8 shows connections among the different entities in my system.

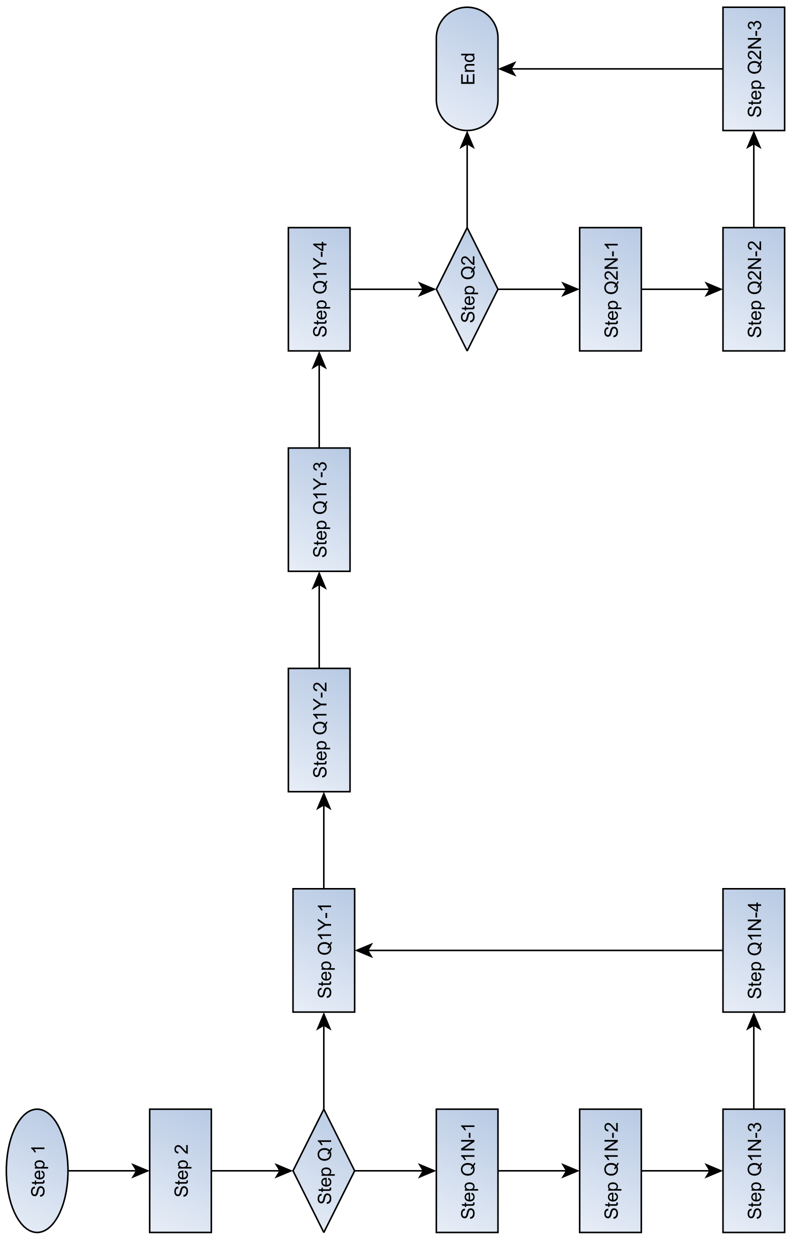


Figure 4.7: Flow chart of how a client downloads a new component

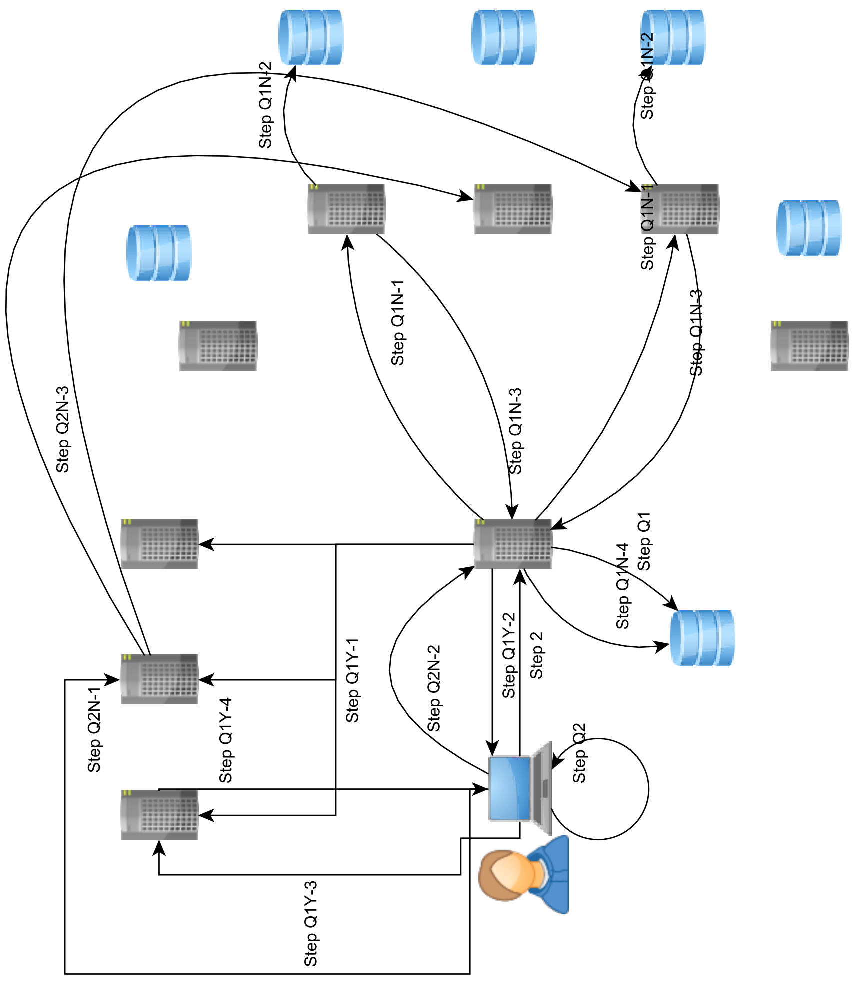


Figure 4.8: Communication between different entities

# Chapter 5

# Assessment

## 5.1 Introduction

To assess my strategy, I needed to run a number of simulations. I used BRITE to create a network topology that was structurally representative of the Internet but scaled down by a factor of approximately 1000 (in terms of area covered, number of routers, hosts, etc.) to make the simulations feasible in the time available. I used SSJ to develop my event-based simulation. In this chapter, I explain my usage of BRITE and SSJ and I describe the results from the different scenarios I used to assess my approach.

## 5.2 Simulation

I chose to use the BRITE [MLMB01] network generation tool in my research to create realistic models of the Internet on which I could simulate my system's behaviour. BRITE is an open source and universal network generator that was designed and implemented at Boston University. There are eight models supported in BRITE but it also has the ability to be extended, which means new models can be added to it or imported from other network generators [MLMB01].

BRITE defines five groups of models, which are *Flat Router model*, *AS Flat model*, *Hierarchical Top-down model*, *Hierarchical Bottom-up model*, and *models* imported from other network generators such as GT-ITM [MLMB01].

There are two models that are used in all BRITE groups except, of course, when importing from some other generator. The first model is the "Waxman" model in which the nodes are placed in the plane uniformly randomly and interconnected based on Waxman's function as described earlier in the Related Work chapter. The next model is "Barabasi" in which the tendency of a new node to connect to a popular node is higher than to other nodes. These two models are used in four sub-models of BRITE's Flat Router and Flat AS models. The hierarchical top-down model where first the ASs are created and then in each AS, nodes are connected based on some router models is also supported. The other sub-type is the Bottom-Up model where first nodes are connected based on an underlying router model and then these routers are grouped into different ASs. Figure 5.1 shows the BRITE models structure.

**BRITE Configuration**

In BRITE, there are two key configuration choices. The first is placement of nodes. There are two types of placement: random and heavy tailed.

The second configuration choice is bandwidth assignment between pairs of nodes.

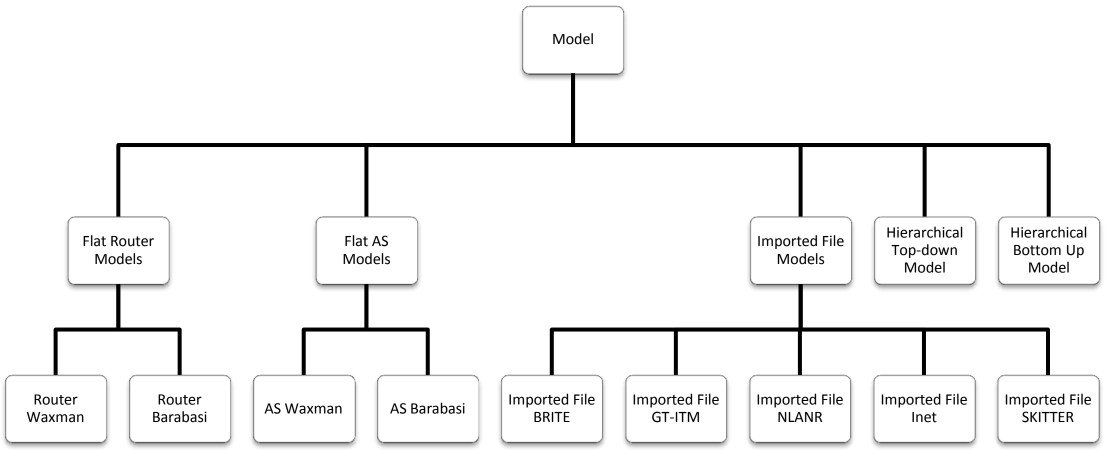


Figure 5.1: Structure of BRITE models adapted from [MLMB01]

The method used is *AssignBandwidth*. The value of the bandwidth is unit-less and can be interpreted based on specific requirements.

The *AssignBandwidth* method needs three variables to be set to calculate the bandwidth between two nodes:

1. Minimum bandwidth.

2. Maximum bandwidth.

3. Bandwidth distribution, which is the method of assigning the bandwidth. There are four different possibilities:

   - Constant: BRITE picks the minimum bandwidth for all bandwidths.

   - Uniform: The bandwidth value will be a uniformly distributed value between minimum and maximum bandwidth.

- Exponential: Bandwidth will increase exponentially by the mean BW(min)[1]
- Heavy-tailed: The value is Pareto distributed with minimum and maximum bandwidth (see Equation 5.1 [MLMB01] where $\alpha, k > 0$ and $x \geq k$ where k is the smallest random value).

$$p(x) = \alpha k^{\alpha} x^{\alpha - 1} \tag{5.1}$$

BRITE first reads its configuration parameters and then generates the requested topology based on the configuration file parameters. Tables 5.1 and 5.2 explain the parameters available for the two major models: Flat topology and Top-down hierarchical topology.

**BRITE Output**

After configuring BRITE through the GUI or the configuration file and running BRITE to generate the topology, BRITE produces output based on its configuration. BRITE's output file has three sections. The first section describes the model configuration. The second section has information about nodes and each line has information for a given node (see Table 5.3). The final section provides edge connection information between nodes (see Table 5.4).

[1]Minimum Bandwidth.

Table 5.1: Flat Topology (AS Only or Router Only) Parameters [MLMB01]

| Parameter | Meaning | Values |
|---|---|---|
| HS | Size of one side of the plane | $int \geq 1$ |
| LS | Size of one side of a high-level square | $int \geq 1$ |
| N | Number of nodes | $int 1 \leq N \leq HS * LS$ |
| Model | Model ID | $int \geq 1$<br>1: Router Waxman<br>2: Router Barabasi<br>3: AS Waxman<br>4: AS Barabasi<br>5: Top-down hierarchical<br>6: Button-up hierarchical |
| alpha | Waxman-specific exponent | $0 < \alpha \leq 1, \alpha \in R$ |
| beta | Waxman-specific exponent | $0 < \beta \leq 1, \beta \in R$ |
| Node Placement | How nodes join the topology | 1: Incremental<br>2: Random |
| m | Number of links per new node | $int \geq 1$ |
| Growth Type | How nodes join the topology | 1: Random<br>2: Heavy-tailed |
| BWdist | Bandwidth assignment to links | Constant<br>Uniform<br>Exponential<br>Heavy-tailed |
| MaxBW, MinBW | minimum, maximum link bandwidth values | $float > 0$ |

Table 5.2: Top-down Hierarchical Topology Parameters [MLMB01]

| Parameter | Meaning | Values |
|---|---|---|
| Edge Connection | Method for interconnecting router topologies | 1: Random node<br>2: Smallest degree<br>3: Smallest degree non-leaf<br>4: k-Degree |
| Intra BWdist | Intra-domain bandwidth assignment distribution | 1: Constant<br>2: Uniform<br>3: Exponential<br>4: Heavy-tailed |
| Intra BWMax/Min | Minimum, Maximum bandwidth values intra-domain links | $float > 0$ |
| Inter BWdist | Inter-domain bandwidth assignment distribution | 1: Constant<br>2: Uniform<br>3: Exponential<br>4: Heavy-tailed |
| Inter BWMax/Min | Minimum, Maximum bandwidth values for inter-domain links | $float > 0$ |

Table 5.3: Nodes Information in BRITE's Output File [MLMB01]

| Value | Description |
|---|---|
| NodeID | The identifier of the node |
| xPos | x-axis coordinate in the plane |
| yPos | y-axis coordinate in the plane |
| Indegree | Indegree of the node |
| Outdegree | Outdegree of the node |
| ASid | The AS's ID that node belongs to it |
| Type | The type of the node lke router, AS |

Table 5.4: Edges Information in BRITE's Output File [MLMB01]

| Value | Description |
|---|---|
| EdgeID | The identifier of the edge between two nodes |
| From | The origin nodeID |
| To | The destination nodeID in edge |
| Length | Euclidean length between two nodes |
| Delay | Propagation delay |
| Bandwidth | Edge's bandwidth |
| ASFrom | The AS's ID of the origin node |
| ASTo | The AS's ID of the destination node |
| Type | The type of the edge by classification routine |

Figure 5.2 shows a simple example of BRITE output for a network with 50 nodes and 107 edges between these nodes. Note that some details are omitted for simplicity. These sections are shown with ellipses.

```
Line1: BRITE Simulation Output
Line2: Header Section:  N,      HS,      LS,      Node Plancement, m,      alpha,  beta,
       Growth type,    BWDistribution, MinBandwidth,   MaxBandwidth
Line3: Node Section:            Node ID, xPos,  yPos,   indegree,       outdegree,
          ASid
Line4: Edge Section:            EdgeID, fromNode,       toNode, Length, Delay,
    Bandwidth, ASFrom,      ASTo
Line5: Topology: ( 50 Nodes, 107 Edges )
Line6: Model (5 - TopDown)
Line7: Model (3 - ASWaxman):  5 1000 100 1  2  0.15 0.2 1 1 1000.0 3000.0
Line8: Model (1 - RTWaxman):  10 1000 100 1  2  0.15 0.2 1 1 100.0 500.0
Line9: Nodes: ( 50 )
Line10: 1 258 964 4 4 0
...
Line11: 11 901 601 3 3 1
...
Line12: 21 209 323 5 5 2
...
Line13: 31 893 951 6 6 3
...
Line14: 41 611 377 4 4 4
...
Line15: Edges: ( 107 )
Line16: 7 3 2 310.04837 0.982 315.65 0 0
Line17: 8 3 1 523.2447 2.033 257.38 0 0
...
Line18: 27 15 13 1027.6498 3.109 330.55 1 1
Line19: 28 15 12 552.0082 3.046 181.24 1 1
...
Line20: 47 29 30 362.00552 2.092 173.03 2 2
Line21: 48 29 28 236.59671 1.192 198.57 2 2
...
Line22: 67 35 31 443.90652 1.503 295.27 3 3
Line23: 68 35 34 658.96436 2.182 301.95 3 3
...
Line24: 87 45 46 482.14935 1.453 331.94 4 4
Line25: 88 45 47 913.9125 1.846 495.05 4 4
...
Line26: 107 8 13 724.6544 0.603 1201.83 0 1
Line27: 108 35 17 379.86972 0.217 1750.55 3 1
Line28: 109 26 5 298.69046 0.101 2967.17 2 0
Line29: 110 39 4 1038.6309 0.437 2375.44 3 0
Line30: 111 26 16 1070.6937 0.747 1433.48 2 1
Line31: 112 49 16 982.28406 0.438 2245.19 4 1
Line32: 113 41 32 370.41733 0.174 2129.85 4 3
```

Figure 5.2: Example of a BRITE output

The output from BRITE begins with information describing the high level parameters of the network being generated. These are shown in the first 10 lines of the example in Figure 5.2. This is followed by sections detailing the individual nodes in the network [starting at line 11] and the edges (communication links) between them [starting at line 22 in the Figure 5.2].

For example, the line 1 258 964 4 4 0 means node 1 has xPosition=258, yPosition=964 and has an indegree of 4 and outdegree of 4 and is in AS of 0. For the edge example, 8 3 1 523.2447 2.033 257.38 0 0 means there is an edge between node 3 and node 8 from the same ASs (both are 0), the distance is 523.2447 and delay= 2.033 and the bandwidth=257.38 (the unit of the bandwidth can be set by the user). Presumably the unit for the distance and delay are also left unspecified and are thus subject to interpretation by the user. If so, you should say this. If not, specify the units.

#### Use of BRITE in my Thesis

In my simulation I generated a network topology with BRITE. In my simulated network, I placed 4000 routers in 10 different autonomous systems (ASs). The minimum bandwidth between ASs was 3Gbps and the maximum bandwidth was 20Gbps. And the minimum bandwidth between routers in each AS was 100Mbps and the maximum bandwidth was 4Gbps. In my network topology, there were 8020 edges generated between these routers.

### 5.2.1 Use of SSJ in my Thesis

I chose SSJ (Stochastic Simulation in Java) [PE05] from the Université de Montréal to run my thesis simulation.

In my simulation, I used the event view of SSJ. In event view, each simulated event needs to extend from the Event class and has an action method. I created the following events corresponding to activities in the proposed system and in my implemented prototype:

- EventGenerateClientRequestMessage: In this event, after reading a user request, the simulation creates a message that is sent to a cache location server near the client.
- EventSearchCacheServerDatabase: In this event, the cache location server looks for the requested component in its database. If the cache location server finds the component in its database, it will send a response to the client otherwise it will create a hash function event.
- EventHashFunction: If the cache location server cannot find any information about the requested component, the hash function must be run to determine the selected master location servers that will be asked to provide a location for the requested component.
- EventSearchMasterServerDatabase: After receiving cache location servers message for a requested component each selected master location server must search their databases for the best two component storage servers (based on distance to the client).
- EventAddMasterServerResponsestoCacheServer: Master location servers send the result of their searches to the requesting cache location server that must update its information.

- EventGenerateClientMessageToDownloadFromRepository: Cache location servers wait for all responses from master location servers and then send the best component storage servers ID (based now on ping time to the client) to the client so it can create a message to download the component from a component storage server.

- EventRepositoryChecksDatabase: After receiving a message from a client, the component storage server, must search its database for the component and send the component to the client.

- EventCheckIftheClosestRepository: The client checks if the component storage server was its closest component storage server or not. If not, it triggers an EventGenerateClientMessageToUpload event (so there will be a nearby copy in the future).

- EventGenerateClientMessageToUpload: The client creates a message to upload the component to its closest component storage server.

- EventUploadComponentToRepository: The client uploads the component to its closest component storage server.

- EventSendMessageToMasterServers: After receiving a replica of the component, the new local component storage server sends a notification to the associated master location server (based on hash function) that it now has a new component.

- EventSendMessagetToUpdateCacheServer: The component storage server sends another notification to the client's cache location server that it has the component.

- EventRequestCompleted: When a request is completed, the status of the request changes to completed. This is used for statistics gathering during the simulation.

- EventUpdateRequest: If a request is an update request, it creates a new record in the "updated component table" (For any search operation request, first simulation checks this table if there is a new version of component available) for future requests and starts uploading the updated component to the clients closest component storage server.

- EventFailure: When a host failure event occurs, based on the scenario (failure for cache location server, for master location server or failure for component storage server) a random server is chosen and added to the "failed event table" (when sending message to any server, first the simulation checks this table and if the server is failed, tries a different server).

- EventMovingClient: When a user changes location, a move event occurs and the client moves to a new location where a new cache location server is needed.

- EventNewCacheServer: When a cache location server associated to the client fails, the client must find a new cache location server.

A simulation starts with events corresponding to user actions such as uploading or accessing components occurring at simulated hosts where the users reside. These lead to the creation of other events at other simulated hosts in response to messages created to “implement” the user actions. During the course of a simulation, certain hosts (possibly running various servers) may fail. Also, users may relocate to new locations. These latter two scenarios are asynchronous to, but may affect, normal user request processing. They are generated stochastically in a subset of my simulation scenarios.

## 5.3 Results from the Simulation Work

After creating my simulation with SSJ and Java, I designed different scenarios to assess the behaviour of my system.

### 5.3.1 Scenarios

I designed 12 different scenarios and in each scenario I just changed one property of the system. There are also simulation constants that are fixed for all scenarios.

#### Simulation Constants

To make the simulations run in a reasonable time I had to model a scaled down version of the Internet (reduced by a factor of approximately 1000). This means I scaled the earth to a surface of 380km by 380km ( 145,000 km$^2$) instead of 149 million km$^2$ of land surface on the earth. Then I placed 4000 routers on this surface.

The first step of the pre-run phase of the simulation was to randomly select six dense population areas corresponding to large cities with many expected users of the system. I selected six random routers which have at least 6 and a minimum of 15 routers around themselves with distance less than 20km. These main routers have distance of at least 100km from each other. From the rest of the routers, I chose one single router from every 100 routers where I placed component storage servers and master location servers and called them component storage routers and master location routers. Finally I selected one router from every four routers to have cache location servers which is called cache location routers which should be widely distributed for overall system efficiency. All these routers were selected uniform randomly.

The next step was to assign component storage servers, master location servers and cache location servers to these selected routers. For dense population areas' routers, I assigned $\lceil(4 * 2 * number\ of\ routers\ in\ the\ area)/(number\ of\ routers\ in\ the\ area + 1)\rceil$ component storage servers and master location servers[2]. I also assigned one cache location server to each router. For the single component storage routers, I assigned 2 component storage servers and one cache location server and also assigned a master location server and one cache location server to each master location routers. Finally, I assigned one cache location server to each cache location routers. For example, in dense population areas like New York or Tokyo, there are more clients to request components in the system and in a small city like Stonewall, because of the population, there are far fewer clients and therefore far fewer requests. Therefore, In New York or Tokyo more servers required and in small cities, we do not need to have a server.

After assigning servers to the system, I placed 500 clients to these routers. At least 80% of my clients belong to dense population area. For each client, I selected

[2]The formula used to determine the number of servers in dense population area was selected to provide more capacity (4*) with redundancy (2*) per server site while maintaining a minimum of two servers of each type (for reliability).

the nearby cache location server and component storage server by distance.

The next step of the simulation was creating components. I created three categories of components which were very popular components, popular components and non-popular components reflecting probability of access. (For example, components in an office processing suite might be very popular and thus widely used while components in a program for simulating river flooding might be non-popular and hence infrequently used and possibly only in a geographically limited area.). The distribution of number of components by category was 1% very popular components, 4% popular components and 95% non-popular components. 90% of search operations were for very popular components, 8% for popular components and 2% for non-popular components. The components' sizes were assigned uniformly randomly between 10KB to 500KB. And also the name and version of the component were selected uniform randomly.

The final step of the pre-run phase of simulation was generating operations. First, all components were uploaded to the system based on generated user operations, and then a sequence of operations (search operation, upload operation, update event , failure event and moving event involving the initially uploaded components) were created to assess different situations of interest.

Other specific constant values of interest in my simulation were generating message time costs, database operation (searching, updating, deleting) time costs, message size between different entities and number of best component storage server and master location server.

## Simulation Default Values

There was a configuration file for each scenario in which one variable was changed for each experiment and the remaining values were fixed. I designed 12 scenarios and for each scenario I defined three experiments. I picked up a default value and changed that value for each experiment and then ran each experiment five times. My default values for each scenario are in Table 5.5

Table 5.5: Simulation Configuration Values

| Field Name | Default Value | Description |
|---|---|---|
| Number of Components | 1000 | Default number of components in the simulation |
| Number of Clients | 500 | Default number of clients in the simulation |
| Number of Operations | 10000 | The first 1000 operation is upload operation for all components and the rest could be search or upload operation or other events |
| Simulation Time Horizon | 15000 | Time to run the simulation |
| Number of Upload Operations | 100 | 10% of operation are upload operations |
| Number of Search Operation | 900 | 90% of operation are search operations |
| Number of Update Events | 0 | Just scenario 4 has update events. |
| Number of Failure Events | 0 | Just scenario 5, 6 and 7 have failure events. |
| Number of Moving Events | 0 | Just scenario 8 has moving events. |
| Percentage of Clients in Dense Population Area | 80 | Default percentage of clients in dense population areas |
| Component Distribution | 1-4-95 | 1% of very popular components, 4% of popular components and 95% of non-popular components |
| Number of Very Popular Components | 13 | These numbers were calculated based on the number of components and component distribution value |
| Number of Popular Components | 40 | |
| Number of non-Popular Components | 947 | |
| Factor for Number of Single Component Storage Routers | 100 | Selecting 1/100 of non-dense population area routers for component storage servers. |
| Factor for Number of Single Master Location Routers | 100 | Selecting 1/100 of non-dense population area routers for master location routers |
| Factor for Number of Cache Location Routers | 4 | Selecting 1/4 of non-dense population area routers for cache location servers. |
| Number of Component Storage Servers | 126 | These number were calculated based on dense population area and the constant variables were discussed in 5.3.1 |
| Number of Master Location Servers | 87 | |
| Number of Cache Location Servers | 1150 | |

I designed 12 scenarios to study the behaviour of the system.

### 5.3.2 Scenario 1 - Increasing the Number of Components

**Goal**

The goal of this scenario was to study the effect of increasing the number of components in the average download time.

**Expectation**

If the number of component is increased, then there is less chance of repeat search operations for components. Therefore, there is less chance to find the location of the component in the client's nearby cache location server. And also there is less chance to download the component from the client's nearby component storage server. Therefore, the result of increasing number of components will increase the average download time.

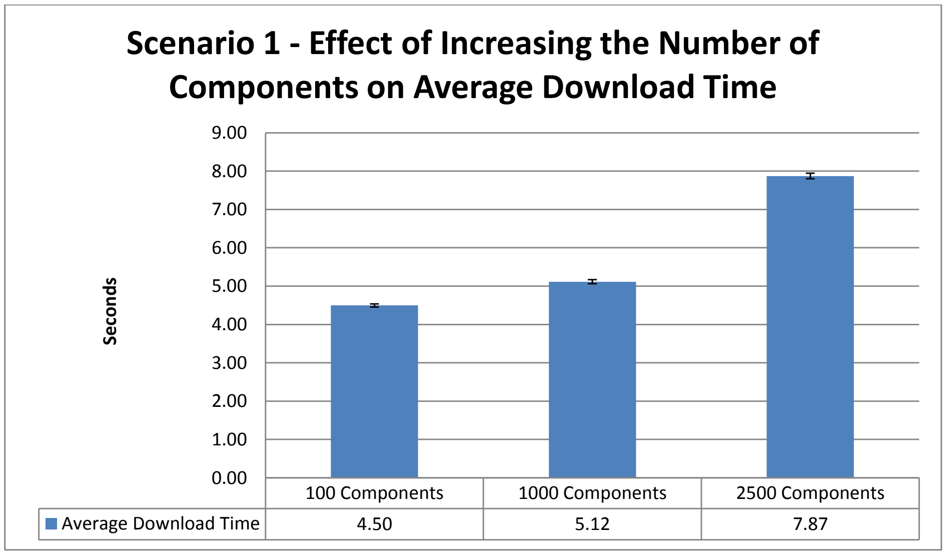
**Result**

I ran this scenario with three different experiments, five times each. The first experiment had 100 components, the second experiment had 1000 components and the third experiment had 2500 components.

Table 5.6 and Figure 5.3 shows the scenario 1 results.

Table 5.6: Scenario 1 - Effect of the Increasing the Number of Components on Average Download Time

| | Experiment | | |
|---|---|---|---|
| | 1 | 2 | 3 |
| **Number of Components** | 100 | 1000 | 2500 |
| **Average Download Time(seconds)** | 4.50 | 5.12 | 7.87 |
| **Standard Deviation** | 0.04 | 0.05 | 0.07 |



| | 100 Components | 1000 Components | 2500 Components |
|---|---|---|---|
| ■ Average Download Time | 4.50 | 5.12 | 7.87 |

Figure 5.3: Scenario 1 - Effect of Increasing the Number of Components on Average Download Time

As expected, the average download time increases with the number of components, ceteris paribus due to less benefit from access to nearby information (storage server locations and component replicas).

### 5.3.3 Scenario 2 - Increasing the Number of Clients

#### Goal

The goal of this scenario was to study the effect of increasing the number of clients on the average download time.

### Expectation

If the number of clients is increased more, clients are placed randomly in more different locations. The number of cache location servers is fixed and the chance that a cache location server connects many clients is going to be decreased since more clients will be in sparsely populated areas. Thus, cache location servers are less busy and their databases contain fewer entries. Hence, the chance that a cache location server has information about a component being searched for will be decreased. Therefore, the average download time should, again, be increase.

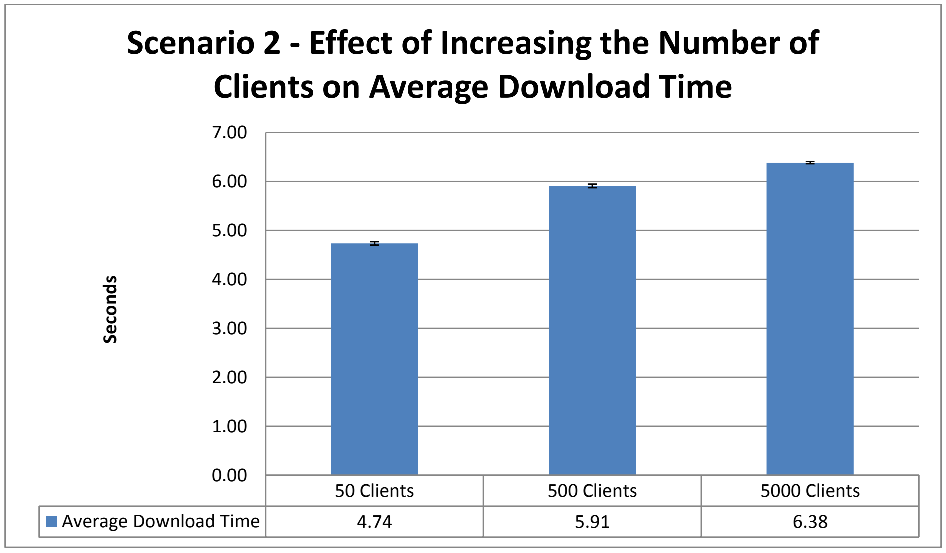
### Result

I ran this scenario with three different experiments, five times each. The first experiment had 50 clients, the second experiment had 500 clients and the third experiment had 5000 clients.

Table 5.7 and Figure 5.4 shows the scenario 2 results.

Table 5.7: Scenario 2 - Effect of Increasing the Number of Clients on Average Download Time

| | Experiment | | |
|---|---|---|---|
| | 1 | 2 | 3 |
| **Number of Clients** | 50 | 500 | 5000 |
| **Average Download Time(seconds)** | 4.74 | 5.91 | 6.38 |
| **Standard Deviation** | 0.03 | 0.04 | 0.02 |



| | 50 Clients | 500 Clients | 5000 Clients |
|---|---|---|---|
| ■ Average Download Time | 4.74 | 5.91 | 6.38 |

Figure 5.4: Scenario 2 - Effect of Increasing the Number of Clients on Average Download Time

As predicted, the average download time increases since the caching of storage locations in less effective. The overall change from 50 to 5000 clients is relatively small since component replicas are still being cached.

### 5.3.4 Scenario 3 - Increasing the Number of Operations

#### Goal

The goal of scenario 3 was to study the behaviour of the system as the number on search and upload operations changes to determine its effect on average download time.

### Expectation

If the number of search operations increases with the same number of clients and components, the chance of a repeat request from a given client for the same component will increase. After a number of initial requests, a nearby cache location server should have a high probability of storing information for a component being requested so the average download time should decrease. Similarly, the chance a cache location server has information about a component needed by other clients will also be increased.

### Result

I ran this scenario with three different experiments, five times each. The first experiment had 10,000 operations and the second experiment had 30,000 operations. Finally, the third experiment had 100,000 operations.

Table 5.8 and Figure 5.5 shows the scenario 3 results.

Table 5.8: Scenario 3 - Effect of Increasing the Number of Operations on Average Download Time

| | Experiment | | |
|---|---|---|---|
| | 1 | 2 | 3 |
| **Number of Operations** | 10,000 | 30,000 | 100,000 |
| **Average Download Time(seconds)** | 5.55 | 4.65 | 4.20 |
| **Standard Deviation** | 0.04 | 0.02 | 0.01 |

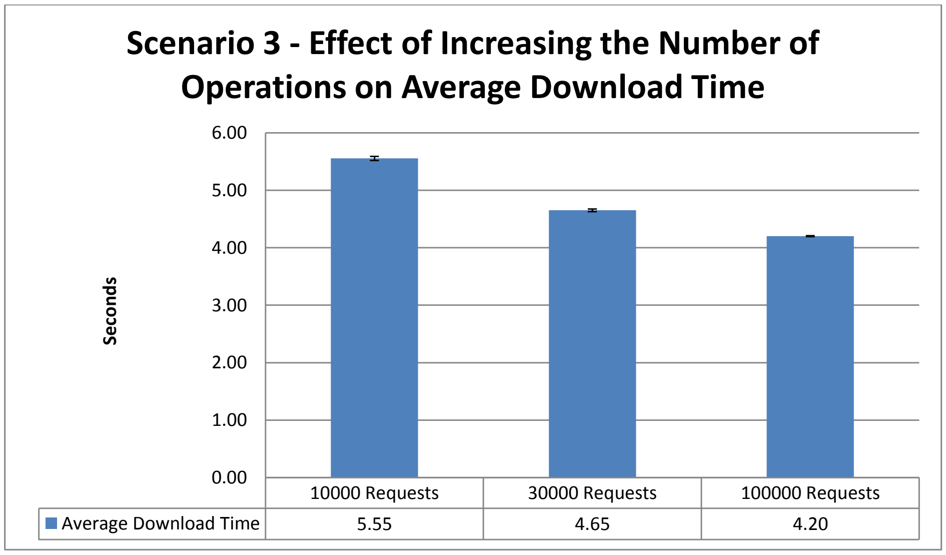


Figure 5.5: Scenario 3 - Effect of Increasing the Number of Operations on Average Download Time

The results confirm that as the number of requests grows, the average download time decreases showing that the system is effective in providing fast access for "popular" components that are accessed repeatedly.

### 5.3.5 Scenario 4 - Increasing the Number of Component Update Events

#### Goal

In the real system, the components will periodically be updated to accommodate feature updates, security patches, etc. The goal of this scenario was to study the average download time when the number of update events increases.

### Expectation

In my simulation, when an update event happens for a component, I update the component's version and increase the size of the component by 10% (reflecting the general trend of code to grow in size). Then the assigned client uploads the component to its nearest component storage server. The component storage server updates the assigned master location servers and makes a copy to its replica. After that, when a client searches for the component with the old version number, the clients download the new version of the component[3] Because information for the new version of the component will not be in nearby cache location servers, the client needs to contact a master location server for information about the new version of the component. Then the nearby cache location server will get an updated record for the new component. Due to the overhead associated with this processing, the average download time is expected to increase with an increase in the number of update events.

### Result

I ran this scenario with three different experiments, five times each. In the first experiment the probability of an update event was 1.96%. In the second experiment the probability of an update event was 9.09%. For the third experiment, I changed the probability of an update event to be 33.33%[4]. In the graphs for this scenario and other scenarios, I rounded the probabilities (e.g. 1.96% was rounded to 2%).

Table 5.9 and Figure 5.6 shows the scenario 4 results.

[3]Of course, it may be desirable to maintain multiple versions of a component but this is not considered in my experiments.

[4]Because in my simulation config file is based on the number of events, not percentage, these numbers are not rounded.

Table 5.9: Scenario 4 - Effect of Increasing the Number of Component Update Events on Average Download Time

| | Experiment | | |
|---|---|---|---|
| | 1 | 2 | 3 |
| **Probability of Update Event** | 2% | 9% | 33% |
| **Average Download Time(seconds)** | 8.89 | 9.74 | 9.91 |
| **Standard Deviation** | 0.09 | 0.04 | 0.08 |

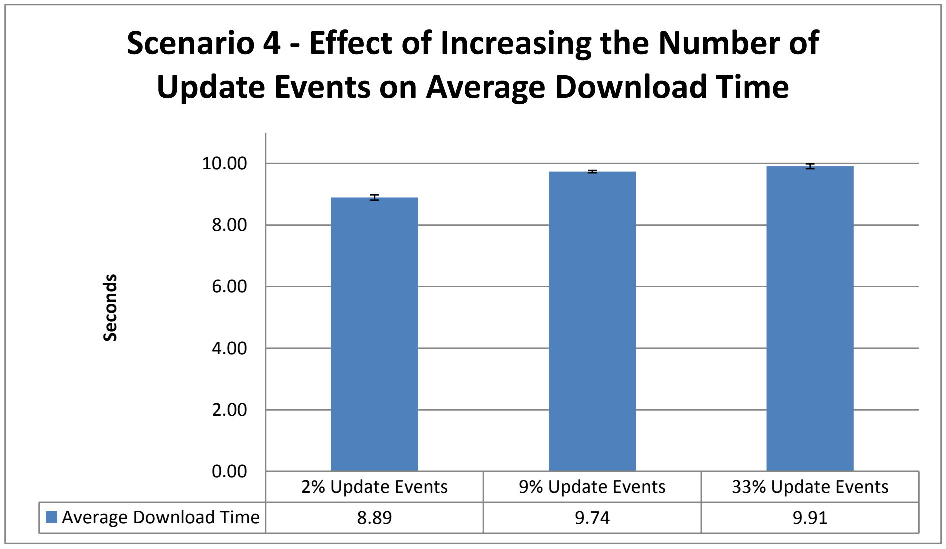


Figure 5.6: Scenario 4 - Effect of Increasing the Number of Component Update Events on Average Download Time

As expected, the overhead of the update process does have an impact on the average download time. When the number of operations is fixed and the number of update events is increased, the number of search operations is decreased. The difference between average download times between experiment 2 and experiment 3 is, at first glance, small but the average number of search operations in experiment 2 was 8112 operations and for experiment 3 was 4476 operations. This is a significant difference as compared to the number of search operations for experiment 1

(8121) and experiment 2 (8112). This explains the smaller difference in the results for experiments 2 and 3.

### 5.3.6 Scenario 5 - Increasing the Probability of Cache Location Server Failures

#### Goal

Any entities in a real system can fail. One of the scenarios I designed was to assess the effect of increasing the probability of cache location servers' failures.

#### Expectation

When a cache location server fails, the client cannot find any information about its requested component in the nearby cache location server. This cache location server failure results in two types of overhead: First, the time out waiting for the cache server that has failed and second, the time to find another nearby cache location server. These two overheads should increase the average download time.

#### Result

I ran this scenario with two sets of configurations. In the first configuration as a default 80% of the clients were placed in dense population areas and 20% elsewhere.

Table 5.10 and Figure 5.7 show the scenario 5 results when 80% of the clients were placed in dense population areas.

Table 5.10: Scenario 5 - Effect of Increasing the Probability of Cache Location Server Failures on Average Download Time when 80% of Clients Placed in Dense Population Areas

| | Experiment | | |
|---|---|---|---|
| | 1 | 2 | 3 |
| **Probability of CLS Failure Event** | 0.05% | 0.3% | 1% |
| **Average Download Time(seconds)** | 5.28 | 5.38 | 5.88 |
| **Standard Deviation** | 0.15 | 0.19 | 0.40 |

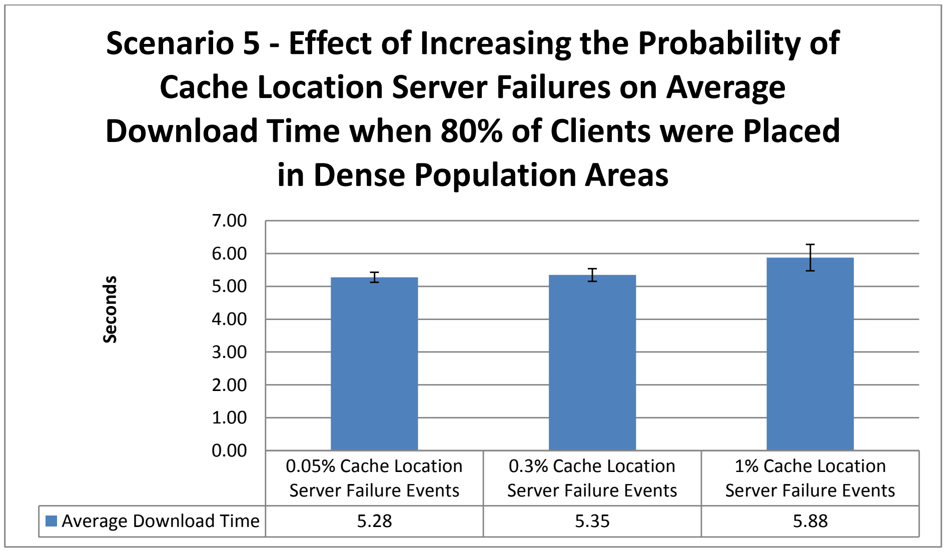


Figure 5.7: Scenario 5 - Effect of Increasing the Probability of Cache Location Server Failures on Average Download Time when 80% of Clients were Placed in Dense Population Areas

There are overlaps between these results when considering the error bars for different experiments. The reason I suspected for the significant error values for each experiment is the variation in the number of failed cache location servers in each run. The number of failed cache location servers and their placement were distributed uniform randomly. When most of the failed cache location servers were in dense population areas in one run and in another run the converse was true, number is different, then the error bars were large.

To confirm my suspicion, I set up another scenario configuration. All the previous configuration values were fixed except the percentage of clients in dense population areas which I set to zero. (i.e. all clients were placed uniform randomly across the geographic space). Table 5.11 and Figure 5.8 show the revised scenario 5 results when all clients were placed uniform randomly.

Table 5.11: Scenario 5 - Effect of Increasing the Probability of Cache Location Server Failures on Average Download Time when all Clients Placed Uniform Randomly

| | Experiment | | |
|---|---|---|---|
| | 1 | 2 | 3 |
| **Probability of CLS Failure Event** | 0.05% | 0.3% | 1% |
| **Average Download Time(seconds)** | 10.68 | 10.98 | 11.46 |
| **Standard Deviation** | 0.06 | 0.08 | 0.20 |

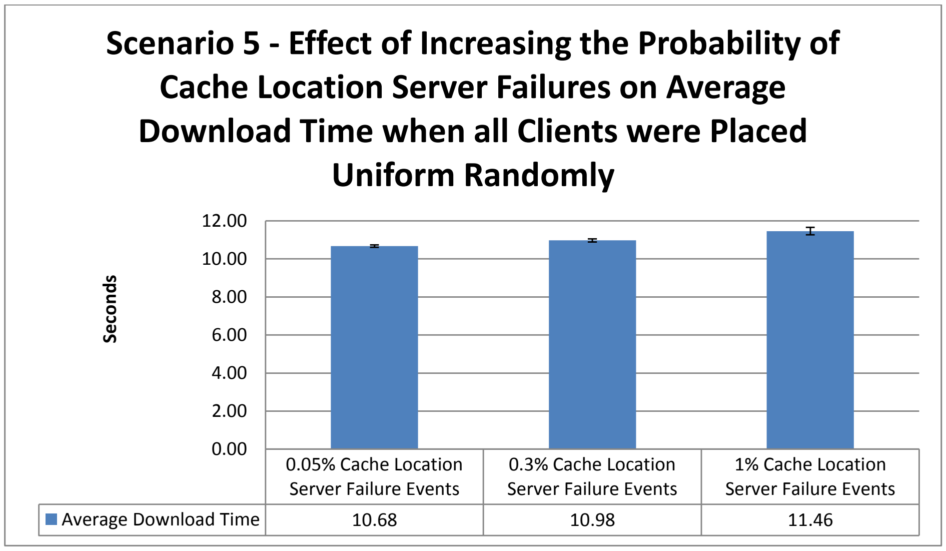


Figure 5.8: Scenario 5 - Effect of Increasing the Probability of Cache Location Server Failures on Average Download Time when all Clients were Placed Uniform Randomly

With uniform random placement of clients, we observe the originally expected behaviour: slower average downloads with additional failures.

### 5.3.7 Scenario 6 - Increasing the Probability of Master Location Server Failures

#### Goal

Like cache location servers, Master Location Servers can fail. I designed this scenario to study the effect of master location server failures on average download time.

#### Expectation

When a master location server fails, there is another (alternate) master location server to respond to the cache location server requests for component information. The failure of a master location server, however, will incur the cost of the time out of connection between the cache location server and the failed master location server. Therefore, by increasing the probability of failure events for master location servers, the average download time should also be increased.

#### Result

I ran this scenario for two different configurations for three different experiments, five times each. In the first configuration 80% of clients were placed in dense population areas. Table 5.12 and Figure 5.9 show the scenario 6 results when 80% of the clients were placed in dense population areas.

Table 5.12: Scenario 6 - Effect of Increasing the Probability of Master Location Server Failures on Average Download Time when 80% of Clients were Placed in Dense Population Areas

| | Experiment | | |
|---|---|---|---|
| | 1 | 2 | 3 |
| **Probability of MLS Failure Event** | 0.1% | 0.5% | 2% |
| **Average Download Time(seconds)** | 5.80 | 8.70 | 19.41 |
| **Standard Deviation** | 0.19 | 1.63 | 3.37 |

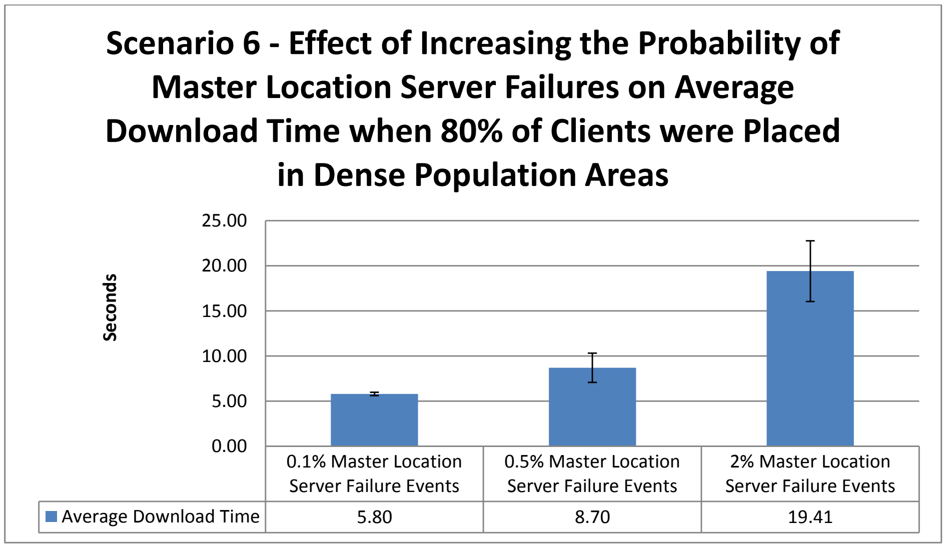


Figure 5.9: Scenario 6 - Effect of Increasing the Probability of Master Location Server Failures on Average Download Time when 80% of Clients were Placed in Dense Population Areas

I also ran this scenario when all clients were placed uniform randomly. Table 5.13 and Figure 5.10 show these results.

Table 5.13: Scenario 6 - Effect of Increasing the Probability of Master Location Server Failures on Average Download Time when all Clients were Placed Uniform Randomly

| | Experiment | | |
|---|---|---|---|
| | 1 | 2 | 3 |
| **Probability of MLS Failure Event** | 0.1% | 0.5% | 2% |
| **Average Download Time(seconds)** | 11.29 | 13.95 | 22.42 |
| **Standard Deviation** | 0.40 | 1.90 | 4.91 |

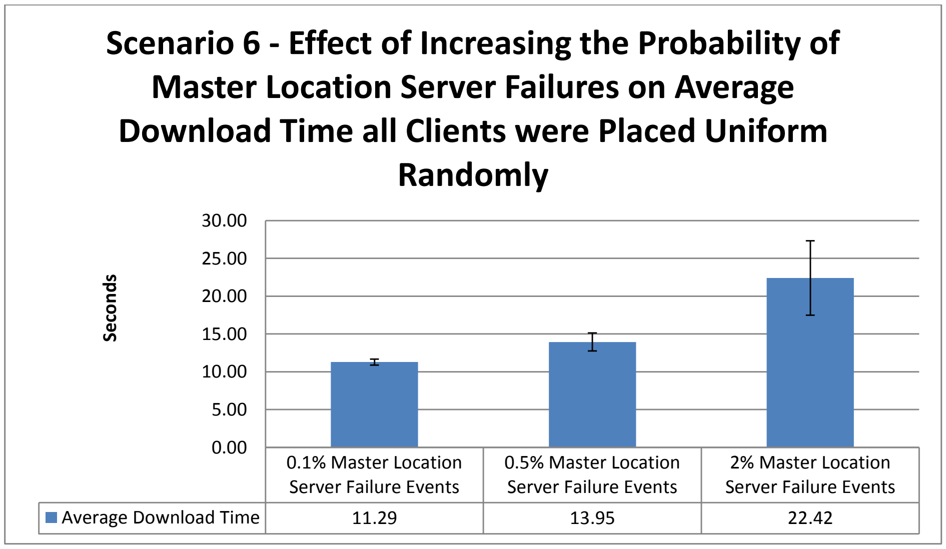


Figure 5.10: Scenario 6 - Effect of Increasing the Probability of Master Location Server Failures on Average Download Time when all Clients were Placed Uniform Randomly

There is a large different in average download time between Figure 5.9 and Figure 5.10 which is expected as in first case, 80% of clients were placed in dense population areas and in the second case, all clients were placed uniform randomly. In scenario 9, I am going to explain more about it.

### 5.3.8 Scenario 7 - Increasing the Probability of Component Storage Server Failures

#### Goal

Another system entity that can fail is the component storage server. I designed Scenario 7 to study of the effect of increasing the probability of component storage server failures on average download time.

#### Expectation

When a component storage server fails, there are four associated time costs. First the client gets the time out from the component storage server trying to download a component. (Of course, the client can no longer download any other previously downloaded components from its nearby component storage server either). There is also a second time out between the connection of the cache location server and the component storage servers to select the best component storage server based on the ping time. Also, access to a replica failed, there is a time delay trying to find another replica in a different nearby component storage server. All these time costs suggest that by increasing the probability of component storage server failure events, the download average time should increase. (Also the average download time for upload operations should be increased as the client cannot upload the components to its nearby component storage server).

## Result

To study this effect, I again designed two configurations for this scenario. In the first configuration, I placed 80% of the clients in the dense population areas.

Table 5.14 and Figure 5.11 show the scenario 7 results for this configuration.

Table 5.14: Scenario 7 - Effect of Increasing the Probability of Component Storage Server Failures on Average Download Time when 80% of the Clients were Placed in Dense Population Areas

| | Experiment | | |
|---|---|---|---|
| | 1 | 2 | 3 |
| **Probability of CSS Failure Event** | 0.05% | 0.3% | 1% |
| **Average Download Time(seconds)** | 6.11 | 14.41 | 22.59 |
| **Standard Deviation** | 0.03 | 5.70 | 13.76 |

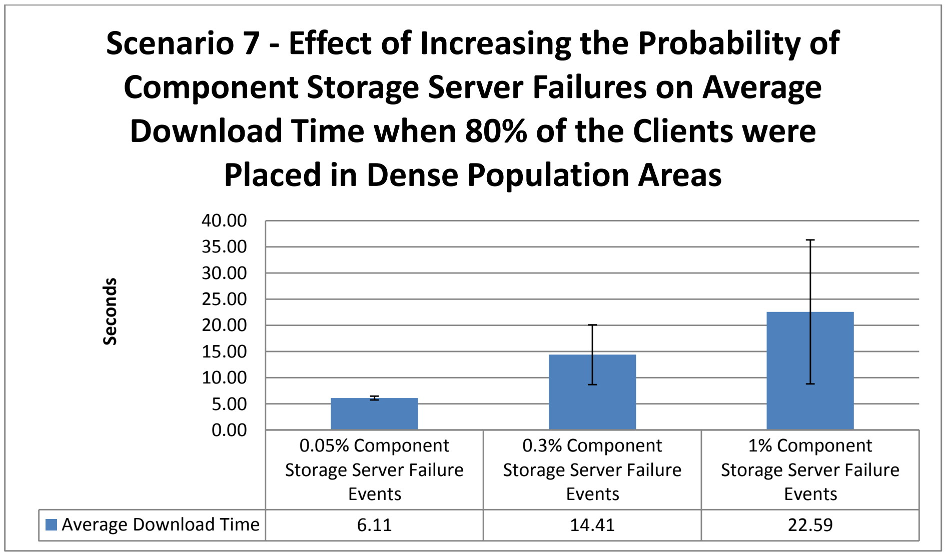


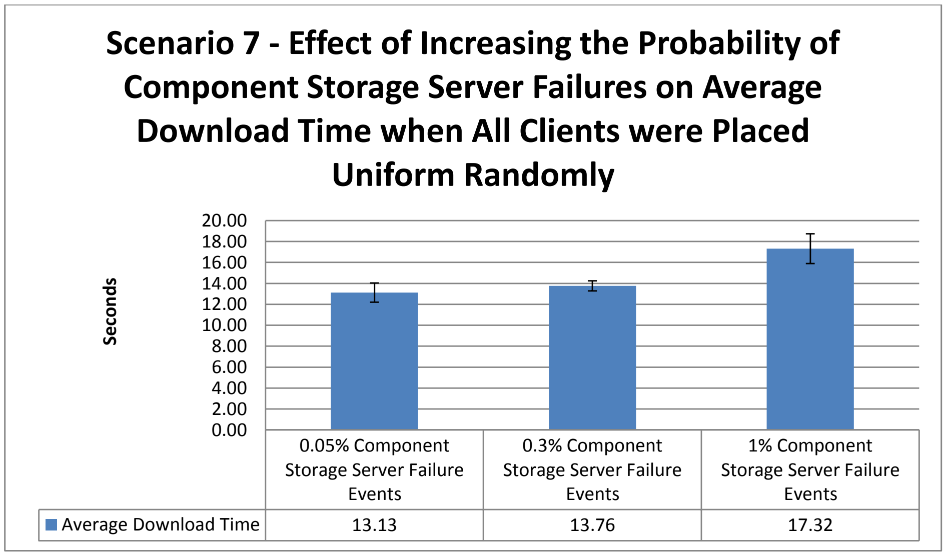
Figure 5.11: Scenario 7 - Effect of Increasing the Probability of Component Storage Server Failures on Average Download Time when 80% of the Clients were Placed in Dense Population Areas

As the results show the average download times were increased but the error bars were large and overlapping for the two right hand bars in Figure 5.11. The reason for these significant error values was surmised to again probably be the placement of the clients. The number of failed component storage servers in dense population areas was probably very different for each run, and thus the standard deviations were significant.

To confirm this assumption, I ran this scenario for another configuration where I placed all clients uniform randomly. The number of components was decreased to 100 and number of operations was set to 1000 and the simulation time horizon was increased to 40000 seconds. I needed to change these default configuration values for this scenario because with the default values there were more outstanding search operations at the end of the simulation which affected the simulations resulting in misleading results (Failing component storage servers takes more time and all search operations could not be completed). Table 5.15 and Figure 5.12 show the revised scenario 7 results when all clients were placed uniform randomly.

Table 5.15: Scenario 7 - Effect of Increasing the Probability of Component Storage Server Failures on Average Download Time when All Clients were Placed Uniform Randomly

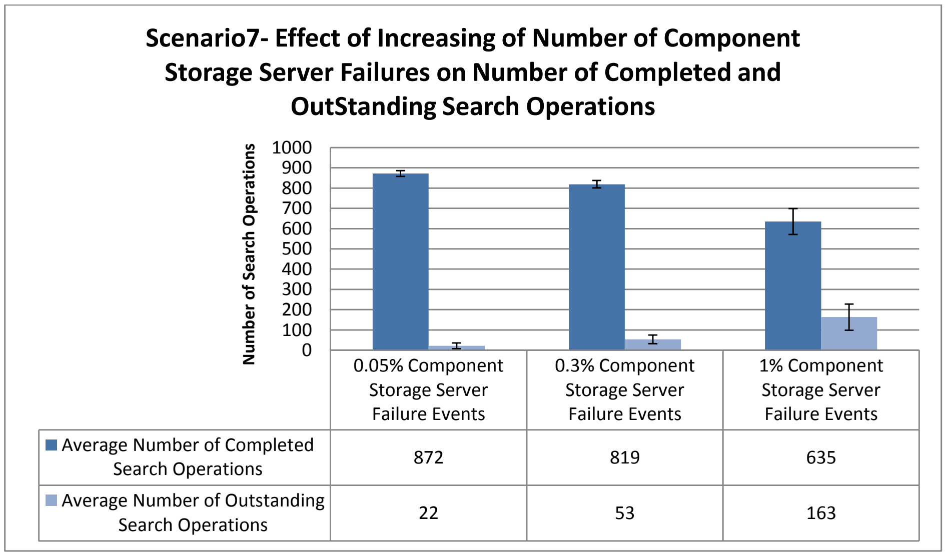
| | Experiment | | |
|---|---|---|---|
| | 1 | 2 | 3 |
| **Probability of CSS Failure Event** | 0.05% | 0.3% | 1% |
| **Average Download Time(seconds)** | 13.13 | 13.76 | 17.32 |
| **Standard Deviation** | 0.09 | 0.48 | 1.42 |
| **Average Number of Completed Search Operations** | 872 | 819 | 635 |
| **Average Number of Outstanding Search Operations** | 22 | 53 | 163 |



| | 0.05% Component Storage Server Failure Events | 0.3% Component Storage Server Failure Events | 1% Component Storage Server Failure Events |
|---|---|---|---|
| 🟦 Average Download Time | 13.13 | 13.76 | 17.32 |

Figure 5.12: Scenario 7 - Effect of Increasing the Probability of Component Storage Server Failures on Average Download Time when All Clients were Placed Uniform Randomly

As the results show in Figure 5.12, the average download time still increased, but this time there was an error bar overlap between the two left-hand bars. All of my experiments ran in a constant simulation time horizon. As a result, when the probability of component storage server failure events increased the number of completed search operations decrease. This means the average download time for experiment 3 was between fewer search operations. Figure 5.13 shows the number of completed and outstanding search operations for each case. As shown in the results for experiments 1 and 2 the difference in average download time is not significant and in a real world the 0.05% failure for component storage server is a more realistic. Thus, the results of the third experiment are unlikely to reflect actual behaviour in an implemented system so we can conclude that there is sufficient redundancy in the storage servers to effectively deal with a small number of failures.



| | 0.05% Component Storage Server Failure Events | 0.3% Component Storage Server Failure Events | 1% Component Storage Server Failure Events |
|---|---|---|---|
| ■ Average Number of Completed Search Operations | 872 | 819 | 635 |
| ■ Average Number of Outstanding Search Operations | 22 | 53 | 163 |

Figure 5.13: Scenario 7- Effect of Increasing of Number of Component Storage Server Failure on Number of Completed and Outstanding Search Operations

### 5.3.9 Scenario 8 - Increasing User Movement

#### Goal

One of today's important features for any system is to be accessible from anywhere at anytime. I designed scenario 8 to study the effect of user mobility on average download time.

#### Expectation

When a client moves from one location to a new location, and if the client has already downloaded its components from its nearby component storage server, the client will not need to download the components from its nearby component storage server until it has replaced the component and needs it again. Then the client needs to contact its new nearby cache location server (wherever it has moved to) to locate a replica of the requested component near its new location. The new nearby component storage server of the client may not have the component and then the client needs to download the components from a more distant component storage server. Then the client must upload the component to its new nearby component storage server. Therefore, user mobility should increase the download time for the first download after relocating. Thus, we can expect user mobility to increase the average download time but perhaps not greatly.

### Result

I ran three experiments for five times each in this scenario. In the first scenario the probability of user movements was 0.1%. The second experimental probability was 1% and for the third experiment the movement probability was 5% (a very high percentage of the population to be mobile at any time).

Table 5.16 and Figure 5.14 show the scenario 8 results.

Table 5.16: Scenario 8 - Effect of Increasing User Movement on Average Download Time

| | Experiment | | |
|---|---|---|---|
| | 1 | 2 | 3 |
| **Probability of User Mobility** | 0.1% | 1% | 5% |
| **Average Download Time(seconds)** | 5.79 | 6.40 | 8.18 |
| **Standard Deviation** | 0.05 | 0.16 | 0.10 |

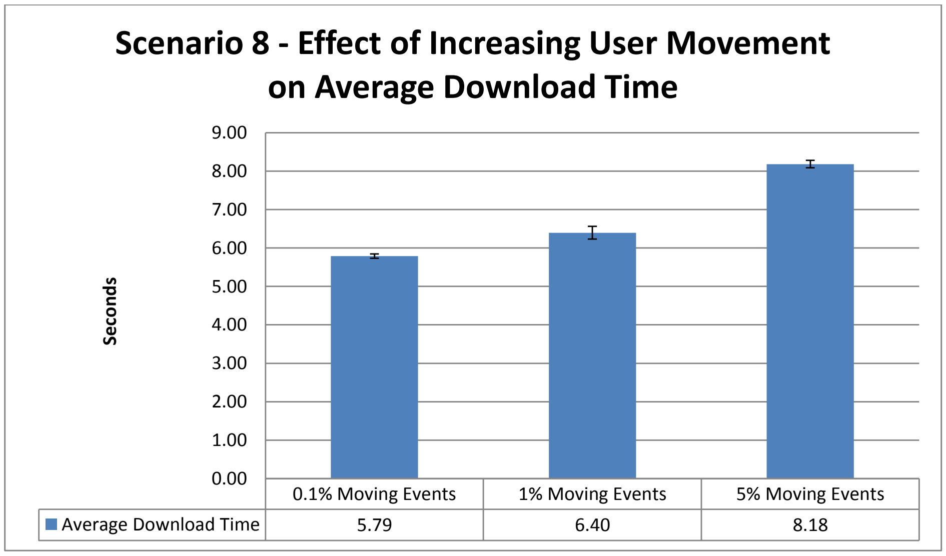


Figure 5.14: Scenario 8 - Effect of Increasing User Movement on Average Download Time

As surmised, movement does negatively impact average download time. The longer the time a mobile user stays in a given location, however, the less significant the impact should be.

### 5.3.10 Scenario 9 - Varying the Number of Clients in Dense Population Areas

#### Goal

It seems logical that dense population areas should benefit more from the caching effects of the system. Thus, for this scenario I wanted to study the average download time based on whether clients tended to be clustered in dense population areas or not.

### Expectation

If there are fewer clients in dense population areas, it means more clients will likely be accessing different cache location servers and component storage servers. Therefore, the probability of a client to download a component from its nearby component storage server the first time it needs it (because another client nearby has already, forced a replicas to be there) will be lower. On the other hand, if all clients are placed in dense population areas, the chance the client downloads its requested component from a nearby component storage server on its first access will be higher. Also, in dense population areas, cache location servers have more chance to already have information about requested components. Therefore, the average download time should be decreased with denser populations.

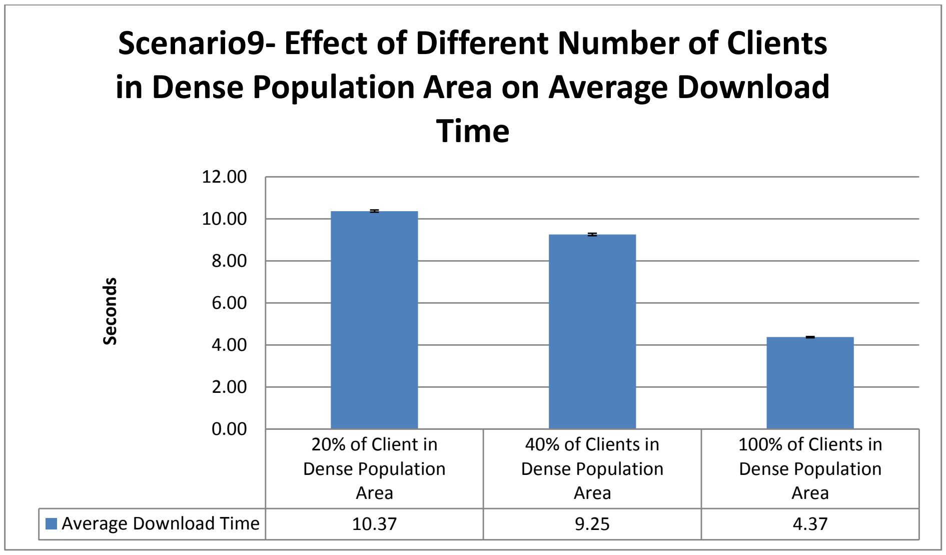
### Result

I ran scenario 9 with three different experiments, five times each. In the first experiment 20% of the clients were placed in dense population areas. In the second experiment, 40% of the clients were placed in dense population area and, finally, in the third experiment all clients were placed in the dense population areas.

Table 5.17 and Figure 5.15 shows the scenario 9 results.

Table 5.17: Scenario 9 - Effect of Varying Number of Clients in Dense Population Areas on Average Download Time

| | Experiment | | |
|---|---|---|---|
| | 1 | 2 | 3 |
| **Percentage of Clients in Dense Population Areas** | 20% | 40% | 100% |
| **Average Download Time(seconds)** | 10.37 | 9.25 | 4.37 |
| **Standard Deviation** | 0.05 | 0.06 | 0.03 |



| | 20% of Client in Dense Population Area | 40% of Clients in Dense Population Area | 100% of Clients in Dense Population Area |
|---|---|---|---|
| 🟦 Average Download Time | 10.37 | 9.25 | 4.37 |

Figure 5.15: Scenario 9 - Effect of Varying Number of Clients in Dense Population Areas on Average Download Time

The benefits of living in a densely populated area are clear and the more people in such areas the greater the benefit will be.

### 5.3.11 Scenario 10 - Varying Component Popularity

#### Goal

The goal of scenario 10 was to study the effect of different numbers of components by types (very popular, popular and non-popular) on average download time.

#### Expectation

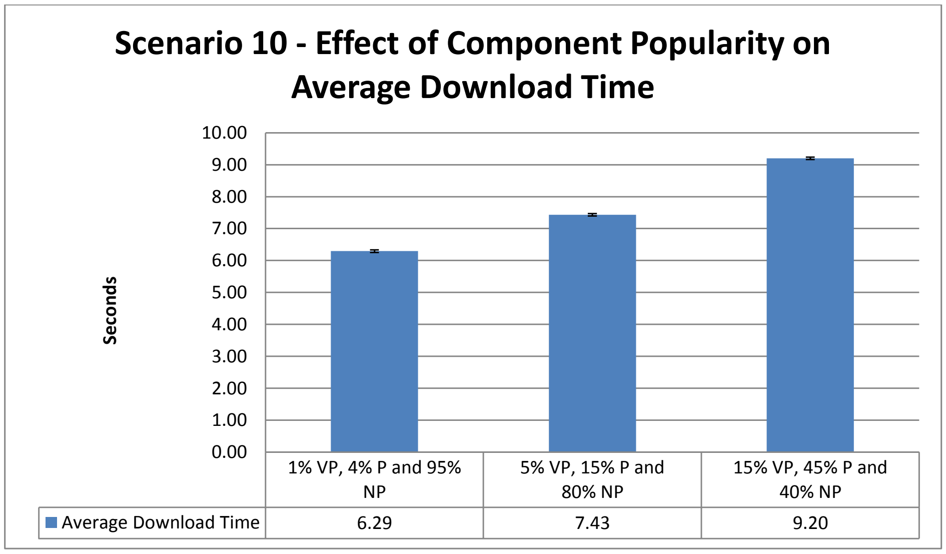
If there are relatively few very popular and popular components in the system, then the chance of cache location servers having the requested components' location in their databases is higher. Also, in dense population areas, clients have a better chance to download the component from a nearby component storage server on a first request. Thus, with relatively few very popular/popular components efficiency in accessing those components should be high. Because they are "popular" there will be many accesses to them so the average access time should be low. If the number of very popular and popular components is higher, however the average download time will be likely increase because the limited resources of the system will have to be spread across more, frequently accessed, components.

#### Result

I ran this scenario with three different experiments, five times each. In the first experiment, I had 1% very popular, 4% popular and 95% non-popular components (probably a realistic choice). Then for the second experiment, there were 5% very popular components, 15% popular components and 80% non-popular components. And finally in the third experiment, there were 15% very popular components, 45% popular components and 40% non-popular components. Table 5.18 and Figure 5.16 show the scenario 10 results.

Table 5.18: Scenario 10 - Effect of Varying Component Popularity on Average Download Time

| | Experiment | | |
|---|---|---|---|
| | 1 | 2 | 3 |
| **Components Category Distribution (VP,P,NP)** | 1%, 4%, and 95% | 5%, 15%, and 80% | 15%, 45%, and 40% |
| **Average Download Time(seconds)** | 6.29 | 7.43 | 9.20 |
| **Standard Deviation** | 0.04 | 0.04 | 0.03 |



| | 1% VP, 4% P and 95% NP | 5% VP, 15% P and 80% NP | 15% VP, 45% P and 40% NP |
|---|---|---|---|
| ■ Average Download Time | 6.29 | 7.43 | 9.20 |

Figure 5.16: Scenario 10 - Effect of Varying Component Popularity on Average Download Time

As seen in Figure 5.16 the average access time does decrease with a high percentage of very popular/popular components. Fortunately, this is unlikely to occur in practice. There is much software available but relatively few truly popular programs (e.g. Firefox, MS Word, Adobe Acrobat, etc.) Much of the software created is simply too special purpose to be generally popular.

### 5.3.12 Scenario 11 - Varying the Number of Component Storage Servers

#### Goal

In this scenario I wanted to study what effects would occur if I decrease the number of component storage servers in the system. Again, the key metric is average download time.

### Expectation

If there are more component storage servers in the system, it means that clients can download their requested components from component storage servers that are more likely to be nearby. Therefore, the average download time should be decreased.

### Result

I ran this scenario with two sets of configurations with four experiments, five times each. In the first configuration, I had 400 Component Storage Servers, 80 Component Storage Servers, 16 Component Storage Servers and 4 Component Storage Servers, respectively with only 3 components (one very popular, one popular and one non-popular). Table 5.19 and Figure 5.17 show the scenario 11 results with just the 3 components.

Table 5.19: Scenario 11 - Effect of Varying the Number of Component Storage Servers on the Average Download Time with 3 Components

| | Experiment | | | |
|---|---|---|---|---|
| | 1 | 2 | 3 | 4 |
| **Number of CSS** | 400 | 80 | 16 | 4 |
| **Average Download Time(seconds)** | 8.06 | 8.26 | 8.41 | 7.94 |
| **Standard Deviation** | 0.08 | 0.05 | 0.06 | 0.06 |

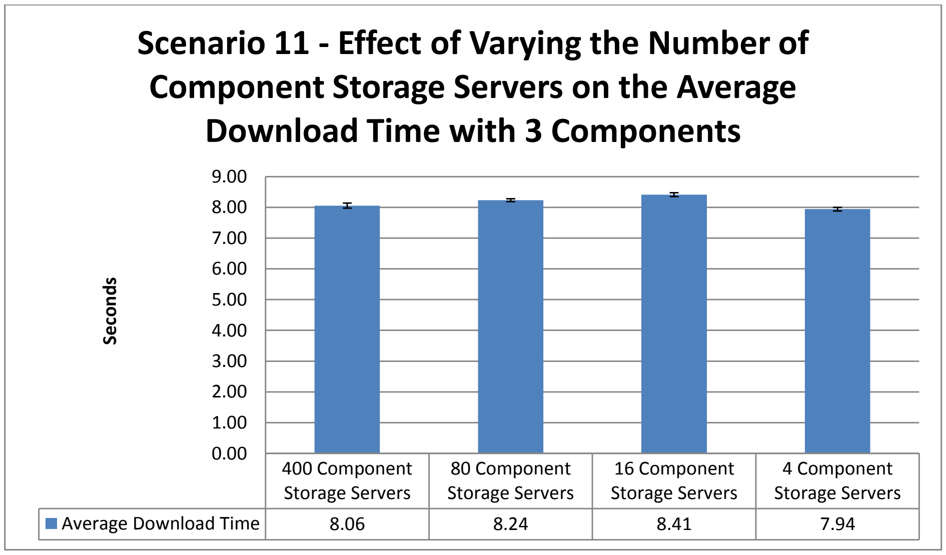
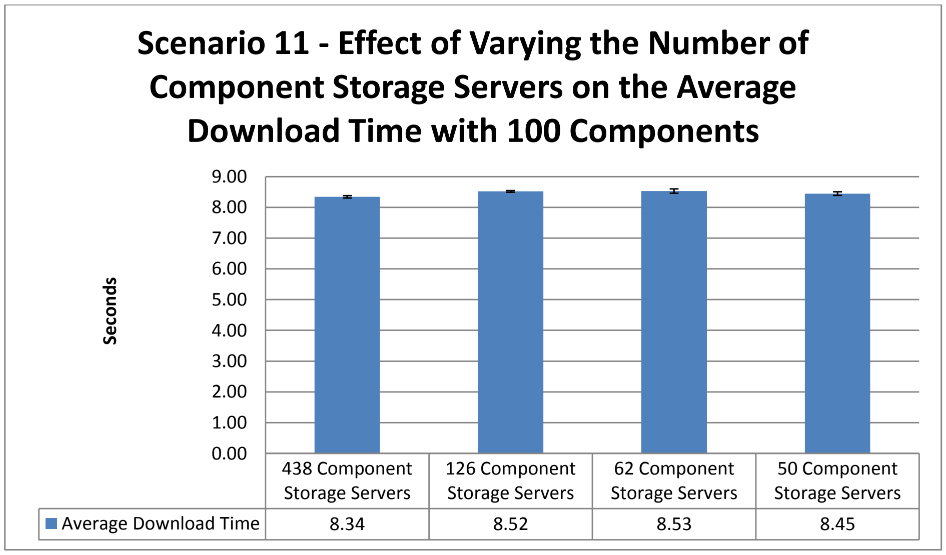


Figure 5.17: Scenario 11 - Effect of Varying the Number of Component Storage Servers on the Average Download Time with 3 Components

I then ran this scenario again with 438 Component Storage Servers, 126 Component Storage Servers, 62 Component Storage Servers and 50 Component Storage Servers and now with 100 components. Table 5.20 and Figure 5.18 show the scenario 11 results with 100 components.

Table 5.20: Scenario 11 - Effect of Varying the Number of Component Storage Servers on the Average Download Time with 100 Components

| | Experiment | | | |
|---|---|---|---|---|
| | 1 | 2 | 3 | 4 |
| **Number of CSS** | 438 | 126 | 62 | 50 |
| **Average Download Time(seconds)** | 8.34 | 8.52 | 8.53 | 8.45 |
| **Standard Deviation** | 0.04 | 0.03 | 0.07 | 0.06 |



| | 438 Component Storage Servers | 126 Component Storage Servers | 62 Component Storage Servers | 50 Component Storage Servers |
|---|---|---|---|---|
| ■ Average Download Time | 8.34 | 8.52 | 8.53 | 8.45 |

Figure 5.18: Scenario 11 - Effect of Varying the Number of Component Storage Servers on the Average Download Time with 100 Components

As expected when the number of component storage servers is decreased there is less chance for a client to download the component from a nearby component storage server. Thus, the average download time is increased. This effect becomes clearer when the simulation runs for a longer time. In these scenario, when there are few, experiment 4 with 4 component storage servers in the first configuration and experiment 4 with 50 component storage servers in the second configuration, the average download time is decreased (which was not expected). The reason for this behaviour is the relationship between the size of the components, the bandwidth between nodes and the distance between nodes. As described before, the maximum distance between any nodes in the simulations was 380km. Given that realistic latencies and bandwidths were used in the simulation, the message transit times are thus comparatively small, so the impact of accessing a more remote component storage server is not great. Further, when there are few component storage servers in the system, there is more chance for a given cache location server to already have information about the required component storage servers' ID. Therefore, cache location servers do not need to contact the master location servers which decreases the download time of the component.

### 5.3.13 Scenario 12 - Varying the Number of Cache Location Servers

#### Goal

In this scenario I wanted to study the effect of varying the number of cache location servers on average download time.

#### Expectation

If there are more cache location servers in the system, then clients can quickly send their requests to a nearby cache location server and get a fast response. This means that decreasing the number of cache location servers, should increase the average download time because the cache location server used could be further from the client. But when there are more cache location servers in the system, while access to the cache location server may be quicker, the chance that a client connects to a cache location server which another client used for the same component is going to decreased resulting in lower "cache hit rate". Therefore, the cache location servers' database have fewer entries (at least at the beginning of the simulation) and therefore cache location servers need to connect to master location servers more frequently (incurring the associated overhead). Therefore when this happens, the average download time may be decreased by increasing the number of cache location servers.

**Result**

I ran this scenario with two sets of configurations with five different experiments, for five times each for the first configuration and **ten** times each for the second configuration. In the first configuration, I had 1000 Cache Location Servers, 80 Cache Location Servers, 20 Cache Location Servers, 8 Cache Location Servers and 4 Cache Location Servers. There were 3 components (one very popular, one popular and one non-popular) and the number of operations was 10000 with 5000 clients. The simulation time horizon was 15000 seconds. Table 5.21 and Figure 5.19 show the scenario 12 results with 3 components.

Table 5.21: Scenario 12 - Effect of Varying the Number of Cache Location Servers on Average Download Time with 3 Components

| | Experiment | | | | |
|---|---|---|---|---|---|
| | 1 | 2 | 3 | 4 | 5 |
| **Number of CLS** | 1000 | 100 | 20 | 8 | 4 |
| **Average Download Time(seconds)** | 8.82 | 8.46 | 8.48 | 7.97 | 8.38 |
| **Standard Deviation** | 0.09 | 0.06 | 0.06 | 0.03 | 0.07 |

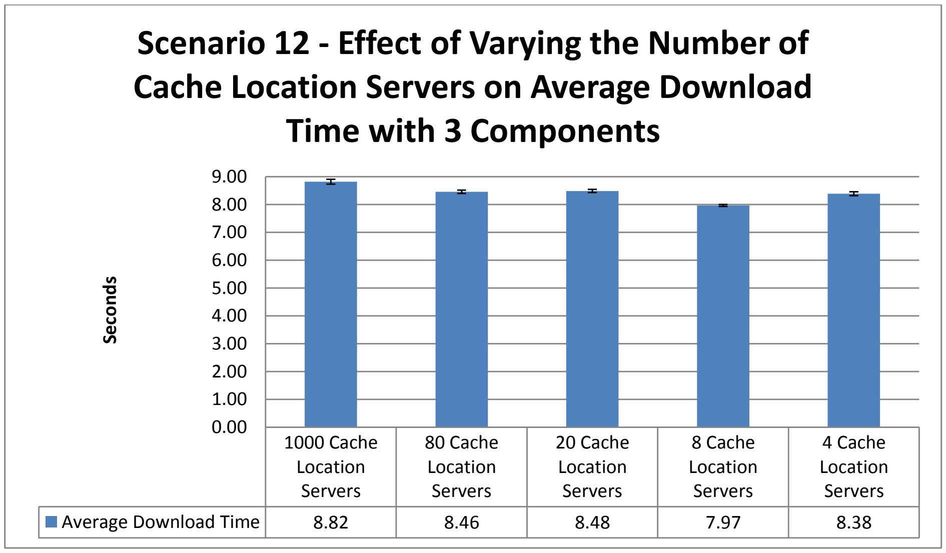
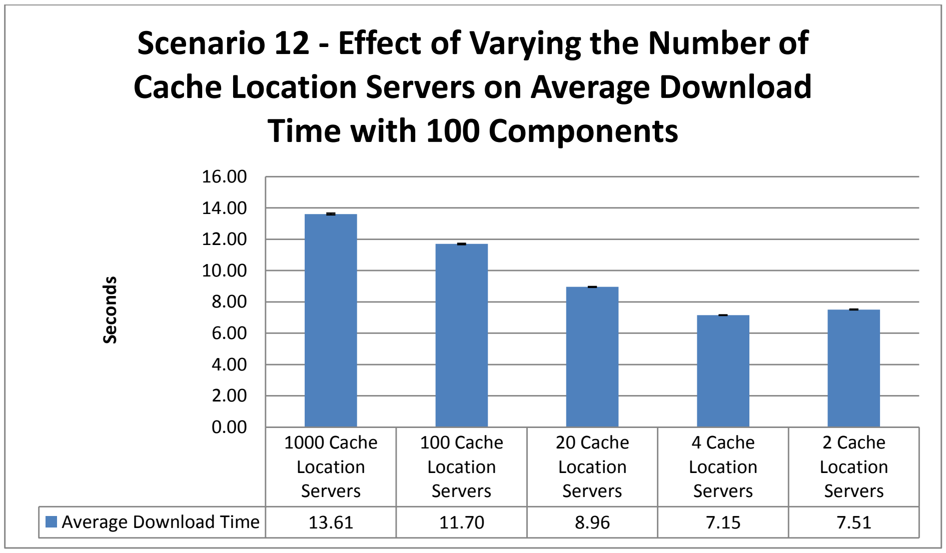


Figure 5.19: Scenario 12 - Effect of Varying the Number of Cache Location Servers on Average Download Time with 3 Components

I then ran this scenario again with 1000 Cache Location Servers, 100 Cache Location Servers, 20 Cache Location Servers, 4 Cache Location Servers and 2 Cache Location Servers with 100 components and 5000 clients. There were 20000 operations in 35000 seconds. Table 5.22 and Figure 5.20 show the scenario 12 results with 100 components.

Table 5.22: Scenario 12 - Effect of Varying the Number of Cache Location Servers on Average Download Time with 100 Components

| | Experiment | | | | |
|---|---|---|---|---|---|
| | 1 | 2 | 3 | 4 | 5 |
| **Number of CLS** | 1000 | 100 | 20 | 4 | 2 |
| **Average Download Time(seconds)** | 13.61 | 11.70 | 8.96 | 7.15 | 7.51 |
| **Standard Deviation** | 0.05 | 0.03 | 0.02 | 0.02 | 0.02 |



| | 1000 Cache Location Servers | 100 Cache Location Servers | 20 Cache Location Servers | 4 Cache Location Servers | 2 Cache Location Servers |
|---|---|---|---|---|---|
| ■ Average Download Time | 13.61 | 11.70 | 8.96 | 7.15 | 7.51 |

Figure 5.20: Scenario 12 - Effect of Varying the Number of Cache Location Servers on Average Download Time with 100 Components

As the simulation results show for both configurations, decreasing the number of cache location servers will actually decrease the average download time until there are very few cache location servers in the system when the lack of locally available information outweighs overhead.

# Chapter 6

# Conclusion

## 6.1 Summary and Contributions

In my thesis research, I designed a placement algorithm exploiting information about previous accesses to components to decide where best to place new component replicas and also a location-aware component lookup algorithm. I then implemented and tested a simple Java-based prototype built using these algorithms. I then developed a simulation system to assess the prototype's reliability, performance and scalability. After running a number of simulation experiments, using a scaled down model of an Internet-like environment, I conclude that my system performs well (assuming the constraints of the simulations done), providing good performance and reliability in the face of system failure and user mobility.

## 6.2 Future Work

There are a number of questions that arose in this thesis that could be areas for future work.

The first, obvious one is running the simulation on a larger scale. I can create a network topology with more routers with longer distances between them. Then, I can run the simulation for more clients and components over a longer simulation time to build confidence in my belief that my system will scale well to very large deployments. This would confirm the consistency of my simulation results between the small and large scale.

Another area of future might be including different devices with different network properties (e.g. 3G networks [LW02]) into the simulations to assess their impact on performance of the system.

In the longer term, I would like to do further, more detailed studies on network failure like failure of routers and network congestion to assess the impact of such failures on the system.

A couple of final areas for future work would be simulations with different system properties like search queries times on different databases to assess their impact as well.

Finally, a real deployment using something like PlanetLab [PLA] would allow confirmation of simulation assumptions.

# Bibliography

[AR99] Barabasi A. and Albert R. Emergence of Scaling in Random Networks. *SCIENCE*, 286:509, 1999.

[AWSBL00] W. Adjie-Winoto, E. Schwartz, H. Balakrishnan, and J. Lilley. The Design and Implementation of an Intentional Naming System. *SIGOPS Oper. Syst. Rev.*, 34:22–, April 2000.

[Car08] M. J. Carey. Soa What? *IEEE Computer*, 41(3):92–94, 2008.

[CD88] G. F. Coulouris and J. Dollimore. *Distributed Systems Concepts and Design*. Addison-Wesley, 1988.

[CDZ97] K.L. Calvert, M.B. Doar, and E.W. Zegura. Modeling Internet Topology. *Communications Magazine, IEEE*, 35(6):160 –163, June 1997.

[CIS] `http://www.boson.com/netsim-cisco-network-simulator`.

[Ciu09] E. Ciurana. *Developing with Google App Engine.* Apress, Berkeley, CA, USA, 2009.

[CLA] `http://docs.oracle.com/javase/1.4.2/docs/api/java/lang/ClassLoader.html`.

[CS01] W. L. Currie and P. Seltsikas. Exploring the Supply-side of IT Outsourcing: Evaluating the Emerging Role of Application Service Providers. *Eur. J. Inf. Syst.*, 10(3):123–134, 2001.

[Cus10] M. Cusumano. Cloud Computing and SaaS as New Computing Platforms. *Commun. ACM*, 53(4):27–29, 2010.

[DC] `http://publib.boulder.ibm.com/infocenter/txformp/v6r0m0/index.jsp?topic=%2Fcom.ibm.cics.te.doc%2Ferziaz0015.htm`.

[GS96] R. Guerraoui and A. Schiper. Fault-Tolerance by Replication in Distributed Systems. In *Proceedings of the 1996 Ada-Europe International Conference on Reliable Software Technologies*, Ada-Europe '96, pages 38–57, London, UK, UK, 1996. Springer-Verlag.

[HEHB08] T. Hau, N. Ebert, A. Hochstein, and W. Brenner. Where to Start with SOA: Criteria for Selecting SOA Projects. In *HICSS '08: Proceedings of the Proceedings of the 41st Annual Hawaii International Conference on System Sciences*, page 314, Washington, DC, USA, 2008. IEEE Computer Society.

[HKW$^+$08] J. Hutchinson, G. Kotonya, J. Walkerdine, P. Sawyer, G. Dobson, and V. Onditi. Migrating to SOAs by Way of Hybrid Systems. *IT Professional*, 10(1):34 –42, jan. 2008.

[ILM09] N. Ibrahim and F. Le Mouel. A Survey on Service Composition Middleware in Pervasive Environments. *International Journal of Computer Science Issues, IJCSI*, 1:1–12, 2009.

[KKLL09] W. Kim, S. D. Kim, E. Lee, and S. Lee. Adoption Issues for Cloud Computing. In *MoMM '09: Proceedings of the 7th International Conference on Advances in Mobile Computing and Multimedia*, pages 2–5, New York, NY, USA, 2009. ACM.

[KV04] V. Koutsonikola and A. Vakali. LDAP: Framework, Practices, and Trends. *Internet Computing, IEEE*, 8(5):66 – 72, 2004.

[Lev99] J. R. Levine. *Linkers and Loaders.* Morgan Kaufmann; 1st edition, 1999.

[LW02] J. Liu and K. P. Worrall. Theory and Practice in 3G Network Planning. In *3G Mobile Communication Technologies, 2002. Third International Conference on (Conf. Publ. No. 489)*, pages 74–80, 2002.

[MD95] P. V. Mockapetris and K. J. Dunlap. Development of the Domain Name System. *SIGCOMM Comput. Commun. Rev.*, 25:112–122, January 1995.

[MLMB01] A. Medina, A. Lakhina, I. Matta, and J. Byers. BRITE: Universal Topology Generation from a User's Perspective. Technical report, Boston, MA, USA, 2001.

[MMB00] A. Medina, I. Matta, and J. Byers. On the Origin of Power Laws in Internet Topologies. *SIGCOMM Comput. Commun. Rev.*, 30(2):18–28, April 2000.

[Moc87a] P. V. Mockapetris. IETF RFC 1034: Domain Names-Concepts and Facilities, 1987.

[Moc87b] P. V. Mockapetris. IETF RFC 1035: Domain Names-Implementation and Specification, 1987.

[Nit09] Nitu. Configurability in SaaS (Software as a Service) Applications. In *ISEC '09: Proceedings of the 2nd India Software Engineering Conference*, pages 19–26, New York, NY, USA, 2009. ACM.

[OL88] B. M. Oki and B. H. Liskov. Viewstamped Replication: A New Primary Copy Method to Support Highly-Available Distributed Systems. In *Proceedings of the Seventh Annual ACM Symposium on Principles of Distributed Computing*, PODC '88, pages 8–17, New York, NY, USA, 1988. ACM.

[PE05] L'Ecuyer P. and Buist E. Simulation in JAVA with SSJ. In *Proceedings of the 37th Conference on Winter Simulation*, WSC '05, pages 611–620. Winter Simulation Conference, 2005.

[PLA] `http://www.planet-lab.org/`.

[SKG09] B. Sajed Khosrowshahi and P. Graham. Component Placement and Location for a Dynamic Software Composition System. In *Proceedings of the 2nd Canadian Conference on Computer Science and Software Engineering*, C3S2E '09, pages 127–130, New York, NY, USA, 2009. ACM.

[SPM+06] J. M. Schopf, L. Pearlman, N. Miller, C. Kesselman, I. Foster, M. D'Arcy, and A. Chervenak. Monitoring the Grid with the Globus Toolkit MDS4. *Journal of Physics: Conference Series*, 46(1):521, 2006.

[TB01] Z. Tari and O. Bukhres. *Fundamentals of Distributed Object Systems: The CORBA Perspective.* John Wiley & Sons, Inc., New York, NY, USA, 2001.

[Wax88] B.M. Waxman. Routing of Multipoint Connections. *Selected Areas in Communications, IEEE Journal on*, 6(9):1617 –1622, dec 1988.

[WDLW09] Z. Wu, S. Deng, Y. Li, and J. Wu. Computing Compatibility in Dynamic Service Composition. *Knowl. Inf. Syst.*, 19(1):107–129, 2009.

[WPS$^+$00] M. Wiesmann, F. Pedone, A. Schiper, B. Kemme, and G. Alonso. Understanding Replication in Databases and Distributed Systems. In *Distributed Computing Systems, 2000. Proceedings. 20th International Conference on*, pages 464 –474, 2000.