Universidade do Minho
Escola de Engenharia

Francisco António Ferraz Martins de Almeida Maia

Bolt Cloud – A Working Set Dissemination Service

Novembro 2009

Universidade do Minho
Escola de Engenharia

Francisco António Ferraz Martins de Almeida Maia

Bolt Cloud – A Working Set Dissemination Service

Mestrado em Engenharia Informática

Trabalho efectuado sob a orientação do
Doutor Rui Carlos Mendes de Oliveira

Novembro 2009

Acknowledgements

To all of my family, which has always supported me along these years and is always there whenever I need. Specially my parents António e Rosalina, my brothers Luís and José and my little sister Margarida. I also want to thank my cousins that always help me to look and go forward: João, Madalena, Ana, Clara, Nuno, Maria, Maria M, Paulo, António, João M, Dora, Francisca and Xavier.

To Professor Rui Oliveira for his guidance throughout the thesis and for his capacity to inspire.

To Professor Francisco Moura for his help reviewing the thesis.

To Professor José Orlando Pereira, Professor António Luís Sousa and Professor Carlos Baquero for their continuous help and patience to answer every question.

To my uncle Professor José M. da Silva and my cousin João Maia for their help reviewing the thesis.

To Francisco Cruz and João Paulo for all the help, critics and all the non serious stuff.

To Nuno Carvalho, Ricardo Vilaça, Bruno Costa, Miguel Matos, Paulo Jesus, Ana Nunes, Filipe Campos and Nuno Castro for their help and for the outstanding working environment we have at the Distributed Systems Lab. I want to thank specially to Nuno Carvalho and Ricardo Vilaça for their precious help and Bruno Costa for his constructive criticism.

To all of my friends who really help me be who I am.

Finally, I want to thank the lunch break fantastic discussions to my friends: Fábio, Granja, Nuno, Miguel Pires, Carlos and Miguel.

Resumo

Utilizando os diversos aparelhos computacionais actualmente ao nosso dispor, constantemente consultamos, editamos e criamos ficheiros. Com facilidade notamos que, de entre todos esses ficheiros, existe um conjunto que consideramos mais importante e que mais utilizamos. Verificamos ainda que, ainda que desejável, é difícil manter este conjunto de ficheiros acessível constante e coerentemente em vários aparelhos e sítios ao mesmo tempo. Para resolver este problema têm surgido um conjunto de novos serviços. Chamamos-lhes neste trabalho *"Working Set Dissemination Services"*.

Nesta dissertação descreve-se uma estratégia de implementação de "Working Set Dissemination Service". São analisados problemas decorrentes da grande quantidade de possíveis utilizadores de um serviço deste tipo e são propostas melhorias significativas dos serviços já existentes.

Abstract

When using all the different computing devices available to us nowadays we constantly access, edit and create files. We can easily notice that we have a working set of files which are the most important and most intensively used. Often, we want these files to be accessible in different devices and places but it is difficult to keep the files synchronized and up to date everywhere. To make synchronization of files between different devices easier a new set of services has been arising. We call these services *Working Set Dissemination Services*. Throughout this work we are going to describe our approach to implementing one of these services. We will be looking at the problems raised by the large amount of prospective users of this kind of service and at some improvements that can be made to existing solutions.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

One keyword we should emphasize when characterizing our society is *connectivity*. Nowadays, staying connected is very important as much for personal as for business relationships. The Internet enables easy and instant interaction and new services are always emerging. People see these services as an answer to their natural needs and tend to massively adhere to them.

With the diversity of services offered and the different devices at our disposal people tend to *digitalize* their lives. In addition, people's mobility creates the need to find ways to deal with all the data they access, edit or produce. In fact, the same data is often replicated through various locations. Immediate evidence of this can be found in our personal computers. Typically, we have a set of files, that compose our working set. These files are those we access most frequently and the ones we tend to carry around. Moreover, we need to have this set of files accessible in various locations or devices, for example, in our home computer, our work station and our laptop.

Adding to the problem of having the files available in all of our computers, is the problem of having them in a consistent state. Considering we have the files replicated through different computers, if we edit a file in one computer and then access the same file in an other one, without consistency guarantees, we may arrive at a point

where we have two different versions of the file and none of them is really the version we expected. Thus, making these files available in different under use computers is important but data consistency guarantees must be provided.

We will call the problem of having this working set of files available in our different computers in a consistent state the Working Set Dissemination Problem. This problem has been addressed and some of the approaches made to solve it represent a new set of services that are being offered. These services, in general, offer the ability to have our working set of files synchronized across our personal computers as well as with some reliable storage remotely located. We will call these services Working Set Dissemination Services.

One of the main challenges when trying to develop a Working Set Dissemination Service is ensuring that the working set of files is replicated consistently across all the user's computers. One trivial way to achieve data consistency, considering this type of problem, would be to have the files in a single location and provide a remote access to every computer. Being able to access the files everywhere is very important and if the files were stored in a reliable manner we could have guarantees against data loss. However, the files would not be accessible without Internet connectivity and we would loose the ability to access the files arbitrarily in our normal working environment. Therefore, Working Set Dissemination Services typically work over actual replicas of the set of files distributed by our different computers and the remote reliable storage. The challenge materializes now in being able to efficiently synchronize the different replicas and provide an adequate consistency model regarding the intended usage of the system.

It is also important to notice that a Working Set Dissemination Service, being an answer to an everyday life necessity has a very large number of potential users and thus scalability issues will arise. In fact, if we consider millions of users we must expect a very large

amount of data to manage.

With this work we propose a system architecture and a set of APIs to implement a Working Set Dissemination Service. In Section 2 we present some of the different existing approaches to the working set dissemination problem and these will be used as a starting point to our own approach. Our approach, besides trying to solve the working set dissemination problem, proposes improvements to existing solutions meant to enhance the availability of the data.

1.1 Exploiting workspace collocation

When looking at present solutions to the working set dissemination problem we can see that they obey to the client/server model. Nowadays, the client/server model is implemented over the Cloud Computing model and examples of this are Amazon Web Services [3], Google's App Engine [9] and Microsoft's Windows Azure [21].

Cloud Computing refers to an abstraction of infrastructure and software, which are then seen as a service [24]. This service exports an API to interact with it. The client knows nothing about the Cloud and how the service is implemented but knows how to use it. As the service is transparent to the client it is seen as an unlimited source of resources (computational or storage).

Even though the Cloud Computing model helps the implementation of a Working Set Dissemination Service by providing an infrastructure with elastic resources[1], the data produced in one computer is sent to the server or cloud and then from the server to the other computers. Considering a very large number of users, this model can be improved if we take into account some locality factors.

[1]The infrastructure is elastic as its capabilities grow on demand. Whenever the application needs another server or storage is made available. This mechanism gives the application the illusion of infinite resources and such infrastructure can respond to virtually any load scenario.

These locality factors can be:

- Users that have more than one computer connected in the same network.
- Different users behind the same IP address, users from the same University or Institution.

We meant to explore these observations and try to optimize the synchronization process between computers. In fact, even considering a server with infinite resources we are still limited by data transfer rates. Therefore, if we minimize data transference between server and clients and are able to take advantage of local area network transfer rates, we can provide a better service. Furthermore, today personal computer capabilities can be used as a complement to the server side computing effort.

The first locality factor we pointed out comes from the observation that often the potential users of a working set dissemination service have more than one computer connected in the same network, typically at home. With the objective of synchronizing the working set of files between those computers, the approach taken by current Working Set Dissemination Services is to send changes made in one computer to the server and then from the server to other computers. With our work we propose that the step of downloading data from the server is avoided by resorting to the local network.

Looking at the second locality factor, let us imagine we have a certain number of users of the Working Set Dissemination service in the same university. In this case we can not synchronize the different computers on the network because they belong to different users with different sets of files. We can however have a pro-active approach relying on the different users collaboration. We can notice that these users share most of their working time together. If each user dedicates some disk space to the service we can disseminate the working set of other users to that space. This way that space is

used as a cache of the remote storage the service provides. Suppose a user is at home and uploads a file to the system. If the system knows a colleague at the user's university it can send the file to that colleague's computer. When the user arrives at the university and connects the laptop the file will be downloaded directly from the colleague's computer saving time and reducing file requests to the remote storage.

With this work we propose a system that takes these observations into account to deliver an improved Working Set Dissemination Service.

1.2 Structure of the thesis

This dissertation is organized as the following. In Chapter 2 we will describe some existing work about the problem of working set dissemination by describing available solutions. Chapter 3 is dedicated to the contributions of our work, namely, the architecture, protocols and prototype. We evaluate our system in Chapter 4 and conclude with Chapter 5.

Chapter 2

Related Work

In this chapter current approaches and solutions to some of the problems and challenges pertaining to Working Set Dissemination Services are reviewed. We first overview existing systems enabling file backup, sharing and synchronization and then we conclude with the analysis of emerging services for massive data storage particularly suited to be deployed under the *Cloud Computing* model.

2.1 Existing Applications

Currently, several applications and services aimed at file sharing and backup through the remote storage of files are available. Typically these:

- Offer remote storage.
- Allow ubiquitous access to the data stored through web access.
- Synchronize files or folders across different devices.
- Offer a remote backup solution.

We divide these applications into categories. This will help to understand differences between them and also to position our own approach amongst them.

2.1.1 Sharing Applications

In the first category we will place the applications that offer a way to store files on a server and to share them with other users. This process is done manually file by file. Examples of these are RapidShare [15], MegaUpload [11] and MegaShares [10].

These applications are suitable to share files with everyone when those files are bigger than email attachments constraints. However, they do not represent a very good solution to the Working Set Dissemination problem, as the user must upload the files every time she changes location and carry around links to those files for later download. Moreover, the files are deleted by the service if the user does not download them for a period of time.

2.1.2 Backup Applications

In this second category we place applications for which data backup is the main focus. It is important to notice that the Working Set Dissemination problem is quite different from the data backup problem. However, services from this category can be a building block to other more complex services. The following are representative examples of this category:

- Mozy [13] - Client/Server backup.
- OpenDrive [14] - Client/Server backup.
- Carbonite [6] - Client/Server backup.
- Box.net [5] - Client/Server backup.
- Pastiche [28] - Peer-to-Peer backup.
- Flashback [29] - Peer-to-Peer backup.

These applications differ on architecture and usage methods.

Mozy, OpenDrive and Box.net are examples of a set of applications that offer a certain amount of remote disk space where one can backup our files. There is no concern of keeping files synchronized between various locations. These services can be automatic: Mozy, OpenDrive or Carbonite, or manual: as it is Box.net or iDisk (MobileMe [12]).

Then we have Pastiche and Flashback. These two are intended to work in Personal Area Networks. The idea is that in a set of peers, each peer shares some amount of disk space with the others. This space is used by the system to backup other peers' data. This approach has the advantage of avoiding remote dedicated storage and takes advantage of disk space that is often wasted.

2.1.3 Working Set Dissemination Applications

Considering now the last category we will group here the applications that try to solve the Working Set Dissemination problem as a whole. These applications, in general, offer a way to synchronize a folder or a set of files across multiple devices. They also keep a copy of the files and folders in remote storage. These applications differ on the number of devices they support and in extra features like sharing permissions or the way they deal with media files. Examples are:

- DropBox [7]
- SugarSync [17]
- LiveMesh [22]
- Syncplicity [19]
- SpiderOak [16]
- Ubuntu One [20]

This last category is the one we will use as the basis for our work. We extract from these applications the common features they offer and then try to explore ideas from other applications such as those from the second category. These common features are:

- To provide a remote storage to host a copy of our working set
- To synchronize our working set across different computers
- To provide access to the files everywhere

In order to better differentiate these applications we depict their main features in Table 2.1. A more complete table can be found in [18] . The table enumerates some features and then marks the presence/absense of those features in each application. The features considered are:

- File backup with recovery.
  The application provides a way to backup files and a way to recover those files in case local copies are lost.
- Real-time update.
  The application synchronizes the system automatically and right after any change is made. Applications that do not provide this feature rely on manual synchronization.
- Versioning with the ability to restore.
  Besides backing up the current version of the files, the applications stores files' old versions and allows the user to restore any of those versions.
- Sync files on multiple computers.
  Synchronization between different computers is automatic. Those applications that do not present this feature must manually retrieve a copy of the files in each computer.
- Access files from a web browser.
  The application provides a web interface to access stored files.

- Share folders.
  The user can share files with other people directly from the system supporting the application.
- Share folders with permissions and password.
  To the previous feature is added an access control mechanism.
- XP and Vista Support, Mac OS X Support, Linux Support.
  The application supports the specified operating system.

2.2 Massive Scalable Storage Systems

With the emergence of Cloud Computing, service providers had to find ways to deal with huge amounts of data. The traditional relational databases are not able to cope with such quantity of data and new storage systems are being developed. With regard to the Working Set Dissemination problem, we also have to be concerned about the massive data storage required to provide the service. Namely, we need to know where to store the user's files. The ideal solution when implementing a Working Set Dissemination Service would be to have an independent reliable and scalable system providing storage related services.

From the existing massive storage services we identified Amazon's S3 [1] as the only one being an immediate answer to the Working Set Dissemination Service needs. In fact, it is the only service capable of storing arbitrarily large objects. Amazon's Simple Storage Service (S3) is an object store that provides a reliable, scalable and fast service. In S3 one can store an unlimited number objects until 5 gigabytes in size. Each object can have metadata associated. Objects are created, deleted or retrieved through a SOAP or REST API and are organized into buckets which are object containers. This service is, as it was intended, very simple but can free developers from scalability issues when dealing with large amounts of data.

Caracteristic	**SugarSync**	**DropBox**	**MobileMe**	**Carbonite**
File backup with recovery	✓	✓	✓	✓
Real time upload of changes	✓	✓	-	-
Versioning with the ability to restore	✓	✓	-	✓
Sync files on multiple computers	✓	✓	-	-
Access files from a web browser	✓	✓	✓	✓
Share folders	✓	✓	-	-
Share folders with permissions and password	✓	-	-	-
XP and Vista Support	✓	✓	-	✓
Mac OS X Support	✓	✓	✓	✓
Linux Support	-	✓	-	-

Table 2.1: Features of existing Working Set Dissemination applications.

We studied other massive scalable systems. Among these are Google's Bigtable [26], Amazon's SimpleDB [2] and PNuts [27]. These storage systems are not suitable to the implementation of Working Set Dissemination Service because their intended to store small tuples. However, it is important to notice that the abstraction level these services provide and their APIs could be ported to a service where storing large objects is a requisite. This would lead to a new set of services competing with Amazon's S3 and suitable to the Working Set Dissemination requisites.

Chapter 3

Bolt Cloud

Bolt Cloud, a novel approach to developing a scalable, efficient and modular Working Set Dissemination Service, is described in this chapter. Section 3.1 discusses *Bolt Cloud*'s architecture and how it solves the Working Set Dissemination Problem. In Section 3.2 we describe the *FlashSync* system, an innovative solution combining the client-server and peer-to-peer paradigms to improve the availability and efficiency of the service. The description of our prototype closes this chapter.

3.1 A Working Set Dissemination Service

As mentioned earlier, a Working Set Dissemination Service is a service that allows a user to have a set of files, typically a folder, synchronized across every computer the user owns. Additionally, the service keeps a safe and reliable copy of the set of files.

Bolt Cloud addresses common features of Working Set Dissemination Services, notably:

- Synchronization of a folder among the user's computers.
- Keeping a copy of the user's files in a dependable manner.

- Providing a conflict detection system for concurrent changes made to the files.
- Relying on a non-intrusive interface that minimizes the need to have user interaction.
- Being a modular system.
- Designed for personal use only.

It is important to emphasize the last feature on the list. *Bolt Cloud* is for personal use only. With personal use we mean that for each account there is only one user. Each user can have more than one machine connected but changes to the working set of files are expected to be made to only one machine at a time. Following this usage model and admitting each machine to be always online, each change made to the files would be disseminated to all machines within a certain period of time. If that period of time was always smaller than the time the user needs to switch machine, we would never have to deal with conflicting versions of the files because the user would always see and edit their last version. Therefore, it would not be necessary to provide a conflict detection system. However, there is the possibility of the user to switch to another machine before the synchronization process between all the machines is concluded and more important, the possibility of the user changing the working set of files while offline. In both cases, the user can accidently change old versions of files or folders. The *Bolt Cloud* system considers these scenarios and responds accordingly including a conflict detection system detailed later.

The first decision to make when thinking about synchronizing a set of files is the model of synchronization. The synchronization process can be done between the different computers directly or through a server. When designing *Bolt Cloud* we opted for including a server in the process. Having a remote server coordinating the system eases the synchronization process as well as allows the user to access the

system from every location, provided there is enough connectivity. This is very important to enable mobility, specially when the user is using a laptop. Another positive aspect of having a server is that we can build it to be reliable and scalable. Following the Cloud Computing model we can expect the server to be deployed on a *Cloud* with elastic resources and therefore relieve the clients of almost all the computing effort.

Bolt Cloud is composed by three modules as depicted in Figure 3.1. Along this section we describe the *Bolt Cloud* client, *Bolt Cloud* server and the storage.

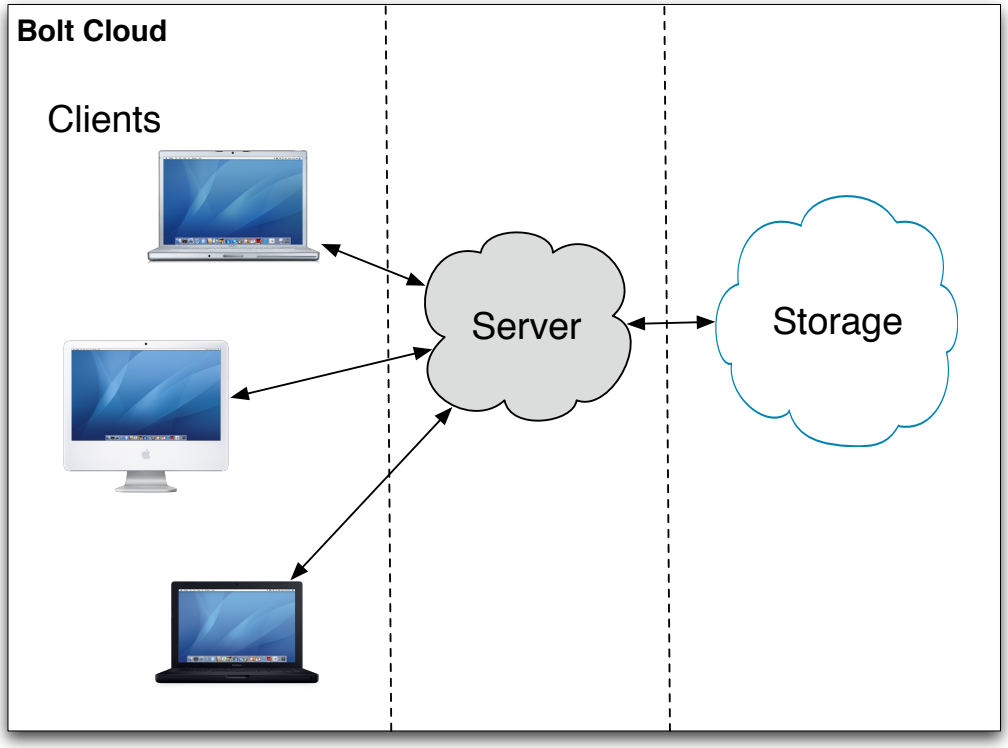

Figure 3.1: Bolt Cloud High Level Architecture.

3.1.1 Client

The Client is deployed in each of the user's machines. The Client monitors a folder chosen by the user and is responsible for detecting changes made to that folder and its contents. Periodically, the Client contacts the Server and sends the changes it has detected until then. The server processes those changes, detecting possible conflicts and publishing those changes to the other machines of the same user. Afterwards, the server responds to the client by sending a set of instructions that force the client to update its state and become synchronized with the rest of the user's machines.

The synchronization process is periodic: the system itself initiates the process after a certain period of time. Nevertheless, the system also contemplates the possibility of the process being initiated manually by the user. Having both the automatic and manual approaches to how each client synchronizes is not common in the existing services (described in Chapter 2) but it is a very useful feature notably when the user knows he will disconnect from the network in the near future. In fact, if we have a system solely based on an automatic approach we don't have control of when the synchronization is actually held. On the other hand, a system that always depends on the user to initiate the synchronization process may become tedious to use. Having the ability to provide both features allows the user to rely on a periodic synchronization, which ensures that at least a subset of the changes made to the watched folder are synchronized and at the same time have the ability to initiate the synchronization process. When the user initiates the synchronization process she has direct control over the system, which is very important when the user wants to make sure a certain file or set of files, are in fact, synchronized, as would be the case of a deliberate backup.

The client has the architecture depicted in Figure 3.2. The Client is composed of three modules. A *Communication* module, the *Manager* and the *File System Watcher*. Designing the system to be

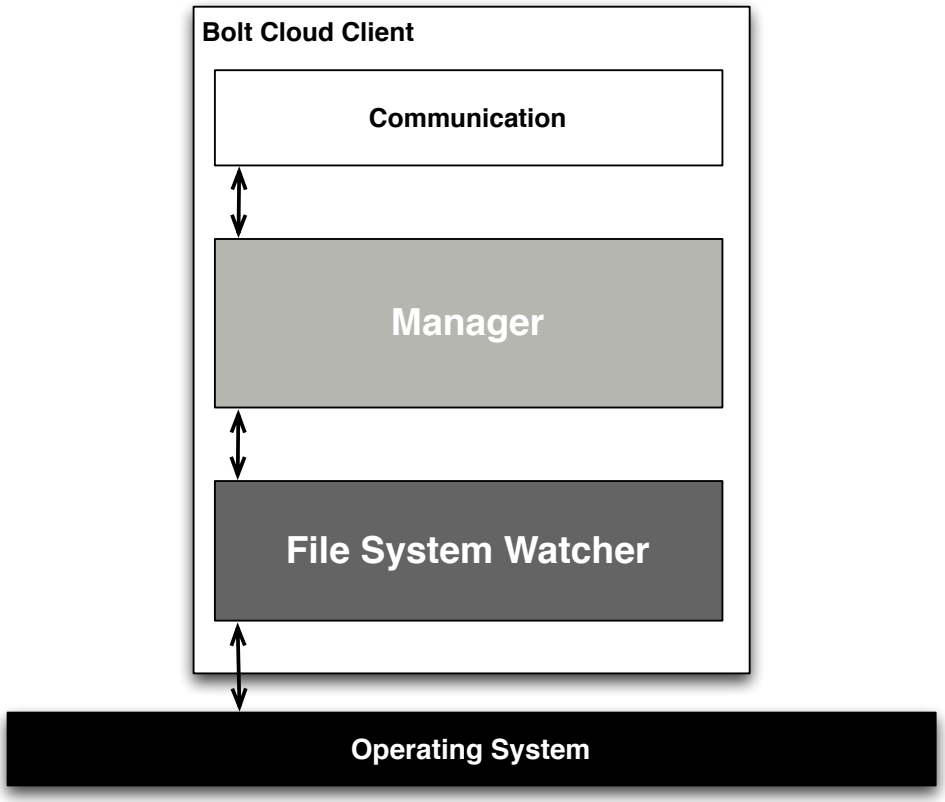

Figure 3.2: Bolt Cloud Client Architecture.

modular and have simple APIs allows code reutilization and eases changing or improving the system. If, for example, the system underneath the file system monitoring suffers any modification, the *File System Watcher* module can be rewritten and replaced easily. This is valid for each module. We now describe each one of the client's modules detailing their responsibilities.

File System Watcher Module

The *File System Watcher*, as the name implies is responsible for monitoring the file system. In particular, it is responsible for monitoring changes happening to the folder being watched by the system. This module is intrinsically connected with the operating system to be able to detect changes to the watched folder. On change detection, this module identifies the change type, which can be a file creation, modification or deletion or folder. These changes are then noted and this module also has the ability to detect changes made to the watched folder while the system is offline. When the *Bolt Cloud* client starts, the *File System Watcher* module compares the contents of the watched folder with information it stored previously, determining if changes occurred while *Bolt Cloud* was not running. These changes are also noted and the client continues to run normally. The module exports a method that enables other modules to poll for changes.

The changes detected are stored as a list of changes with information about the path and the type of change. In the particular event of *file modified* change the system stores more information than just its occurrence. The *File System Watcher* also stores information about the file blocks changed. This information is used afterwards to avoid sending the whole file to the server. The *Bolt Cloud* client will only send the blocks that have changed when a file is modified saving resources. This also enables faster synchronization times by reducing the amount of data sent to the server.

Manager Module

Synchronization may happen periodically or be manually started by the user. In both cases, the *Manager* module calls the *File System Watcher* to retrieve the changes detected and processes them. The *Manager* module is therefore responsible for triggering periodic synchronization process as well as controlling the whole process. The changes retrieved consist of a list of paths and respective change types as well as a data structure containing information about modified file blocks. The *Manager* module then sends the changes to the *Bolt Cloud* server. The server processes the information sent by the client and responds with a set of instructions the client must follow in order to become synchronized with all the other clients. These instructions are received and applied by the *Manager* module terminating this round of synchronization.

Communication Module

The Communication module is responsible for managing the communication between the *Bolt Cloud* client and the *Bolt Cloud* server. In section 3.3.1 we will go into further detail about the technology behind the communication module.

3.1.2 Server

The *Bolt Cloud* server is responsible for managing user accounts, managing the storage and the conflict detection system. Following the same approach we used previously to describe the *Bolt Cloud* client we will consider the different server modules and explain their roles. The *Bolt Cloud* server is composed by three modules as depicted in Figure 3.3. The *Controller* module is responsible for receiving the clients requests and process them. To accomplish this task it has a connections to the *Conflict Detection System* and to the storage through the *Storage Driver*, which isolates the *Bolt Server*

from the underlying storage system.

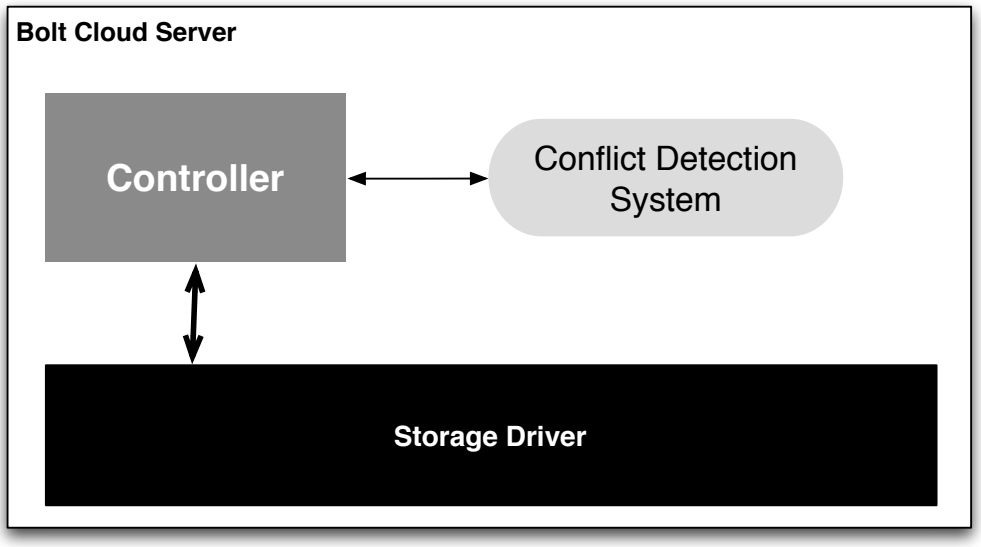

Figure 3.3: Bolt Cloud Server Architecture.

Controller Module

The *Controller* module handles all client requests. This module needs authentication information and to access user's profile. This includes user credentials, information about the user's machines and the index to access the user's files.

When a *Bolt Cloud* client connects to the server the user's identity is authenticated and the *Controller* loads all the information belonging to that user. To the set of information related to a certain user we will call the *user's state*. The user's state has information about the user credentials, the name of the machines owned by the user and a list of pending changes for each machine. During each synchronization process the list of pending changes is updated so that each machine receives the changes from all the other machines.

Even though *Bolt Cloud* is intended for personal use conflicting operations can occur. The system answers these scenarios and always

tries to preserve the user's intentions and data. This is achieved by a Conflict Detection System that will be discussed later. Besides detecting usage related conflicts, it is important to have concurrency control mechanisms to coordinate how the *Controller* module deals with the user data. Since the *Bolt Cloud* client has an automatic synchronization system, it is possible that more than one machine from the same user tries to update the user's state at the same time. If nothing is done to prevent this, the system can easily loose updates and evolve to an inconsistent state.

One motivation for this work was the large number of prospective users. As a result, it is crucial that the system is designed to be able to scale and handle thousands of requests simultaneously. To achieve this goal it is desirable that the servers be stateless. Having stateless servers eases horizontal scalability, i.e., if the demand for service grows the system only has to start more server instances to answer all the requests. However, the server has to access the user's state and to conform to a concurrency control mechanism. This could mean that all the servers would have to share state and agree in all the changes made to that state. Note that having an agreement protocol to control the access to the state and avoid conflicting changes is possible but it compromises scalability.

To avoid having shared state between the different instance of the server, we use a distributed coordination service. This service itself is distributed and fault tolerant. This service assigns each user a lock. When a server instance manages a certain user data, it has to acquire the lock and fetch the user state from the storage. This ensures that the object is changed only by one server instance at a time avoiding undesired conflicts. The coordination service used and its properties will be described in Section 3.3.4.

The Server architecture including the Distributed Coordination Service is depicted in Figure 3.4.

Having described how the user state is handled we will continue to

describe the *Controller* module behaviour. After fetching the user's state the *Controller* reads the changes sent by the user's machine requesting the synchronization. Then, it asks the *Conflict Detection System* to compare the changes read with the set of changes that are pending for that machine. These pending changes are those changes published by other machines since the last synchronization. The *Conflict Detection System* module detects possible conflicts and produces two sets of instructions. One of these instructions sets is to be applied by the server in order to update the storage state and the other to be sent back to the client machine. After these instructions are followed both the storage and the client machine become synchronized and the connection is closed. The *Controller* is now free to serve another request.

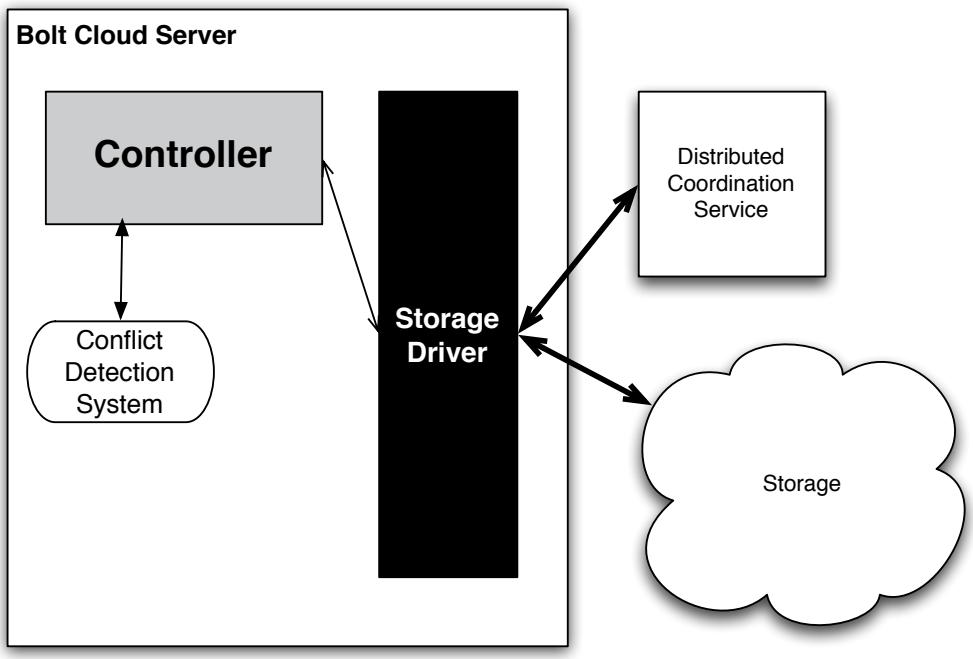

Figure 3.4: Complete Bolt Cloud Server Architecture.

Conflict Detection System

When we defined the Working Set Dissemination Service and the goals considered in this work, it was stated that the service would be intended for personal use only. This means that changes are made only in one computer at a time. Theoretically, this would mean that no conflicts could ever occur. Whenever a change occurred, it would be propagated to every computer and when another change was made, it would be made to the last possible version of the file or folder. However, we must introduce the possibility of having offline computers at the time of change or synchronization and having changes during synchronizations. These aspects open the possibility to conflict occurrence. Conflict means that a file or folder is subject to a change in at least two different computers and these changes leave the file or folder in two different and valid states.

A conflict can occur for example when the user changes a file in one computer and turns it off before synchronizing the file. If the user edits the same file in another computer the file will diverge into two valid versions. When the first computer starts and synchronizes with the server it will receive the notification of a change made to the file. As the computer also has a change over that file to publish we may end up with a conflict.

In the design of the *Conflict Detection System* we considered as top requirement the preservation of the user's intentions. The system was designed to detect and not to solve the conflict. The automatic resolution of the conflicts is impossible in situations as the one just described. When unable to track the order of changes any generic tie-breaking policy would lead to undesired losses and a generic merging policy is simply unattainable. Instead of actually solving the conflict by trying to automatically merge the two versions of the file or folder, we devised a strategy to preserve both versions and signal the user that a conflict occurred. This way the user never looses any information and can choose one of the versions or even merge them.

The conflicts that can occur are:

- *File Add or Modify – File Remove*
- *File Add or Modify – File Add or Modify*
- *File Add or Modify – Folder* containing the file *Remove*
- *Folder Add – Folder Remove*
- *Folder Add – Folder Add*
- *Folder Remove – Folder Remove*
- *File Remove – File Remove*

For each type of conflict the system will perform a set of steps to address it. The actions performed for each situation are described in Table 3.1.

	File Add or Modify	**File Remove**	**Folder Add**	**Folder Remove**
File Add or Modify	The computer that initiated the synchronization process renames its version to *filename + (conflict)* and receives the remote version.	The file remove is ignored and the file is synchronized.	-	If the folder contains the file being added or modified it is created and the file synchronized.
File Remove	Same approach as the upper column.	-	-	-
Folder Add	-	-	The folder is created in both places.	The Folder Remove instruction is ignored and the folder is added.
Folder Remove	The same approach as the File Add or Modify / Folder Remove column.	-	The Folder Remove instruction is ignored and the folder is added.	-

Table 3.1: Conflict handling strategy.

The operations described always preserve the user's intention by never deleting any changes made to the files and folders. By renaming the files in conflict the system alerts the user to the conflict and the user can act accordingly.

Storage Driver

The *Storage Driver* module is the mediator between the *Controller* and the Storage. The API used by the *Controller* to communicate with the Storage Driver is the following:

- ***sendAllObjects****(user,client connection)*
  The storage driver asks the storage for a special object containing an index of all the user's objects. Then asks the storage for all the objects in that index and sends them through the client connection. This method is invoked the first time a machine is registered in order to receive all the files already in the system.

- ***sendFile****(user,key,client connection)*
  The storage driver asks the storage for the user's object with the specified key and sends it through the client connection. This method is invoked each time a file has to be added to the client synchronizing.

- ***putObject****(user,key,object)*
  The storage driver puts the object in the storage. This object can be a file or folder.

- ***removeObject****(user,key)*
  The object with the specified key is removed from the storage.

- ***modifyObject****(user,key,diff)*
  The object with the specified key is retrieved from storage and information about differences made to the object in *diff* are applied. The the updated object is put in the storage.

- *bool* ***userHasMachine****(user,machine)*
  The storage driver asks the storage for the user state object. Then verifies if the machine specified already is registered for that user.

- ***addMachine****(user,machine)*
  The machine specified is added to the user's machine list.

- ***addUser****(login,pass,firstmachine)*
  A new user is added by creating a new user state object.

- *userState* ***getUState****(user)*
  This method returns the user state object. This method is called whenever a synchronization process is initiated on the server side.

- ***setUState****(user,userState)*
  This method allows changes to be made to the user state.

3.1.3 The Storage

The storage is a stand alone system that can be completely independent from the *Bolt Cloud* system. The system need only to know how to interact with the storage, which means it only has to know the storage API.

The storage API considered in *Bolt Cloud* is the following:

- ***putObject****(key,object)*
  If the object already exists it is replaced.

- *object* ***getObject****(key)*

- ***removeObject****(key)*

When designing *Bolt Cloud* we considered a key-value storage. This type of storage allows a client to store data objects, which can be

anything, and access them by defining a key for each of them. This model meets the *Bolt Cloud* requirements as files and user information can be mapped to objects by generating keys adequately. As the key-value massive scalable storage systems are often offered as a service we can use them directly by implementing an adequate storage driver. Implementation details about the storage system are described in section 3.3.5.

3.1.4 Synchronization Process

The synchronization process involves the server and the various machines of a single user. Each machine is running an instance of the *Bolt Cloud* client.

The client monitors a folder chosen by the client and detects changes made to that folder. We will start to describe the synchronization protocol by admitting a change occurred in one of the user's machines.

The machine where the change occurred has the description of this change in a data structure. This data structure has the relative path of the file or folder changed and the type of change (add, delete or modify). The relative path is the file or folder path from the root folder (the folder being watched). The relative path is, therefore, equal across machines. If the change is of type *file modified* the data structure will also have the data concerning blocks changed.

When a synchronization begins it starts to collect the information about the changes that occurred since the last time the process run. From now on we will refer to this information only by changes. After collecting the changes, the client contacts the server and authenticates by sending information about user login, password and machine name. If the authentication is successful, the client sends the changes to the server and waits for a response.

For each machine the server has a list of pending changes. These

pending changes are changes other machines made to the files or folder and are not yet synchronized. After receiving the client changes the server picks up the list of pending changes for the machine name the client provided. Then the server compares the pending changes with the new changes coming from the client. From this comparison the server concludes if there are conflicts to be handled[1]. In either case, the server produces two set of instructions. One set of instructions to be applied by the server to the storage system and the other to be applied by the client. After applying these instructions, the client and the server become synchronized. The set of instructions intended to be applied to the storage is also added to the set of pending changes of the other machines so that all machines are notified about the changes made.

This process is then repeated for the other machines as soon as they initiate the synchronization process (manually or automatically).

The client that was waiting for the response receives it in the form of the instruction set. Applying the instruction set can involve creating or deleting folders, deleting files or downloading new file versions from the server. After this process the client finishes the synchronization process and disconnects from the server.

Figure 3.5 depicts this process.

3.2 *FlashSync*

The *FlashSync* system is an improvement to the existing Working Set Dissemination Services. It is composed by two different subsystems: the *Personal FlashSync* and the *Group FlashSync*. These systems were designed based in some observations related to way Working Set Dissemination Services work today.

[1] The way conflicts can occur and the way they are handled have already been described in Section 3.1.2

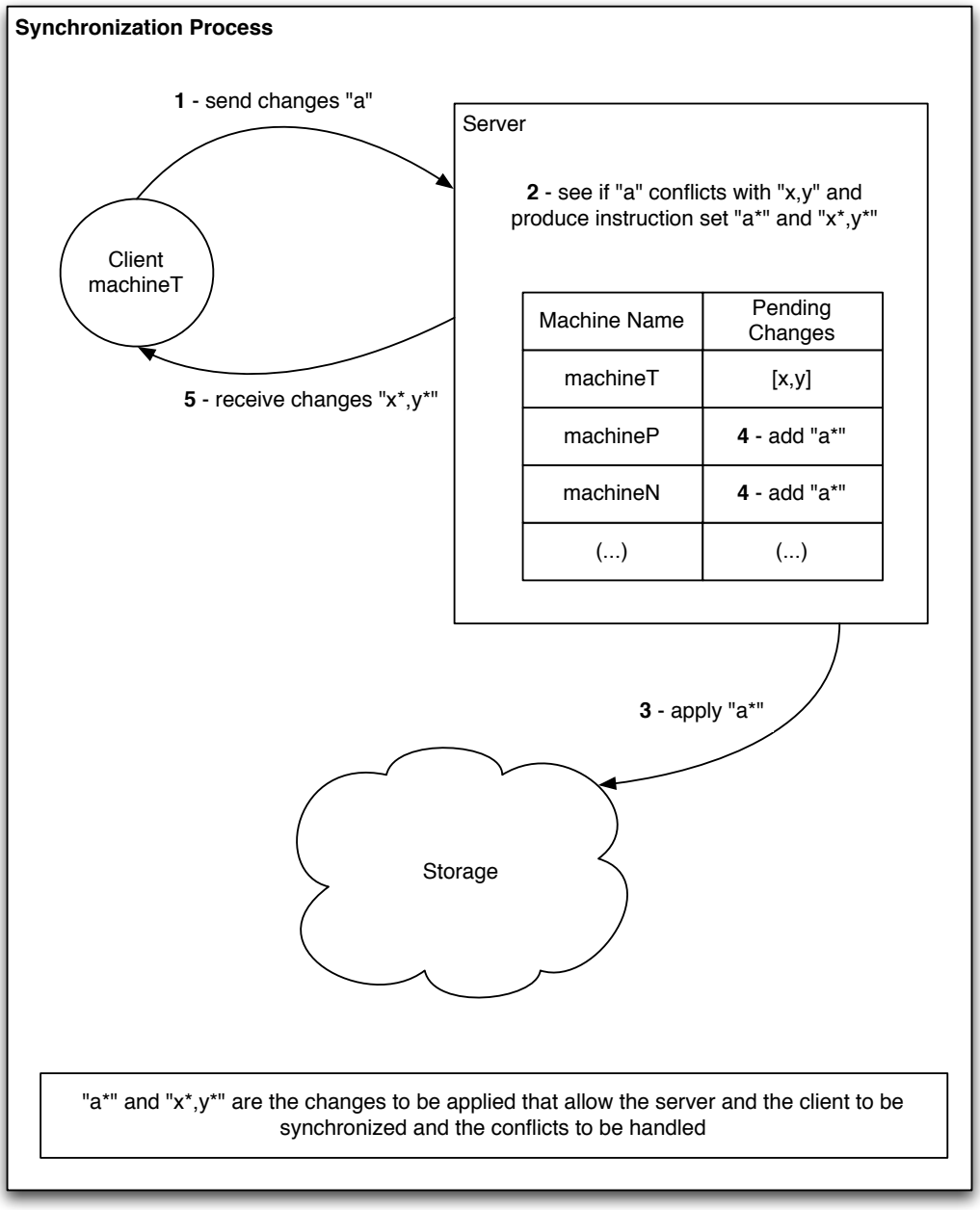

Figure 3.5: Synchronization Process.

As it is implemented in existing Working Set Dissemination Services, the synchronization process involves data travelling from one computer to the server and from the server to the other computers. Often, several of the user's computers are in the same network, typically at home. Therefore, it does not make much sense to send the data to the server and then download it back. Furthermore, we can take advantage of the fact that often a user is connected behind a firewall in a local area network alongside many potential *Bolt Cloud* users, for example, at the university. If at the time a user arrives at the university and connects her laptop the data to be synchronized is already available locally, in a colleague's computer for example, she will save a significant amount of time in the synchronization process.

The *Personal FlashSync* system is aimed at optimizing the synchronization between computers in the same network and belonging to the same user while the *Group FlashSync* system involves a collaborative environment where computers from different users have information related to each other. Along this section we describe these two systems detailing their objectives and how they work. We begin by describing how they were incorporated into *Bolt Cloud*.

3.2.1 *FlashSync* and *Bolt Cloud*

To incorporate the *FlashSync* system, three modules were added to the *Bolt Cloud*'s system architecture, two in the client and one in the server. Client and server architectures are depicted in Figure 3.6 and Figure 3.7.

The server-side *FlashSync* module stores information about the users who activated the *Group FlashSync* system. Each time a user activates the *Group FlashSync* system the user's IP is recorded and indexed to be easily connected with other users behind the same firewall or router. This module is also responsible for managing the unique identifiers assigned to each group of changes processed. In

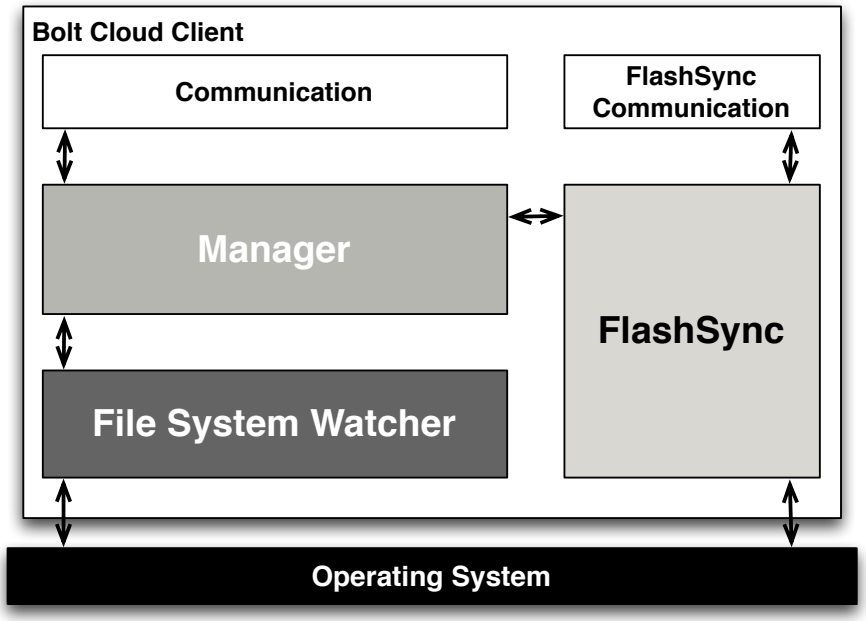

Figure 3.6: Bolt Cloud client architecture including FlashSync

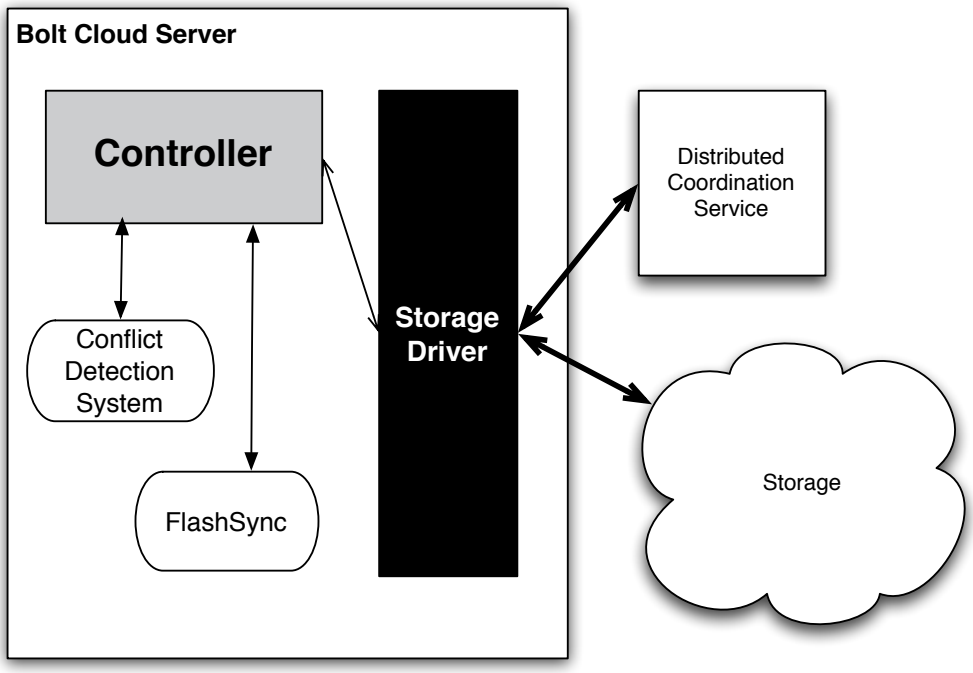

Figure 3.7: Bolt Cloud server architecture including FlashSync

fact, while some changes are pushed to other users' computers new changes can arrive to the server. To distinguish new changes from those it had already pushed through the *Group FlashSync* system, the *FlashSync* system has to stamp each group of changes with an unique identifier.

The server-side *FlashSync* module also tracks the validity of pushed changes. When a client connects on a network where there is no other *Group FlashSync* enabled computer with available information, it will synchronize with the server directly. When this happens, all the pushed changes relating to that client become obsolete and are ignored in future synchronizations.

The client-side *FlashSync* module is used by the *Personal FlashSync* system and by the *Group FlashSync* system to discover all the other *Bolt Cloud* clients running in the same network. This process is done in collaboration with the *FlashSync* communication module, which relies on different technologies as described in Section 3.3.

The client-side *FlashSync* module also controls the synchronization process between machines in the same network and deals with possible conflicts using similar strategies to the ones described in Section 3.1.2.

3.2.2 *Personal FlashSync*

The peer-to-peer concept in its origins suppresses the need to have a server mediating communication and cooperation between peers. Each peer or node communicates directly with its pairs and together collaborate to achieve a common goal. In the particular case of the Working Set Dissemination Service, we devised the *Personal FlashSync* to allow computers, within the same network and belonging to the same user, to synchronize with each other without all of them having to contact the server. When the *Manager* module initiates the synchronization process it asks the *FlashSync* module to begin

the local area network sync. All the computers in the same network and belonging to the same user are discovered and changes to be synchronized are sent to each of them. This process is done one computer at a time to ensure that all of them can contribute with changes and that in the end all share the same state. After this process the computer that initiated the synchronization contacts the server and the server also becomes synchronized.

The *Personal FlashSync* system does not avoid the need for a server mediating the process but minimizes the amount of data traveling in the network optimizing the synchronization process. Even though we considered along this work the server deployed in a *Cloud* with elastic resources, the performance of the system is always bounded by network constraints. Taking advantage of the fact that today personal computers have significant computational resources, the *Personal FlashSync* system works as a complement to the *Cloud* and results in an optimized system.

Personal FlashSync **Process**

The *Personal FlashSync* process runs in each synchronization but it is initiated before the client contacts the server. *Personal FlashSync* provides the ability to synchronize all the user's machines in the same network.

The first step consists in discovering all the machines in the same network that have the *Bolt Cloud* client running and that belong to the same user. This process is done by a separate thread so this information is always available to the rest of the system.

When initiating the synchronization process the client contacts all those machines in the same network and informs them that a synchronization is about to begin. This way all of them suspend their own synchronization processes and change their application icon signaling the user that a synchronization is in progress. Afterwards, the

client contacts all these machines one at a time sending them the changes. The contacted machine will act as server and handle the conflicts and will respond with a set of changes of their own. Both the client that initiated the process and the contacted one apply the respective changes and become synchronized.

This process has a small but very important detail that distinguishes it from the synchronization process. As a client has to synchronize with potentially more than one machine in the network it has to propagate its own changes but also the ones it receives from the other machines. Each time the client receives changes from one machine it has to send them to all the machines that already contributed to the synchronization process. This way, at the end all the machines become synchronized.

After these steps, the client initiates the normal synchronization process with the server but also sends the server the names of the machines that already reflect those changes. This way, the changes sent will not go to their pending changes list. When these machines try to synchronized with the server they will not have to download those changes and the system as a whole benefits from it.

Finally, the client contacts all of the machines informing the synchronization process ended and continues to run normally.

Figure 3.8 depicts this process.

3.2.3 *Group FlashSync*

While *Personal FlashSync* works between computers of the same user and is intended typically for home networks, *Group FlashSync* is intended for computers working in a larger environment such as the university or a company. In these scenarios, we normally connect our computers to a network behind a firewall and share this network with colleagues' or coworkers' desktop machines and laptops. We can use these devices as a cache for synchronization information.

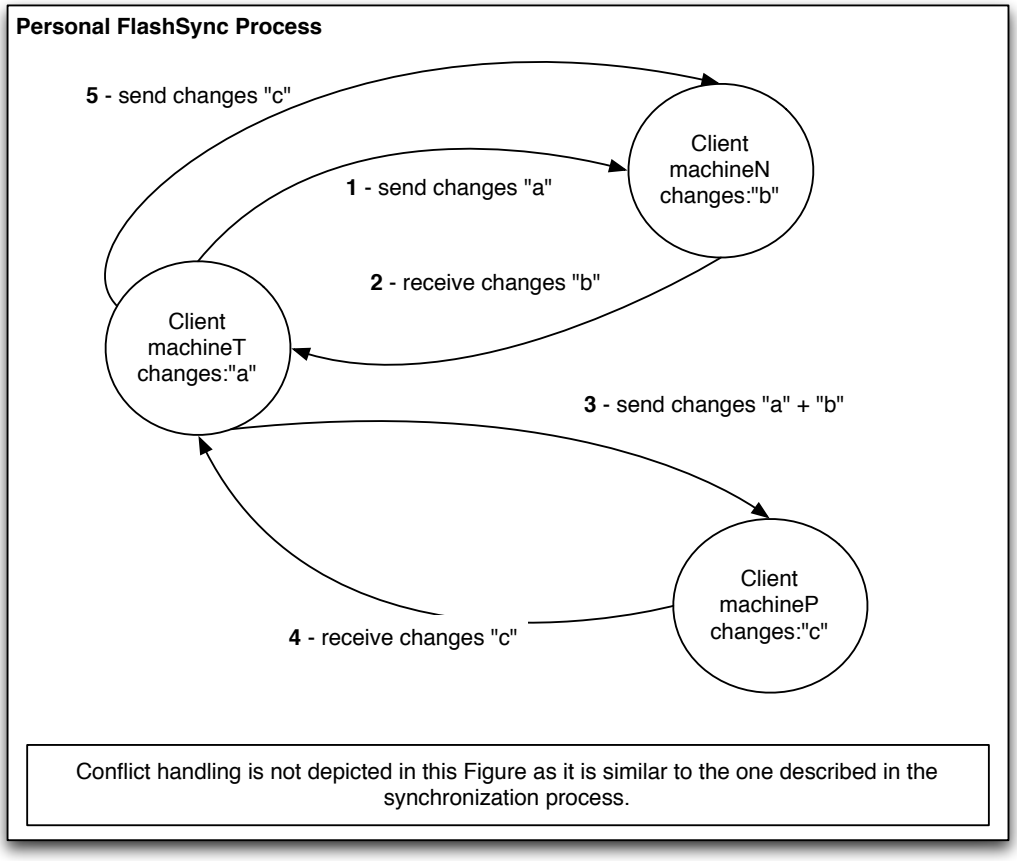

Figure 3.8: *Personal FlashSync* Process.

Each user that activates the *Group FlashSync* system is asked to share a certain amount of disk space with other users. Then the system detects the IP from where the *Group FlashSync* was activated and this IP will be the IP from the Firewall, which is the same for every colleague's or coworker's computer. Every time a client synchronizes from the same IP the *Bolt Cloud* server will send along with the normal synchronization, information from other *Bolt Cloud* users that have activated the *Group FlashSync* for that IP. The client synchronizing performs the normal operations to its folder and stores the information regarding to other users encrypted in a separate folder.

When a user connects in a *Group FlashSync* enabled network, she can synchronize pending changes (changes made at home for example) without having to download them from the server because they were already pushed to a colleague or coworker machine. Furthermore, there are always some machines at the university or company permanently connected and these are ideal to be the cache for all the *Bolt Cloud* users in that network.

Group FlashSync Process

When a client connects to a *Group FlashSync* enabled network, discovers all the *Bolt Cloud* clients running in that network and asks if any of them has changes for it. In case there are, the computer that has those changes replies to the client with the identifier of the changes it owns. The client sends these identifiers to the server and asks for changes. The server checks if those identifiers are valid and up to date. If they are, the server replies to the client with only new changes and instructions to synchronize the already pushed changes. Otherwise, the server instructs the client to ignore locally available changes and replies with all the available changes.

This process is depicted in Figure 3.9.

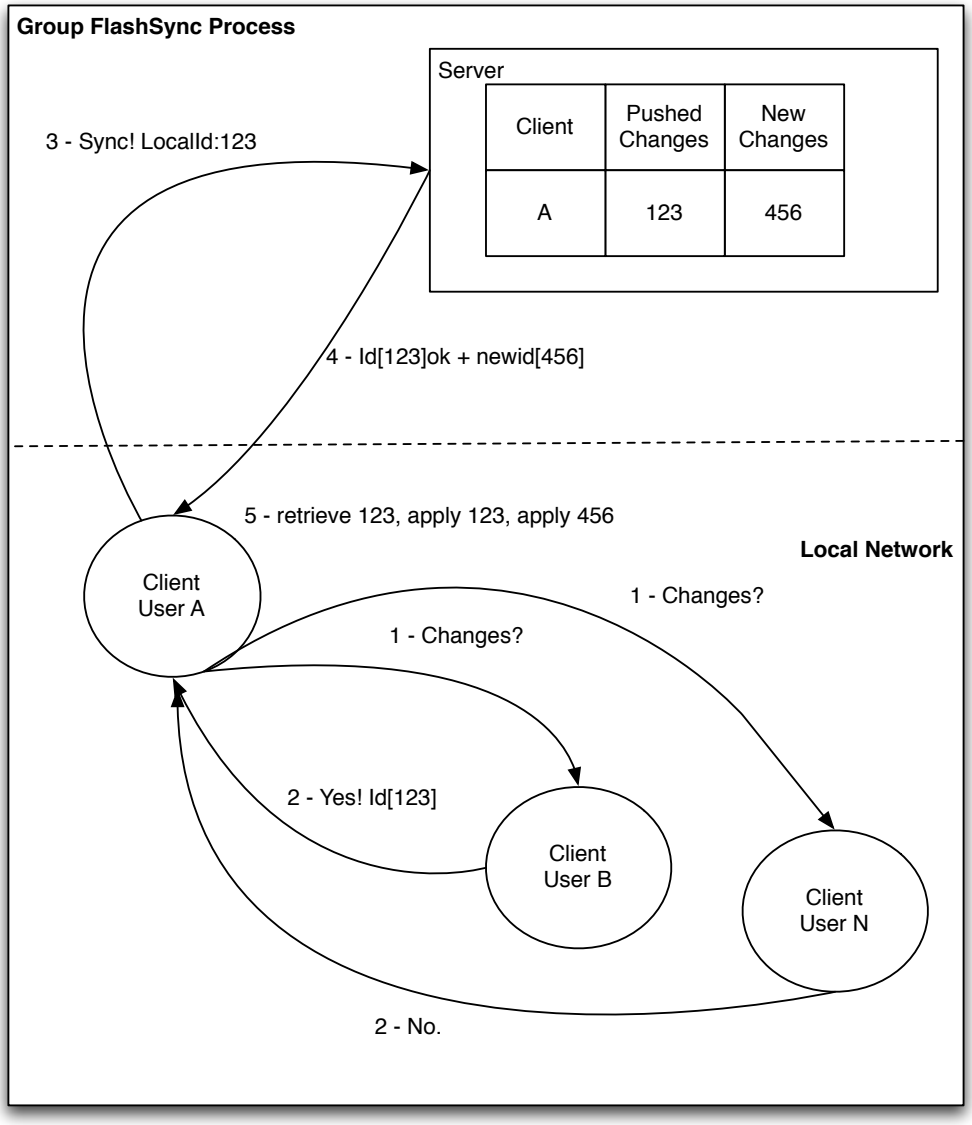

Figure 3.9: *Group FlashSync* Process.

3.3 Prototype Implementation Details

In this section we describe some of the prototype details. Some of these details will be useful to better understand the architecture proposed and the *Bolt Cloud* system.

3.3.1 Programming Language Considerations

In order to implement the *Bolt Cloud* system, a decision was taken to start with a client written in Objective-C and the server written in Java. Since the author is a Macintosh user, Objective-C was the logical choice for the Mac OS X operating system as it is directly supported in the system. Specifically, when building the File System Watcher this choice enabled the use of the FS Events framework directly. Details about the File System Watcher will be in Section 3.3.3.

The *Bolt Cloud* server was written in Java. Java is a very mature language and has a very interesting framework supporting it. Java's portability was also a key asset as it allow the server to run in almost every machine or system. This is important because we want to be able to run our server in different machines but also in different virtualized environments, which are very popular when applying the Cloud Computing model.

Even though both choices seemed logic, together they represented a challenge. Implementing the communication between the server and the client was one of the most time consuming tasks. The absence of standard frameworks or modules to ease the process forced the implementation of all the communication based purely on BSD sockets, which is known to be rather taxing. The communication modules were implemented but possibly at the cost of not being able to enhance other aspects of the system.

As described earlier, there were two different communication mod-

ules. One module wase described in the previous paragraph and is the mediator between the *Bolt Cloud* client and the server. The other one is used by the *FlashSync* module to communicate with all the computers in the same network. This module is implemented using the *Bonjour* services. *Bonjour* [4] is Apple's implementation of zero-configuration networking [30]. The *Bonjour* system allows automatic discovery of devices and services on IP networks. On top of bonjour, the *FlashSync* module uses the Distributed Objects technology from the Objective-C language to allow communication between machines. Using these technologies, what really happens in our system is that each *Bolt Cloud* client has a *Bonjour* service associated. This service allows other clients in the same network to discover it and interact with it. This service serves as mediator between the different clients and enables the local area network synchronization process.

3.3.2 User Interface

The user interface is non-intrusive and the amount of user interaction needed is minimum. This complies with one of our initial goals. Therefore, the user interface consists of only two different components. The first one is a dialog window that appears when one first runs *Bolt Cloud*. This window is used to obtain the user information necessary to initiate the system. Besides the login and password, the user also has to choose the name of the machine where the client is running (this is important to distinguish the different user's machines), and the folder name to be watched. Finally, the user chooses where he wants the watched folder to be created. This window is depicted in Figure 3.10.

The other component appears when the application is running. This component is a Status Item located on the menu bar on top of the screen of the Mac OS X system. This icon has a menu associated that allows the user to close the application and to manually initiate

the synchronizing process. The icon is depicted in Figure 3.11. This icon changes when a synchronization is in process alerting the user.

3.3.3 File System Watcher

The File System Watcher is a module intrinsically connected to the operating system. This being the case, we will describe in more detail its implementation. The File System Watcher has to monitor a folder and report or store all the changes made to that folder specifying the type of change and the object of the change. This means that for each change the File System Watcher must record which file or folder was changed and what was the change (add, remove or modify). To achieve this goal we used the FS Events [8] API offered by the Mac OS X system. Using the FS Events API our application is notified every time a file system change occurs inside a folder we specify as root folder. Events on that specific folder and child folders are all detected as the system is recursive. One particular detail about this

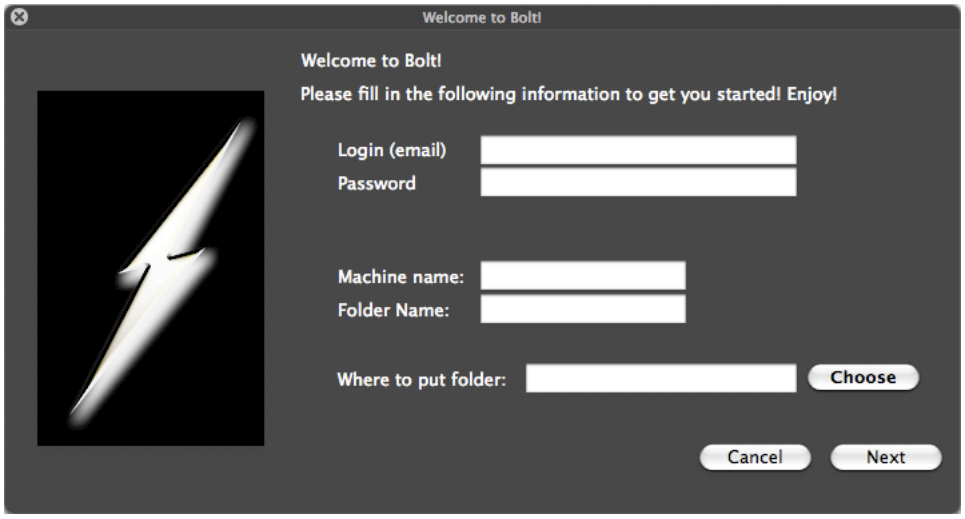

Figure 3.10: Bolt Cloud Login Window.

Figure 3.11: Bolt Cloud Menu Bar Icon.

system is that it is folder oriented, in the sense that attached to each change notification is the path of the folder were the change occurred and not of the object that actual changed. The application is therefore responsible for determining what actually changed in the given folder and act accordingly.

For our File System Watcher we had to implement a data structure and some search methods to search for what actually changed on each notification. The data structure has information about folder contents and file modification time which is later used to determine if a file or folder was added, deleted or modified. This information is then stored to be available to other *Bolt Cloud* modules.

Another important problem we had to overcome was the possibility of offline events. Offline events are those file system changes that are made while the *Bolt Cloud* system is not running. These events must be detected and added to the list of stored changes. The FS Events API offers the possibility of storing the offline events and then send them to the application when it starts. However, the FS Events system does not distinguish the offline events from those that are new. As we run a synchronization process on application start and we wanted to signal all the possible conflicts arising from offline events, we wanted the File System Watcher to be able to clearly distinguish them. To accomplish this we store a snapshot of the file system tree state that allows us to compare to a new possible state when the application starts. This way we detect all the offline events.

Finally, the last feature included in this module is the ability to detect file modification at block level. When the File System Watcher detects a new file it computes an hash value for each file block. This information is stored in an auxiliary file. Later on, if a file is modified the File System Watcher will compute a set of new hash values for the file and compare with the ones it had previously stored. From this comparison it will detect the set of blocks modified, added or

those that were removed. During the synchronization process, instead of sending the whole modified file to the server the client will only send the necessary blocks saving bandwidth.

3.3.4 Distributed Coordination Service

For our implementation we used a system similar to the Chubby lock service [25]. Chubby was successfully used in Bigtable [26], therefore offers the desirable availability and reliability.

The open source implementation of the distributed and fault tolerant coordination service used is ZooKeeper [23]. ZooKeeper allows distributed applications to coordinate processes by exporting a simple API similar to the one of a file system. ZooKeeper provides synchronization, groups and naming services. For the *Bolt Cloud* system we used ZooKeeper as a distributed lock service.

3.3.5 Storage Implementation

The *Bolt Cloud* prototype at this point is using the hard drive directly as storage. However, the storage is intended to be an independent key-value massive scalable storage system. As we described in Chapter 2 there are many storage systems implemented and successfully running. If *Bolt Cloud* was to be deployed in a real world scenario, it would incorporate one of these storage systems.

Our proposal is to use Amazon's Web Services. *Bolt Cloud* would use Amazon's SimpleDB to store user related data and Amazon's Simple Storage Service to store files. These systems provide high availability and reliability and at the same time provide a pay-per-use billing model, which is suitable to *Bolt Cloud*.

Chapter 4

Evaluation

A Working Set Dissemination Service is difficult to evaluate. The large number of prospective users makes it difficult to simulate real life load conditions. In this chapter we describe some experiments we carried and that allow some insight about the performance and resource consumption of *Bolt Cloud*.

4.1 Test scenario

We tested the system under a single server deployment and then with two servers running. The storage in both situations was the local hard drive of one of the servers and Zookeeper was also running in that same server. The servers were two Macbook Pros with an Intel Core 2 Duo 2,2 GHz/2,5 GHz processor and 4GB of RAM each. As clients we used nine iMac computers with 2GB of RAM and an Intel Core 2 Duo 2,0 GHz processor each.

We measured the CPU usage and the memory usage in each server with different number of connections per minute. Various instances of the *Bolt Cloud* client were running watching a folder each. Each client was configured to synchronize every 10 seconds, which means 6 connections/synchronizations per minute to the server. To provide data to synchronize to the client, we wrote a Python script to create

files and folders in each watched folder every 5 seconds.

The synchronization timings and the interval between the Python's creation of files or folders were defined after some preliminary tests. We started by configuring the Python script to create files or folders continuously. This was a really bad choice as the processing capabilities of the client computer were wasted in that task only. The *Bolt Cloud* client almost never could actually start the synchronization process and this lead to really low load rates on the server. Configuring the *Bolt Cloud* client to synchronize continuously also revealed to be a poor choice. Each *Bolt Cloud* client only runs one synchronization at a time, thus when one synchronization was running others would start to create a waiting queue. In fact, in each synchronization files are sent to the server and file transfer rates are bounded to network characteristics, if the time between synchronizations is lower than the average time a synchronization lasts the waiting queue will have infinite growth. This situation also lead to low load rates on the server. With these observations in mind we configured the synchronization timing and the Python interval in order that each synchronization has data to send and the client is always active.

In the next Section we present the results we obtained and some conclusions we can draw from those results.

4.2 Results

In Figure 4.1 we present the CPU usage for the single server scenario. We can see that, has expected, the CPU Usage grows as the number of connections grows. We can also see that after eighty connections per minute the CPU usage starts to stabilize. This is explained by the fact that at this point the network reached its full capacity and the server simply could not attend more simultaneous synchronizations as it could not receive more data.

In Figure 4.2 its depicted the memory usage for the same single server scenario. Again, as expected, the memory grows with the number of connections but never exceeds the thirty megabytes level. The increased growth we can see from the seventy connections per minute forward comes from the fact that the server has to wait for the network to be free to attend all the requests. With the increasing usage of the network bandwidth the server begins to have threads waiting for data, which leads to more memory consumption. These tests are useful to see that for this test scenario the limitations are network transfer rates.

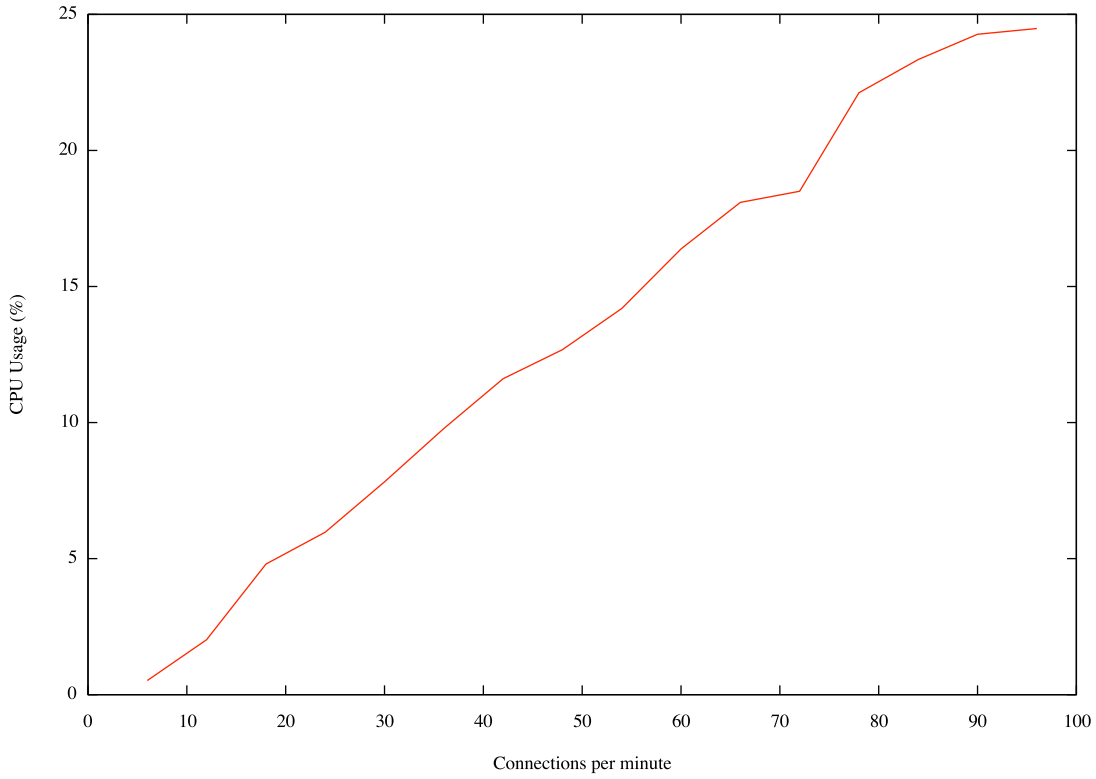

Figure 4.1: CPU usage, single server scenario.

The second test was made with two servers running. Figure 4.3 and Figure 4.4 depict the CPU usage and memory usage in each of the two servers. From these tests we can see that for the same load scenarios the load rate in each server is considerably lower than for

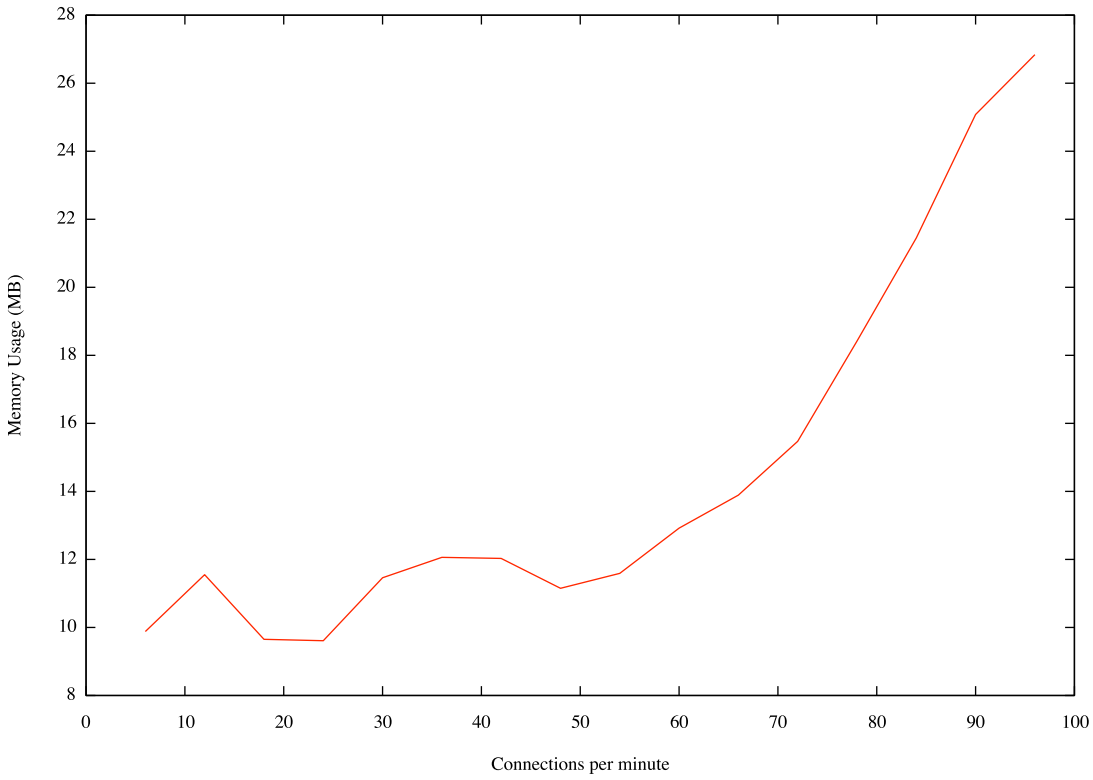

Figure 4.2: Memory usage, single server scenario.

the single server scenario. It is important to notice that Server two has always more CPU usage and memory usage and this is due to the fact that the storage and the Zookeeper service was running in Server one. The remote connection has a clear impact in Server two's performance.

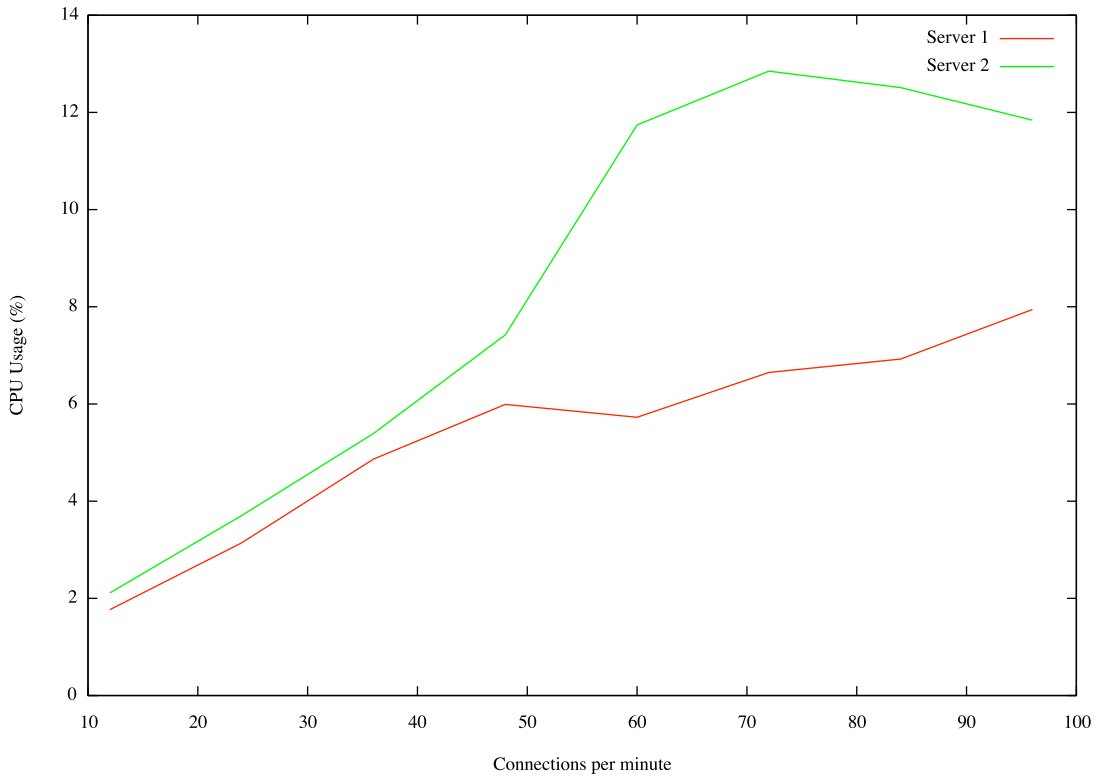

Figure 4.3: CPU usage, two servers scenario.

The important conclusion to withdraw from these tests are the scalability properties of the system. If we double the values obtained from Server two's behavior, which is the one with higher usage values, we will obtain very similar values to the ones obtained in the single server scenario. We can infer from this observation that doubling the number of servers is likely to make the system be able to handle almost twice the load. Even though real life scenario test should be made we can have here a positive evidence about the system's scalability.

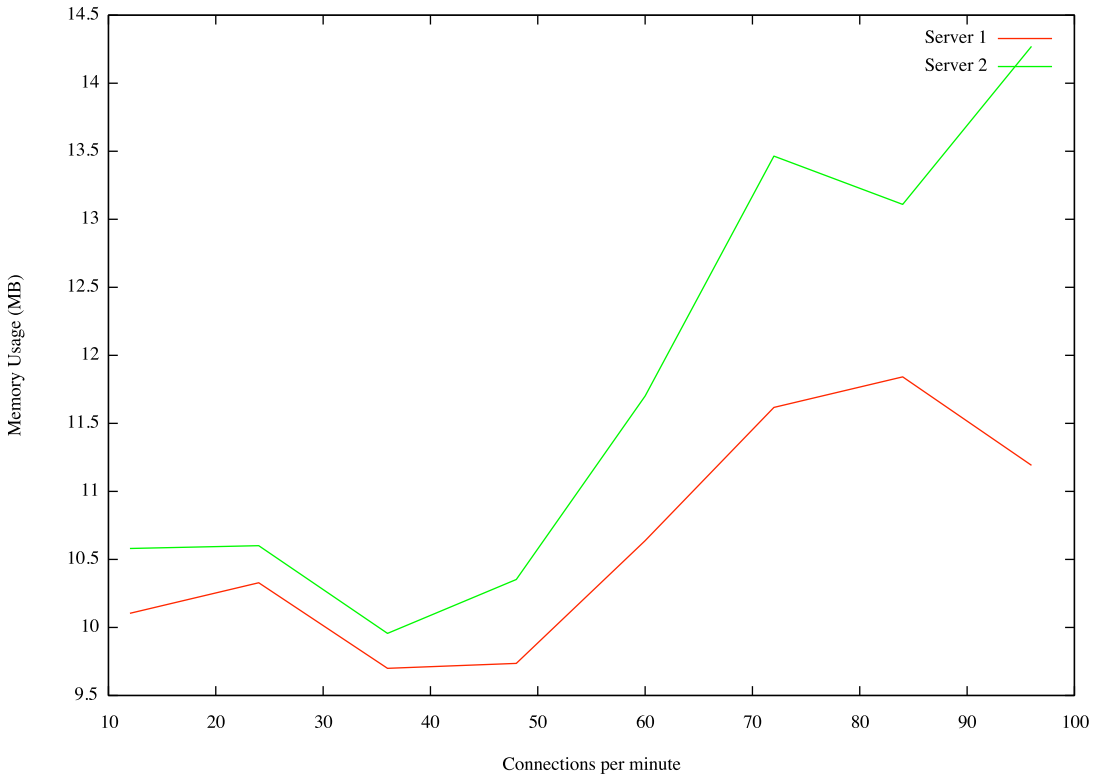

Figure 4.4: Memory usage, two servers scenario.

Chapter 5

Conclusion

The work described in this dissertation is concerned with the Working Set Dissemination Problem and had as its main goal to improve upon existing solutions. The proposed software architecture is embodied in the Bolt Cloud system from which a prototype comprising a client for Mac OS X and a Java server were implemented and evaluated. *Bolt Cloud* architecture solves the Working Set Dissemination problem and includes *Personal FlashSync* and *Group FlashSync*, which are two innovative systems that represent an improvement to how the Working Set Dissemination problem was being solved.

Bolt Cloud system allows a user to have a folder contents synchronized between various computers. *Bolt Cloud* also stores a copy of the files in a reliable storage that can be used as data back up. The user installs the *Bolt Cloud* client in each computer and chooses a folder to be assigned to the system, which then ensures that all folder contents are synchronized.

Personal FlashSync was designed based on the observation that in existing Working Set Dissemination Services, the synchronization process between computers always had the server as an intermediate. Often, computers being synchronized are connected in the same local network, for example, at the users' home. *Personal FlashSync* takes advantage of this locality factor and enables synchronization directly

between computers from the same user sharing a local network.

Group FlashSync emerged from the observation that many potential *Bolt Cloud* users can have a computer sharing the university's or company's network. We can also notice that these users also work at home and need to synchronize their laptops when they are connected to the university or company network. In this scenario, when the laptop starts the *Bolt Cloud* client, changes made at home to the working set of files must be downloaded from the *Bolt Cloud* server and this operation can be time consuming. The *Group FlashSync* system allows that users from the same university or company collaborate with each other to diminish the time this operation takes. Each user is asked to share a certain amount of disk space to be used as a cache for other users. When a user synchronizes the computer with the server, not only downloads her own information but also other users' information. When another user synchronizes within that network will already have the synchronization data available locally. *Group FlashSync* is specially suited for cases where at the university or company, there are computers permanently connected that can be used as cache for multiple *Bolt Cloud* users.

5.1 Future Work

Bolt Cloud system implementation is now operational as a prototype and a proof of concept to the proposed architecture. Having this in mind some of the system modules could be improved in order to provide better reliability guarantees. The first step would be to improve the communication modules between the client and the server. As stated earlier, these modules were built over the BSD socket API and every *marshalling* and *unmarshalling* methods were written specifically to this system. If these modules were rewritten using higher level libraries we could reach a more resilient module and open the possibility for an easier way to add more features to

the system.

Besides the libraries also some protocol related details could be improved. In case of connection failure during a synchronization the system should be able to resume the synchronization process anytime and this does not happen in the present implementation.

Finally, the system could be put into production and released to be used by the public. In this situation, we could test the system performance against real world conditions and the system's ability to scale as well as retrieve usage pattern information. This information could lead us to improve the system and study new features the service could offer. As these features would come from usage patterns they would likely be of interest to the system users.

Bibliography

[1] Amazon simple storage service, developer guide. http://docs.amazonwebservices.com/AmazonS3/2006-03-01/.

[2] Amazon simpledb, detailed description. http://aws.amazon.com/simpledb/#details.

[3] Amazon web services. http://aws.amazon.com/.

[4] Bonjour. http://developer.apple.com/networking/bonjour/index.html.

[5] Box.net. urlhttp://www.box.net/.

[6] Carbonite. http://www.carbonite.com/.

[7] Dropbox. http://www.getdropbox.com/.

[8] Fs events programming guide. http://developer.apple.com/mac/library/documentation/Darwin/Conceptual/FSEvents_ProgGuide/Introduction/Introduction.html.

[9] Google app engine. http://code.google.com/appengine/.

[10] Megashares. http://www.megashares.com/.

[11] Megaupload. http://www.megaupload.com/.

[12] Mobileme. http://www.mobileme.com.

[13] Mozy. http://mozy.com/.

[14] Opendrive online disk. http://www.opendrive.com/.

[15] Rapidshare - easy filehosting. http://www.rapidshare.com/.

[16] Spideroak. https://spideroak.com/engineering_matters.

[17] Sugarsync. https://www.sugarsync.com/tour/.

[18] Sugarsync-like applications comparison. https://www.sugarsync.com/sync_comparison.html.

[19] Syncplicity. http://www.syncplicity.com/Features/.

[20] Ubuntuone. https://ubuntuone.com/.

[21] Windows azure platform. http://www.microsoft.com/windowsazure/windowsazure/.

[22] Windows live mesh. https://www.mesh.com/Welcome/overview/overview.aspx.

[23] Zookeeper. http://hadoop.apache.org/zookeeper/.

[24] M. Armbrust, A. Fox, R. Griffith, A. D. Joseph, R. H. Katz, A. Konwinski, G. Lee, D. A. Patterson, A. Rabkin, I. Stoica, and M. Zaharia. Above the clouds: A berkeley view of cloud computing. Technical Report UCB/EECS-2009-28, EECS Department, University of California, Berkeley, Feb 2009.

[25] M. Burrows. The chubby lock service for loosely-coupled distributed systems. In *OSDI '06: Proceedings of the 7th symposium on Operating systems design and implementation*, pages 335–350, Berkeley, CA, USA, 2006. USENIX Association.

[26] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. E. Gruber. Bigtable: a distributed storage system for structured data. In *OSDI '06: Proceedings of the 7th USENIX Symposium on Operating Systems Design and Implementation*, pages 15–15, Berkeley, CA, USA, 2006. USENIX Association.

[27] B. F. Cooper, R. Ramakrishnan, U. Srivastava, A. Silberstein, P. Bohannon, H.-A. Jacobsen, N. Puz, D. Weaver, and R. Yerneni. Pnuts: Yahoo!'s hosted data serving platform. *Proc. VLDB Endow.*, 1(2):1277–1288, 2008.

[28] L. P. Cox, C. D. Murray, and B. D. Noble. Pastiche: making backup cheap and easy. *SIGOPS Oper. Syst. Rev.*, 36(SI):285–298, 2002.

[29] B. T. Loo, B. T. Loo, A. LaMarca, G. Borriello, and B. T. Loo. Peer-to-peer backup for personal area networks. 2003.

[30] D. Steinberg and S. Cheshire. *Zero Configuration Networking: The Definitive Guide*. O'Reilly Media, Inc., 2005.