University “Politehnica” of Bucharest

Faculty of Automatic Control and Computer Science

DIPLOMA PROJECT

A User Reputation System Using BlobSeer For The Storage Infrastructure

UPB Scientific Coordinators
As.-Prof. Drd. Eng. Alexandru COSTAN
Prof. Dr. Eng. Valentin CRISTEA

Author:
Mihai MIRCEA

2010

Abstract

This thesis addresses the problem of designing, implementing and testing a user reputation system for a large file sharing service relying on a distributed storing infrastructure. We aim to use BlobSeer, a distributed storage system for large data, to offer a cloud service which rewards or fines its user depending on their behaviour. A user's behaviour is influenced primarily by the upload download ratio and the quality of the files uploaded. Our goal is to offer the clients a web based interface to a robust, distribuited storage infrastructure, in which fairplay is an important prerequisite. The system has undergone a series of experimental tests and has revealed a scalable, correct behaviour.

Table of Contents

Introduction

In today's world, information has become the quintessential measure of power, profitability and success. The alleged recipe for success „Time is money“ is rapidly losing ground to the more up-to-date „Information is money“. Therefore, the information flow and all of its dependencies are of the essence. But information is too abstract a notion to be handled by computers, the primary facilitators of information flow. So a more practical, easy to represent notion was needed, and that's how the concept of data arose. By data we mean the representation of the amount of information that needs to be preserved for various later uses.

Nowadays, handling data at a large scale is becoming a very important trait. Impressive chunks of data are generated on a regular basis by many fields such as genetics, climate modeling, various types of statistics, etc. That is why it is vital to provide intelligent ways to manage large data while allowing fine-grain acces to small parts of the data. Databases, multimedia, and data mining are some examples of types of applications that require such efficient scaling to massive data sizes.

The problem addressed here has to do with the need to handle and manage large data batches(files) with great ease. Users (even home-users) are becoming more and more interested in handling their data without any technical know-how prerequisite. Large file sharing and/or storing has become an issue because it's relatively hard to design, let alone implement and test a system that meets the necessary requirements.

Motivation

The cloud is a set of hardware, networks, storage, services, and interfaces that enable the delivery of computing as a service. Cloud services include the delivery of software, infrastructure, and storage over the Internet (either as separate components or a complete platform) based on user demand. According to [6] The three cloud service delivery models are Infrastructure as a Service, Platform as a Service, and Software as a Service, and the

purpose of each model is described next.

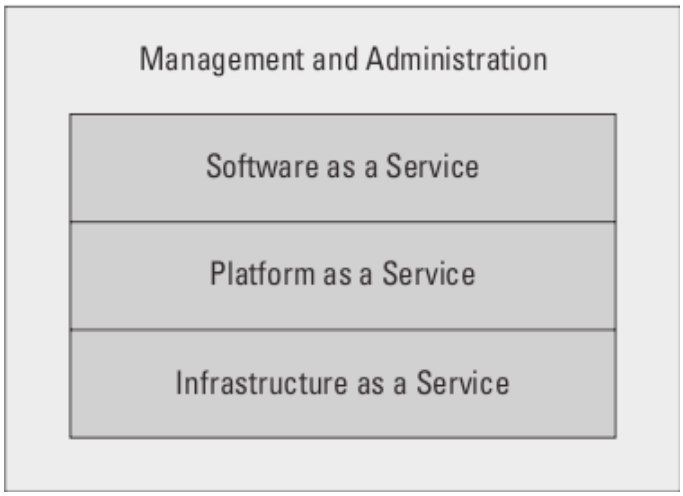
The **Infrastructure as a Service** layer offers storage and compute resources that developers and IT organizations use to deliver custom business solutions.

The **Platform as a Service** layer offers development environments that IT organizations can use to create cloud-ready business applications.

The **Software as a Service** layer offers purpose-built business applications.

We intend to expose BlobSeer as a clod service through the PaaS paradigm, thus offering users an handy method to interract with a distributed storage environment specially desingned to handle large data.

While there are a lot of technical considerations, there also is one fundamental truth: Cloud computing is a business and economic model. It's very hard to say whether cloud computing a replacement for the traditional data centre, because it depends on a lot of things than must carefully be taken into account.



However, for business agility and economic reasons, the cloud is becoming an increasingly important option for companies. Some authors see cloud computing as the foundation for the industrialization of computing.

To close in on the subject, it can be said that cloud computing is the next stage in evolution of the Internet. *The cloud* in cloud computing provides the means through which everything — from computing power to computing infrastructure, applications, business

processes to personal collaboration — can be delivered to you as a service wherever and whenever you need it.

In this paper we thoroughly describe a method to offer a cloud file sharing service to users while addresing the issues of user accounting and user reputation.

Objectives

We intend to provide the means for a user with no previous knowledge to be able to fully use the capabilities of a distributed storing environment optimized for high throughput and concurrency. BlobSeer will be delivered to the client web interface as Infrastructure as a Service (IaaS), and the end user will interact soley with the web interface, therefore providing a comfortable experience.

Also, we want to make sure that the user fully understands that this is meant to be a friendly, collaboration inclined system in which fairness to other users plays a crucial role. For that matter, the concept of reputation is addressed, the environement being permanently monitored by a system that evaluates users based on their specific actions. For example, an user who makes efforts to upload as much as he downloads will be given a high reputation and rewarded accordingly. On the other side, users who do not respect this condition will find their reputation degrading until it reaches a point where certain penalties will be applied.

Structure

The issues discussed next in this thesis will thoroughly describe the details of our approach to deliver a file sharing service using BlobSeer as a storing infrastructure. The design and implementation details of the reputation system will also be revealed along with testing results.

Firstly, there will be a chapter concerning previous work in the field of trust and reputation, followed by an overview of BlobSeer. The core of our thesis will be the description of the reputation mechanism and user accounting schema starting with the top

level components and ending with implementation details.

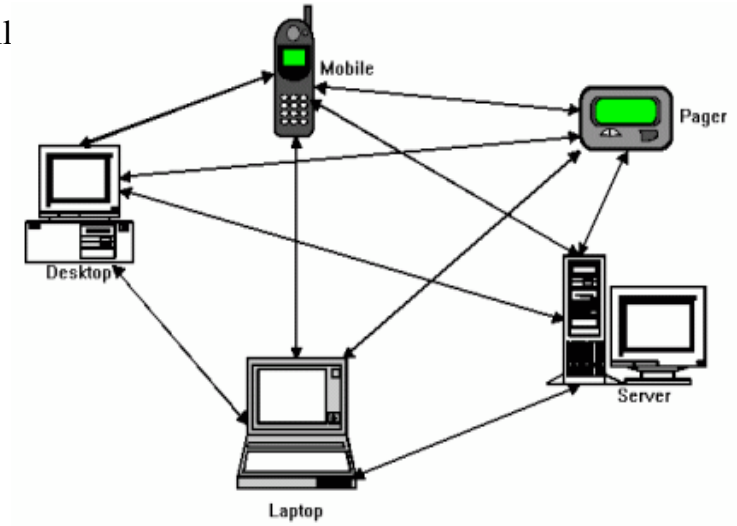
In the end we present the experimental results of our testing and an appendix containing relevant code snippets.

Related Work

Trust and Reputation in P2P networks

P2P Concepts

This section introduces P2P (Peer-to-Peer) networks as a concept and describes the features associated with them as well as a few proposed reputation algorithms. P2P networks are a special kind of networks where participants (usually called peers, or sometime nodes) are connected to each other. Peers are both suppliers and consumers of resources, in contrast to the traditional client-server model where only servers supply, and clients consume.



The nodes might have different configurations, storage capacities, bandwidths or processing speeds. Pure P2P systems change dynamically and posses features such as: no no central database, no central coordination, no peer has a global view of the system, peers/connections are unreliable and peers are autonomus and often annonimus.

The most popular flavour of Peer-to-Peer networking is file sharing, but there are many other useful areas for P2P networking such as: instant messaging and distributed calculation. However, current P2P approaches face unresolved problems that are being researched today such as security and scalability problems.

By their nature, P2P networks have a complex structure which makes them very hard to supervise and control. This is why security is one of the most complicated issues to deal with.

Two of the most important content distribution problems described are:

1) Self replication – a malicious peer answers positively to all queries and returns unwanted contents in a file, leading to fake files being spread in the network.

2) Man in the middle - a malicious computer modifies messages between two peers believing that they communicate directly between themselves. Encryption or secure channel can prevent this.

Reputation Algorithms

In the algorithm BSA [1] the trust is build on the experiences of peers and interactions between other peers. The trust values are saved as a binary string, with values (0,1). The string is built up when the peer is active in a network, first the string is zero but when a transaction is performed it becomes one bit long, at the next transaction it will become two bits long and so on. If the maximum length of the string is reached, it will use the FIFO queue principle to refresh the string values.

In [2], two mechanisms are proposed for computing the reputation score of peers. Essentially, both methods keep count on the data used and contributed by certain peers in the P2P networkby means of a non-negative number of points representing a peer's reputation score. The first one increases the peer's reputation score for uploading and decreases it for uploading; the second method credits peer reputation score for serving content, but doesn't decrease it.

[3] propose the XRep protocol, consisting of the following phases:

1) *Resource searching* – an initiator broadcasts to all of its neighbours a Query message containing the search key words

2) *Resource selection and vote polling* – upon receiving the QueryHits, the initiator selects the resource that best seems to satisfy its request. Then, it querys its peers about the download it is about to execute.

3) *Vote evaluation* – the initiator decides wheather to download the resource; if not enough voters respond positively, the selection process is repeated

4) *Best servent check* – the initiator selects the offererfrom which to execute the download. It bases its decision on the reputation and reliability of the servent.
5) *Resource downloading* – the initiator contacts the chosen servent directly and requests the resource.

Online Auctions

The E-Commerce market is a rapidly expanding phenomena, growing at a rate of about 25%. In this context, online auctions have a very important role, as they generate 15% of all online sales. eBay, the most popular platform for online auctions has an impressive user base (56 million active), annual transactions value (over $23 billion) and daily transactions (about 1 million).

The unpleasant fact reality is that online auctions account for over 40% of Internet fraude and by producing an annual loss of about $14 million it's regarded as the most popular type of Internet fraude.

The trust problem in this dynamic environment is finding a way by which an agent determines what to expect regarding the behaviour of other agents under uncertainty.

Reputation systems are widely used in e-commerce and are practically trust management systems that use reputation. *Reputation* is seen as a perception that an agent creates through past actions about its intentions and norms. The most popular type of reputation system is the propagated reputation, sometimes called majority principle. They are primarily based on feedback, posted by traders after a transaction, and can be positive, negative or neutral.

There are, of course, many issues regarding online auctions, for example:
1) *Risk assymetry of buyers and sellers* - Sellers have little risk, since they require advance payment where as buyers have high risk, since they may not receive goods or receive goods of poor quality . The end result is that buyer reputation is less relevant to sellers than seller reputation is to buyers .

2) *Experience assymetry of buyers and sellers* - Sellers are usually longer in business and participate in more transactions than buyers so they are better known than buyers .

Social Networks

The abrupt world-wide spread of social networks has transformed them into a rich source of personal information about their members. Companies have taken a keen interest in personal profiles from social networks when looking for new partenerships; also the process of selecting freshman applicants has a lot to do with their web profile.

Professional networks, such as LinkedIn, Xing, Naymz, are online communities concerned with interactions and connections of a formal business nature, rather than informal social exchanges, like social networks such as Facebook or Netlog, which are used by their users for pleasure, fun, recreation or for organizing and coordinating groups and events.

These networks often encourage users to endorse other members, so that the online reputation of a member, fundamentally based upon his/her own profile, can be enforced by the judgment that contacts express through the endorsement.

The professional network Naymz stands out because it provides an online algorithm to evaluate the reputation of its members: each time a member establishes a new connection, he/she is asked if he thinks that his new contact is honest, if he would like to be worked with, if he would recommend him for jobs, if he wants to be considered as a reference for the new contact and if he wants to endorse him. These answers increment a *RepScore*, that is the reputation score of the member, on the ground of an algorithm that assigns points to each profile and to each answer, reference, and endorsement got from contacts, the influence of a member on another one being proportional to his own *RepScore*. The *RepScore* is then mapped onto a scale from 1 to 10 *RepScore* levels: thresholds from a level to another one are constantly changing and are dynamically calculated on the ground of the average scores of the Naymz users (Naymz claims to have more than 1 million members).

Naymz's algorithm is therefore an attempt to develop a strategy of people ranking similar to the page ranking model commonly used by search engines to measure the relative importance of each element of a set of links which satisfy a query.

*

Our approach computes the reputation score in a centralized manner, so the trust issues associated with it are not present. There are, still, some problems that could not be avoided, one the most pestering being content fraude. Also, computing all user reputation by a central authority raises important scalability concerns which need to be taken into accout tested, documented and proved to be not threatening to the robustness of the system.

BlobSeer

Context

We designed a method by which we facilitate the end-user's access to the functionalies of a flexible storage environment build especially for large data blocks. For that purpose we created a web interface to handle the user interaction and to issue command to the undelying levels. BlobSeer is offered in a cloud like service model through the PaaS paradigm, our solution standing for a top level layer which shows only the absolute necesary complexity.

Overview

BlobSeer is a binary large object management service with support for heavy access concurrency . It has two sides, firstly, a data sharing service allowing to store massive blocks of data in a distributed, multi-user environment and second, an efficient fine-grain access to the data is provided thanks to distributed, RAM-based storage of data

fragments, while leveraging a DHT- based metadata management scheme, which is natively parallel.

BlobSeer enables efficient versioning of blobs in a highly concurrent environment. In such a context, an arbitrarily large number of clients compete to read and update the blob. A blob grows as clients append new data and its contents may be modified by partial or total overwriting.

Each time the blob gets updated, a new snapshot reflecting the changes and labeled with an incremental version is generated, rather than overwriting any existing data. This allows access to all past versions of the blob. In its initial state, we assume any blob is considered empty (its size is 0) and is labeled with version 0.

Architecture

The service relies on a set of distributed processes communicating through remote procedure calls (RPCs). In a typical setting, each process is running on a different physical node.

Clients. Clients may issue CREATE, WRITE, APPEND and READ requests. There may be multiple concurrent clients. Their number dynamically vary in time without notifying the system.

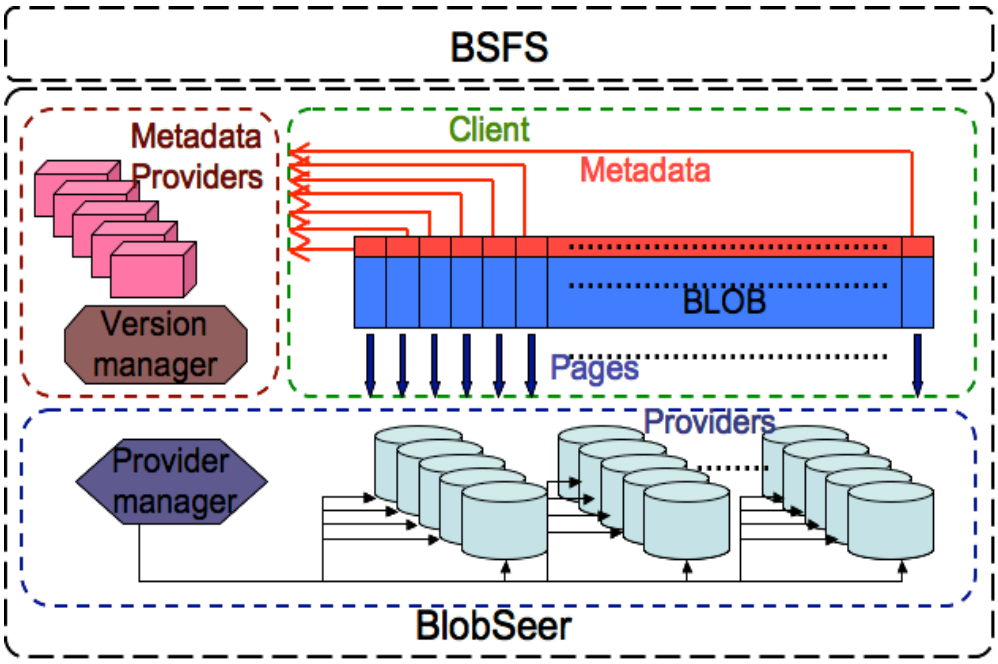
Data providers. Data providers physically store and manage the pages generated by WRITE and APPEND requests. New data providers are free to join and leave the system in a dynamic way.

The provider manager. The provider manager keeps information about the available data providers and schedules the placement of newly generated pages according to a load balancing strategy.

Metadata providers. Metadata providers physically store the metadata, allowing clients to find the pages corresponding to the various BLOB versions. Metadata providers are distributed, to allow an efficient concurrent access to metadata.

The version manager. The version manager is the key actor of the system. It

registers update requests (APPEND and WRITE), assigning BLOB version numbers to each of them. The version manager eventually publishes these updates, guaranteeing total ordering and atomicity.



User Interface

To create a new blob, the CREATE primitive must be used:

```
id = create()
```

This primitive creates the blob and associates to it an empty snapshot 0. The return value represents the *id* of the blob just created and it's guaranteed to be globally unique.

```
vw = write(id, offset, size, buffer)
```

The WRITE primitive is used to generate a new snapshot of the blob (identified by

id) by replacing *size* bytes of the blob starting at *offset* with the contents of the local *buffer*. The call does not know in advance what snapshot version it will generate, but it can be found out after the primitive returns by consulting the value *vw*.

```
va = append(id, buffer, size)
```

APPEND is a particular WRITE, in which the offset is implicitly assumed to be the size of snapshot *va* – 1.

```
read(id, v, buffer, offset, size)
```

A READ results in filling the local buffer with *size* bytes from the snapshot version *v* of the blob *id*, starting at *offset*, if *v* has already been published. If v has not yet been published, read fails.

Other useful functions:

`v = GET_RECENT(id)` – returns the most recent published version of the blob identified by *id*

`size = GET_SIZE(id, v)` – returns the size of the *v* version of the blob identified by *id*

`SYNC(id, v)` – blocks the calling program until version *v* of blob identified by *id* is published

The Reputation System

Concepts

In this section we introduce the notion of reputation and describe a few ways in which it could be implemented. We conclude with our choice and give an objective motivation.

Judging by its most basic meaning, *reputation* is the opinion (more technically, a social evaluation) of the group of entities toward a persone , a group of people, or an organization on a certain criterion. It is an important factor in many fields, such as education, business, online communities or social status. Reputation can be considered as a component of the identity as defined by others.

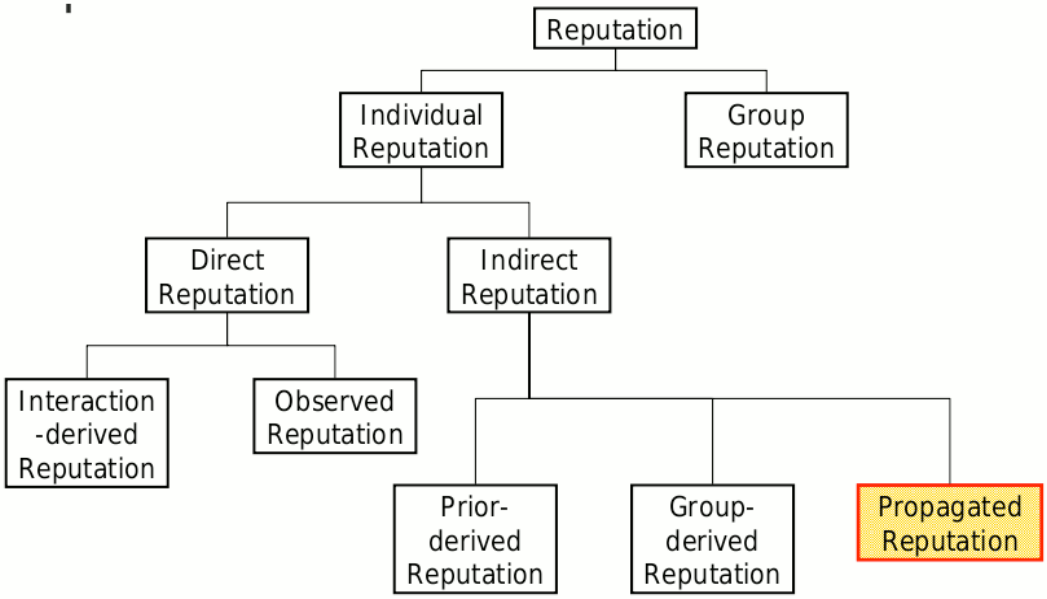
A reputation system computes and publishes reputation scores for a set of objects (such as service providers, services, goods or entities) within a community or domain, based on a collection of opinions that other entities hold about the objects. The opinions are typically passed as ratings to a reputation center which uses a specific reputation algorithm to dynamically compute the reputation scores based on the received ratings.

Reputation, as a concept in the contex of online profiles, represents some sort of measure of an agents fairness and trustworthiness. Reputation of a user arose as a notion in the online environement when a method by which users could differetiate themselves. As in real life, an appropriate, fair behaviour should generate a solid reputation, whereas a malicious, unfair one should rapidly degenerate one's reputation.

The most popular implementation of reputation is within a *reputation system,* which is no more than the particular surroundings reputation is taken into account and decided upon. Since the collective opinion in a community determines an object's reputation score, reputation systems represent a form of collaborative sanctioning and praising.

A few types of reputation systems have been used, as the author of [4] suggests:

Our approach is somewhat simpler than many such reputation systems, because it has a central authority, that does all the user accounting and computes reputation. The closest in meaning would be the direct, observed reputation, but it also features an indirect reputation feature.



Reputation score is another issue in current reputation system design. There are scores represented as binary strings, such as the one described in [1], or as a positive integers. What ever the representation is, it must have a natural, easy to modify and update structure because it is sure to have a very dynamic evolution.

In this thesis we employ a real (float) representation for reputation score, which is permanently updated according to the actions undertaken by the users involved. We belive it's an efficient way to keep track on a user's behaviour and apply certain sanctions, or rewards when and if the reputation score reaches some thresholds.

Architecture

In this section we give a detailed description of the design of our approach along with explinations over the functionalies of the components.

Web Interface. The web interface is all that the user come in direct contact with. It has very simple, intuitive commands, such as download file, upload file, view previous

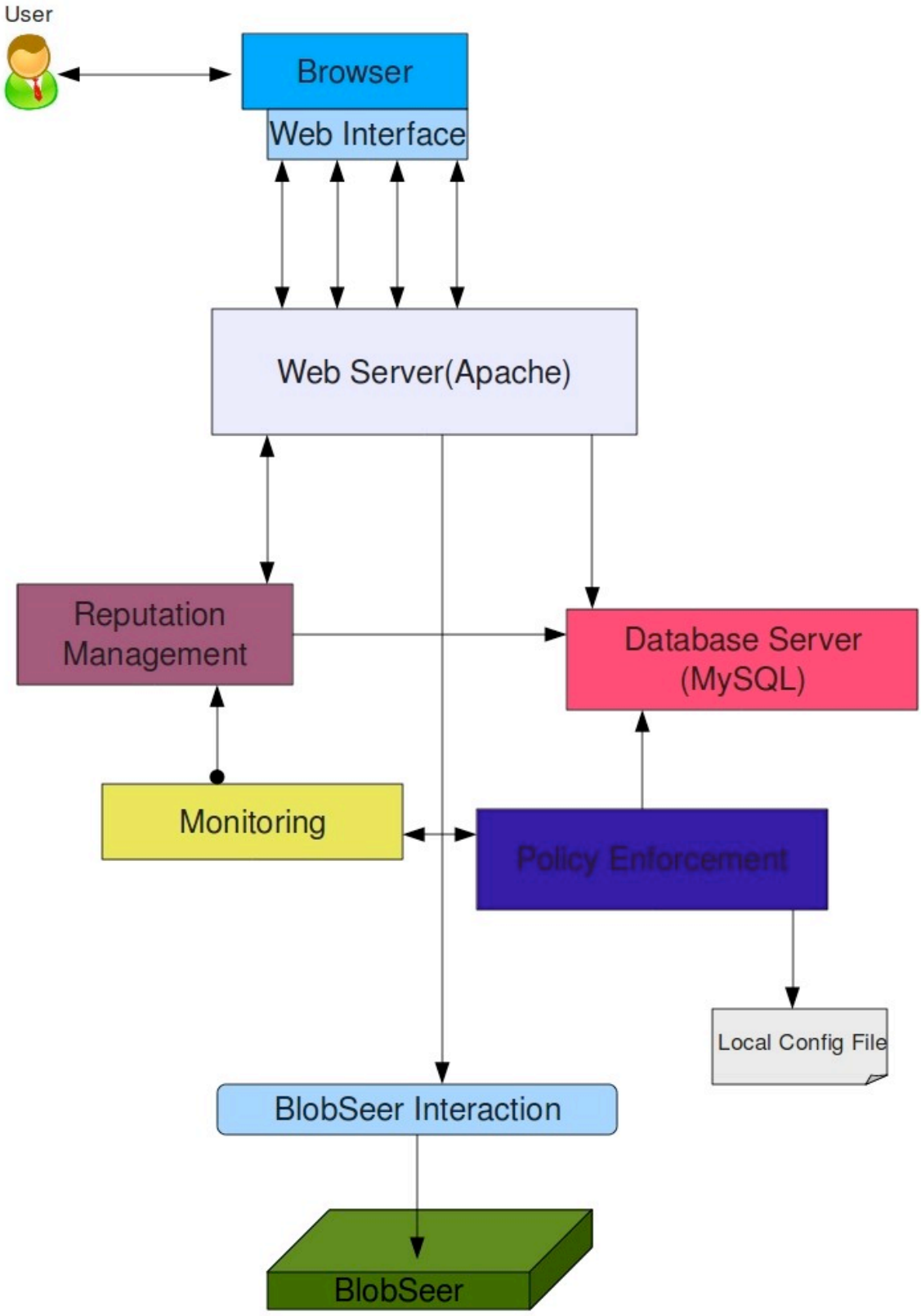
uploaded/downloaded files and view reputation. We aimed at providing the users a means to interact with a distribuited storage service (BlobSeer) without any knowledge of the matter prerequisite.

Web Server. The commands initiated by the user through the web interface are then processed and handled by the web server. This is accomplished with the use of the other modules, presented next.

Reputation Management. The action of a user (upload/download) determines a change in his reputatuon score, and this is the component that does exactly that. It investigates the details of the user's action and computes the new reputation. Then it stores it in the database.

Database Server. This component is used to store various information which will, at some point be requested by another component. The tables are storing information about the users of the system and the files whic currently in store with BlobSeer

Monitoring. This is a small component that runs periodically and does various checking on the users and the files. Its results are used to update reputation score and to set certain flags.


User
Browser
Web Interface
Web Server(Apache)
Reputation Management
Database Server (MySQL)
Monitoring
Policy Enforcement
Local Config File
BlobSeer Interaction
BlobSeer

Policy enforcement. When a user's reputation score reaches some level, be it high or low, certain measures must be taken to reward or sanction the user. When about to apply a policy, it first reads the local configuration file to retrive the method by which the user will be dealt with.

BlobSeer Interaction. When a user uploads a file, it will be written with a BlobSeer WRITE primitive, and when a user downloads a file, a READ primitive will be called. Theese are accomplished with the help of this component, which plays the role of a high level interface for the lower level BlobSeer client interface.

BlobSeer. Interaction with BlobSeer is restricted to commands provided by the previously described component. Files are written and read from BlobSeer whenever a user requests or delivers a file.

Implementation Details

Reputation Computing

As previously stated, we designed our solution so that the reputation score is represented by a float value. This leverages the way we intend to increase/decrease this value according to the user's actions.

Every user, when he/she first created the account and has yet to upload or download something, has a starting reputation score of 1000. After that and begging with the first command to upload or download a file, his/hers reputation will change. We devised a plan to update a user's reputation taking into account three aspects:

1) **The Upload/Download Ratio** – the more a user uploads, the bigger his reputation score gets and vice-versa, the more he downloads, the smaller his reputation will get. This is accomplished by applying the following score adjustments :

Upload

File Size	**Increment** to the Reputation Score
Smaller than 1 KB	1
Between 1 KB and 1 MB	10
Bigger than 1 MB	10 + #MB

Download

File Size	**Decrement** to the Reputation Score
Smaller than 1 KB	0.7
Between 1 KB and 1 MB	7
Bigger than 1 MB	7 + #MB/7

We designed the system so that small files would also have a say in the reputation score, because if we had set the increment/decrement values to be proportional to the size of the file, then it could happen that a user uploading a few tunes in mp3 format a month would have a reputation far greater than a user who is uploading much smaller pdf files every day. Also, we made the decrement smaller than the increment for the same range of file size to ecourage users to upload because they want to share, and not because they are preocupated with the drop in their reputation score.

2) **The quality of the files uploaded –** If a user uploads a file that is very sougth after and is being massively downloaded by other users then he recives a reputation increase for each such download. This encourages users to upload interesting, new and rare content to the system, that way the community being able to provide quality content and help its users find what they seek. We concluded than an increase by 2 of a user's reputation score when another user downloads one of his/her (the former) files is a value that provides equilibrium to the overall score dynamic.

3) **The Frequecy of Use –** The lack of interest of a user should be dealt with somehow, and the method we chose is to diminish his/hers reputation. We do not think it's fair for a user who uploads a file once a month should have the same reputation as a user

who often uploads and downloads. For that matter we designed the system that if a week passes since the user last uploaded/downloaded, it's reputation is decremented by 5.

The measures taken to dynamicaly change users' reputation score are implemented along with the specific action which conceptually generated them. That means that whenever a user executes an action, the system does its background work to compute his/her new reputation.

Upload. When a user uploads a file, a background PHP script is lauched (check_file.php) which is responsible for the following:

- moves the file just uploaded to a more permanent location on the local machine;

- makes a database query to retrive the current user's id, having his/her user name, which is stored in the $_SESSION['MYSESSION'] variable;

- retains the basename of the file just uploaded;

- calls a PHP function that writes the file with a BlobSeer primitive and retains the return value, which is the blob id of the file was stored into;

- makes another database query to insert file information (file name, file size, file type, user id and blob id);

- calls a PHP function to increase the reputation of the current user for uploading

- calls a PHP function to update the last action of the current user, setting it to be the present time;

- after that, if all goes well, the script redirects to the index.php page.

Download. The download process is rather similar, evidently with some different aspects concerting reputation adjustement and database interaction. When a user clicks the download link of a certain file, a PHP script (adddonwload.php) is loaded with the file name and user name passed as variables through the GET method. This script accomplishes several things:

- performs a database query to retrive the requested file's id, size, user and blob id

- updates the current user's last action to the current time

- reads the file with a BlobSeer primitive and stores it somewhere it can easily be served from

- decreases the current user's reputation for downloading

- increases the reputation of the file owner only if is not the same as the current user

- reads the file and serves it to the browser for downloading

The web Interface

The first and single contact the end user has with the system is through this interface, so a fair amount of effort has been put into making it a pleasant experience. Firstly we list the technologies used to design and build the interface:

HTML - HyperText Markup Language

CSS - Cascading Style Sheets

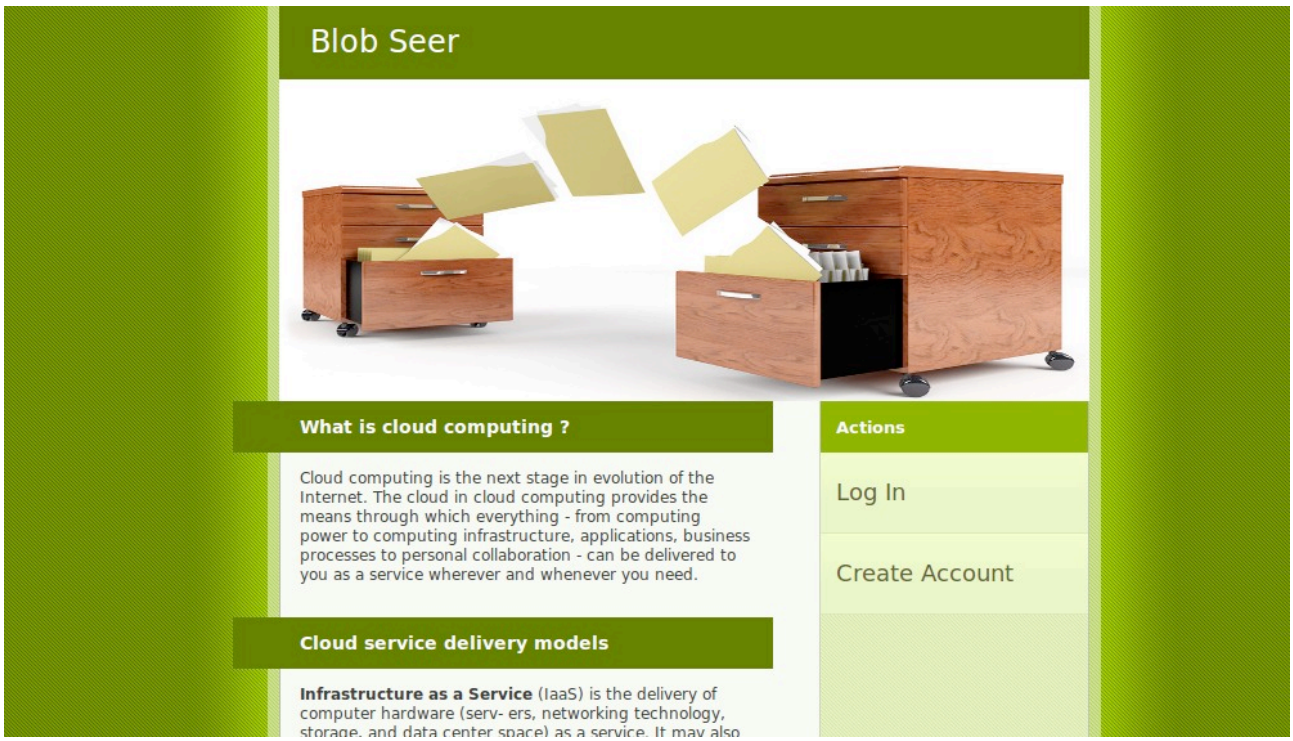
JavaScript – Clien Side Scripting Language [8]

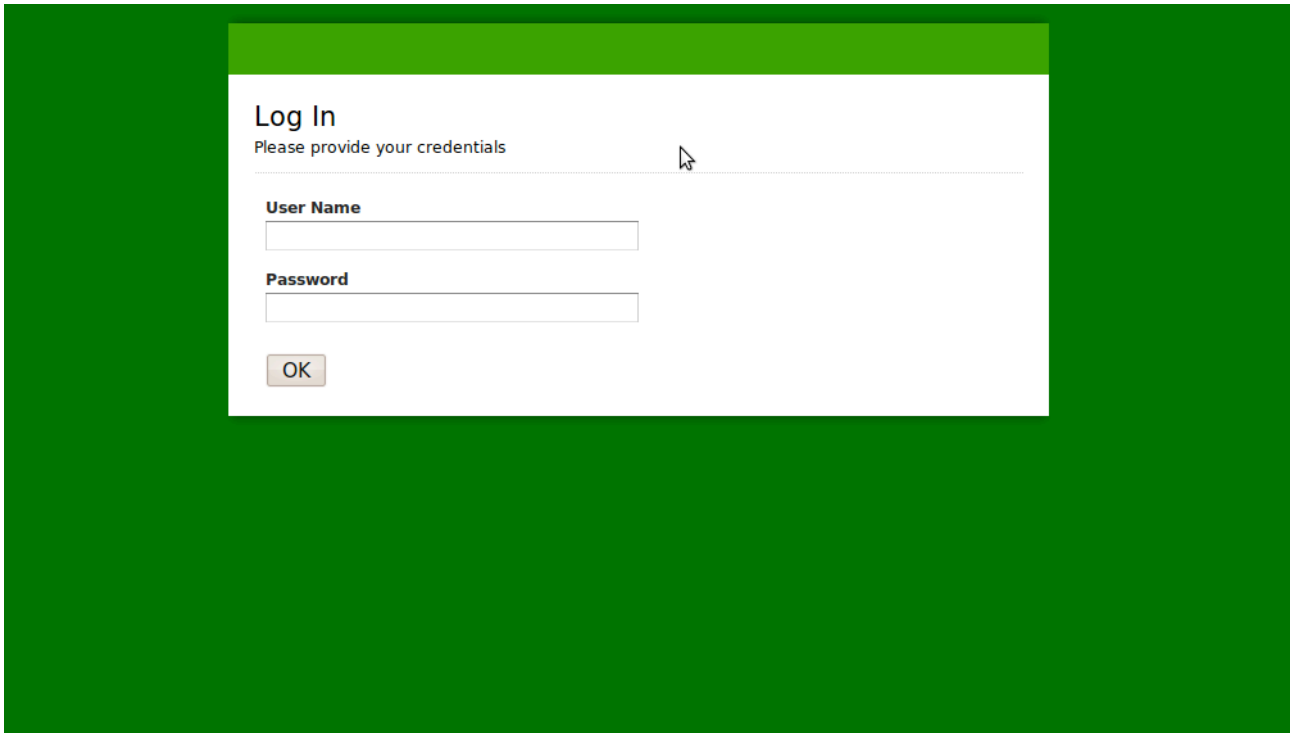
PHP – Server Side Scripting Language [7]

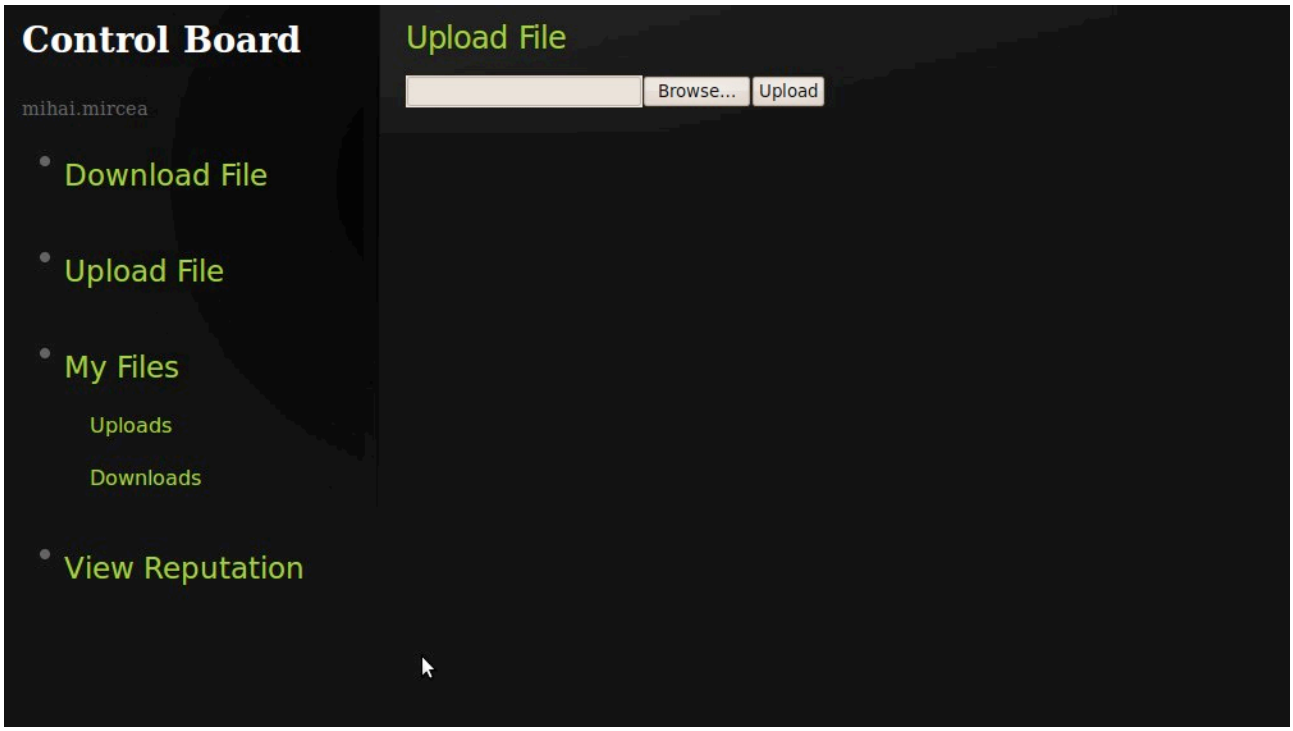
MySQL - Relational Database Management System [7]

Apache – Web Server [7]

The most important pages are the welcome screen, which is the first page loaded, the log in scree and the control board (only for authorized users). We next present these screens in the logical order.


Blob Seer
What is cloud computing ?
Cloud computing is the next stage in evolution of the Internet. The cloud in cloud computing provides the means through which everything - from computing power to computing infrastructure, applications, business processes to personal collaboration - can be delivered to you as a service wherever and whenever you need.
Cloud service delivery models
Infrastructure as a Service (IaaS) is the delivery of computer hardware (serv- ers, networking technology, storage, and data center space) as a service. It may also
Actions
Log In
Create Account


Log In
Please provide your credentials
User Name
Password
OK



The welcome screen has two possible actions, Log In and Create Account. We did not present the create account screen as it is very similar in appearance with the log in screen, and it only asks the user to fill out a form with minimal personal data to create an account.

Once arrived on the log in page the user is asked to provide his credentials (user name and password) before he can access his account. A query is made to the database server to verify the user's credentials and if the user name exists and the password provided and the one retrieved from the database match, than the user is authorized, a session is created and the user is redirrected to the control board.

In the control board, the user can issue four basic commands:

1) **Download File.** A click on this button will reveal on the right side of the page a

list of the currenty held files alog with information such as file name, type and size. On the rightmost column there will be a download button which will initalize a download, after some internal computing will be done.

2) **Upload File.** This button opens an upload form on the right side of the page; The user can then browse for the file he/she wishes to upload and then acctually upload it by hitting the Upload button. A message will apear after the upload has completed, informing the user about the succes/insucces of the operation. Same as with the download button, a few internal operations will be carried out before and after the actual upload. The operations that go along with the download and the upload will be shortly described in detail.

3) **My Files.** This is actually a cover for two subsections, *Uploads* and *Downloads*. Each of them will result in displaying the all the files previously uploaded/downloaded by the current user. This is accomplished with a query to the database server and using a PHP script to display the results.

4) **View Reputation.** As the name suggests, this button will display the reputation score of the current user.

Monitoring

In this section we describe the role played by the monitoring component of the reputation system.

Momentarily, this component has the single purpose of inspecting the last action date-time of the users and reducing the reputation score of users whose last action is older than a week. This is done by using *cron* (a Unix specific time-based job scheduler), a very short bash script and a php script; it's important to metion that a PHP extension is needed to run PHP scrips from the command line, our implementation making use of the *php-cli* package.

A cron job which states that a bash script should run every hour is added to the

current cron jobs list. The bash scrips calls a PHP script called checkUserFrequency.php which effectively cheks out every user who hasn't uploaded or downloaded anything for more than a week. The script employs only a quick database query to handle all the necesary reputation score diminishing.

Database structure

In this section we intend to describe the database design and structure and try to motivate our choices as often as posssible.

The system relies on a MySQL server to handle the queries and store the tables. This was a fairily straightforward decision as PHP has stron support for MySQL and they are both open source products. Our system uses the folowing tables:

User_Data

Name	Type	NOT NULL	AUTO INC.	Flags	Default
id(primary key)	INTEGER	X	X	UNSIGNED	NULL
username	VARCHAR(100)	X			NULL
firstname	VARCHAR(100)	X			NULL
lastname	VARCHAR(100)	X			NULL
email	VARCHAR(100)	X			NULL
password	VARCHAR(100)	X			NULL
reputation	FLOAT	X			NULL
lastaction	DATETIME	X			NULL

File_Data

Name	Type	NOT NULL	AUTO INC.	Flags	Default
id(primary key)	INTEGER	X	X	UNSIGNED	NULL
filename	VARCHAR(200)	X			NULL
filesize	BIGINT(20)	X		UNSIGNED	NULL
userid	INTEGER	X		UNSIGNED	NULL
filetype	VARCHAR(200)	X			NULL
blobid	INTEGER	X			NULL

Downloads_Data

Name	Type	NOT NULL	AUTO INC.	Flags	Default
userid	INTEGER	X		UNSIGNED	NULL
fileid	INTEGER	X		UNSIGNED	NULL

The **User_Data** table stores information about the current users of the system and it is updated in differet contexts. Firstly, a new row is added to the table when a user creates an accout, and has some fields fixed, meaning that it will never be chaged by the system; it's the case for the first six fields: *id*, *username*, *firstname*, *lastname*, *email* and *password*. On the other hand, the *reputation* and *lastaction* fields are constantly changing.

We have already showed when and why the reputation score is updated, be it an increase or a decrease, thus this is just a quick reminder. The *download* action will lead to a reputation decrease (and we stated the details of the decrement) whilst an *upload* will generate an increase (also, we showed the exact amount of this increase). Other events that generate a change in reputation score include a user not executing any action (upload/download) for more than a week, which leads to a decrease in reputation score; an indirect score increase of a user A is triggered by the download by another user B of a file

uploaded by A.

The lastaction field is changed by a specific event. When a user downloads or uploads a file, his/her lastaction is updated to the value of that particular moment. We showed in a previous section the mechanism by which this field is used to detect users who haven't been active for more than a week.

The **File_Data** table stores information about the files in storage and ready to be downloaded by users. Rows are added as files are uploaded and contain information about the file's name, type, size, id of the uploading user and id of the blob. All theese fields are fixed, once a file has been succesfully uploaded and written to a blob, the information associated remains intact in the database.

The data stored in this table are useful in a few situations worth metioning here. Firstly, when a user is downloading a file, it has to first retrive the blob id of the file to pass it as an argument to the BlobSeer Interface *readfile* function. Also when a user is downloading a file it's necessary to retrive it's size to calculate how much will the downloading user's reputation score will be affected; the user id field is needed in the download context to increase the reputation of the user who uploaded the file.

Downloads_Data is a simple, yet very important table, as it stores the downloading history in the form of a user id – file id association. The user id represents the downloading user, and the file id, the file downloaded. This table is primarily used to display the downloaded files of a user when he/she cliks the Downloads button on the web interface. Various other uses can be thought of, statistics, for example, but this thesis only uses this table in the purpose just mentioned above.

Policy Enforcement

This section describes the method we used to reward or sanction our users when their

reputation score will pass certain thresholds.

The purpose of building a reputation system for this service was to apply measures to users with unfair beahviour and to reward users with community-oriented actions. We have already said that it's in the community's best interest to motivate every user to have a good behaviour. Our priority is to offer a rich environment that anyone can use to store and retrive files, so making as sure as possible that users understand and respect this policy has to be the most serious issues we deal with.

In this matter we designed a mechanism so that when the reputation score reaches a fixed value, an imediate action is taken. The policies are described by a XML file, called *policies.xml* which is consulted when the reputation score is changed. The structure of a policy is a logical abstration of answers to the questions:

1) When is this policy applied ?

2) How is this policy applied ?

So far we came up with four such policies:

- When the reputation reaches level 500, restrict the user's bandwidth by a fixed ammount

- When the reputation reaches level 200, lock the download option, thus forcing the user to directly increase his reputation by uploading or wait untill enough user download one of the files previously uploaded by him. The last solution is discouraged because the reputation is degrading anyway if no action is taken for a log period of time.

- When the reputation reaches 0, the user is informed that his accout has been deleted due to unfair behaviour

- When a reputation score reaches 5000 the user becomes a *power-user* and all files uploaded by him now have a trusted source flag.

Dealing With Malicious Users

This section presents the most important threats the system is up against and how

does it deal with each one of them.

Content fraude is a big problem since there is no easy way of determining that the file is what the name says it is. A malicious user can rapidly increase his/her reputation score by uploading a file with a bogus content but with a very interesting file name that might tempt other users to download it. When they will realize that they were tricked into downloading a bogus file, it will be too late, as the system already increased the malicious user reputation. A partial solution would be to let user report such abuses by posting feedbacks on the files they just downloaded, similar to a bit torrent tracker site, (thepiratebay.org, seedpeer.com. etc.). Unfortunately, our sistem does not yet support user feedback, so this remains one the biggest vulnerabilities. In the Future Work section we describe an approach to make the system handle content fraude.

Same file upload represents the action of uploading the same file lots of times to quickly make a reputation increase. It is not a trivial matter because the system cannot forbid the existence of identical files on the server, so another method should be employed. We came up with the plan to forbid the same user to upload the same file more than once; once a user serves a file with a given filename he will not be able to upload the same file ever again. If attemps such an action, he will be reminded that he already uploaded that file and cannot do so again. Despite this same file upload prevention, the user can easily trick the system by changing the file name and then upload it again; this time, the system will not complain and the user will have succesfully, but immoraly increased his reputation score for this upload.

The small world phenomenon is the collaboration between a relativelly small group of individuals who have no real intention of provinding and do not work towards the good of the community. They only supply files which only they realy need and never download files of different interest than their own. We belive it is besides the purpose of our work to track and suspend the activity of such groups if they do not have an abusive behavior and disturb the rest of the users. The system doesn't have anything to loose if lets them function and doesn't get unbalanced by their activity.

Same person, different account is the nickname for the creation of a new account by a user who had his account deleted or will be soon due to low reputation score. This way, he

can use his 1000 points to download until he has 500 or even 200 if he puts up with the decreased bandwidth and then let that account degrade on its own by not accessing it ever again. Instead the malicious user will create a new account and receive 1000 fresh reputation points like any new user created. The system has no imediate way of knowing that a user accout belongs to a person who's previously accout was deleted or having a low reputation.

For this problem we propose a hypothetical solution that has been currently employed before in somewhat similat conditions. A user can only create an accout if he receives an invitation from another user who already has an accout. A user sending an invitation is responsible for the user to whom it gives it. If the new user gets his accout deleted, then the user who gave him the invitation will have his reputation seriously affected and his right to give invitations momentarily suspended. This way honest users who use and need the system will carefully select the ones who get their invitation, as they are not willing to risk any damage to their reputation.

*

At this premature stage the system is very vulnerable to malicious behaviour and keeping the web interface extremely simple has a lot to do with that. More sophisticated ways to maintain the system's equilibrium can be designed and implemented, but for the moment we wanted to demonstrate the functionality and later work more on security.

As a final word to this section and chapter, we belive that security issues will always be present and with a system that has such a strong social inclination such as ours it is very difficult to provide a truly secure and, more important, objective reputation system.

Experimental results

In this section we present the manner in which we tested the validity of our reputation system and show a few illustrations of the ways the sistem evolves over time. Interpretations theese illustrations are also provided. We belive it's truly impossible to properly test the system in a way other than it's open usage because it's rather difficult to

predict the system's complex dynamic due to a high number of unknown variables.

We have to take into account several things when testing the system and also make very clear what is it that we test. In that matter, we came up with the idea to focus on the overall reputation, and see it's evolution. In the next pages we describe how we did that and present the data we obtained.

The testing is done by running a simulation of the entire environment and displaying the results. A PHP script manages exactly that by defining a *ReputationSystem* class that simulates the behaviour of the reputation system, an *User* class which defines the most relevant features, like username and the ability to upload or download files and, finally, a *File* class which encapsulates the important aspects of a file, such as file name, size, and the id of the user who uploaded it.

The script that runs the simulation is called *testingReputation.php* and it can be run in the browser or in console mode; we chose to run it through the browser for better observation by formating the output with HTML features. All this script does is create an instance of the type *ReputationSystem* and call its *run_simulation* method which takes a parameter, the number of days we want the simulation be spread over.

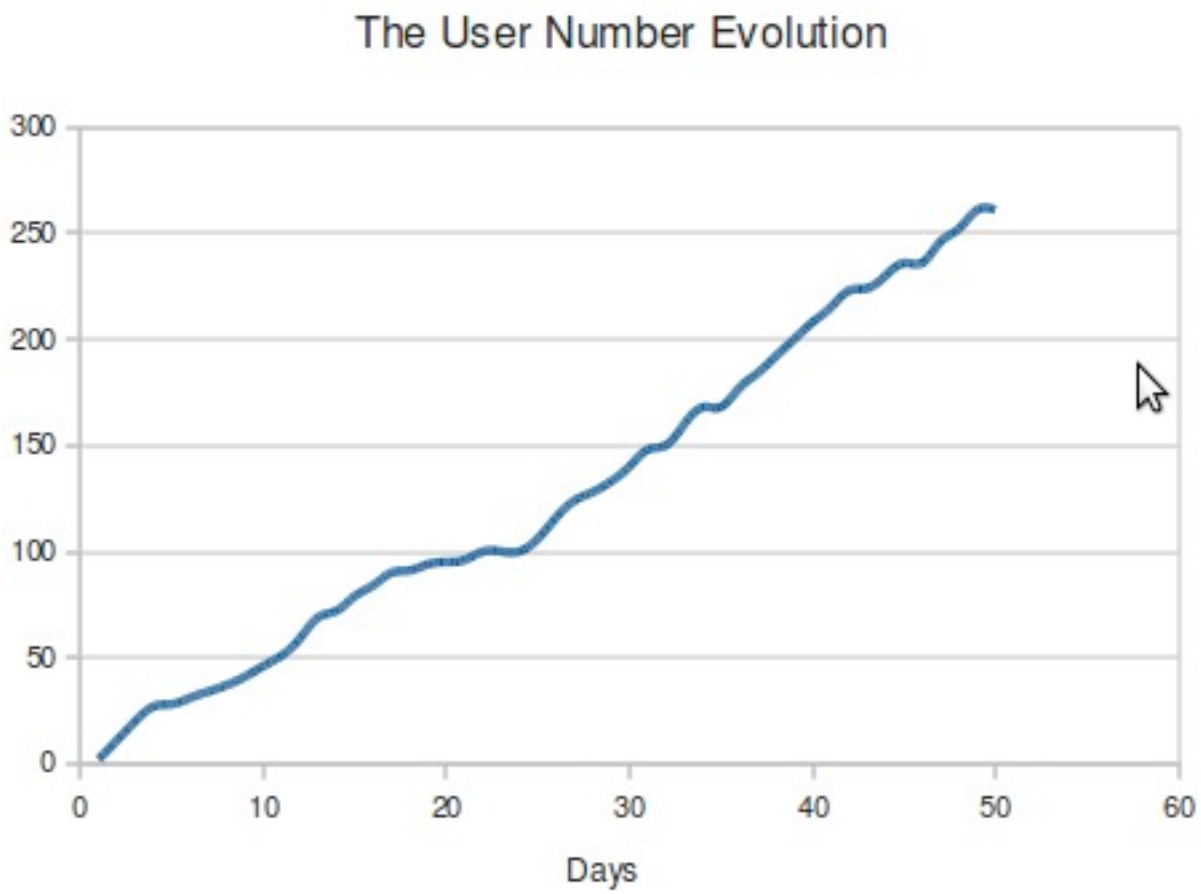
Starting now there will be a sequence of days, each day being filled with a random number of events. Firstly, a random number of users (0 ... 10) will be added to the sytem and then each existing user will execute a total of five distinctive actions. Each action has a equal probability to be an upload, a download, or a null action (we figuered that the user may log on to check his reputation, or look for a file and not find it, so this situation cannot be treated as an action).

The upload action builds a *File* object having a random size, between zero and a hundred megabytes and calls the upload method of the *ReputationSystem* class which adds the file to the list and increases the user's reputation score.

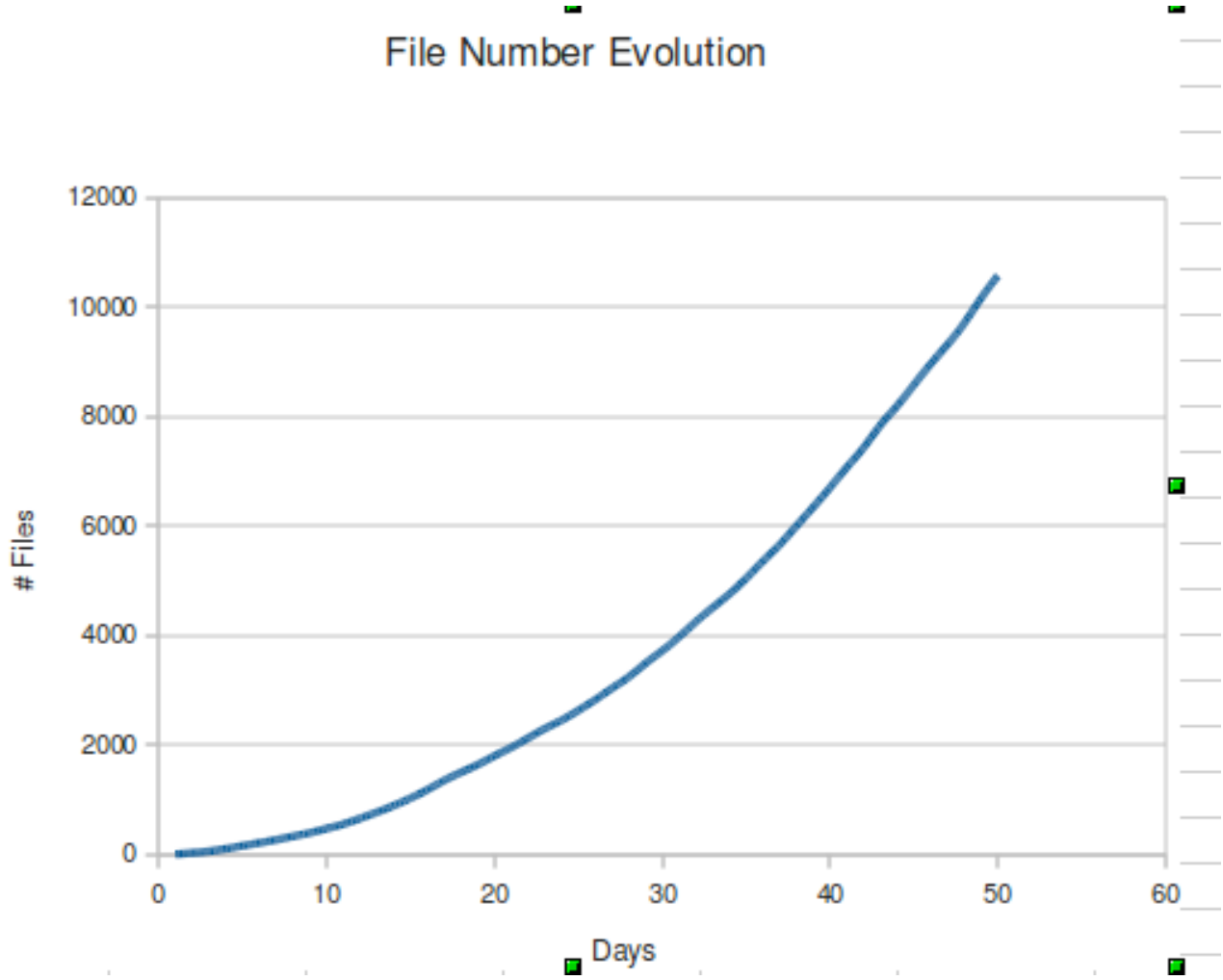
The download action first gets the number of files currently in store on the system and then selects a random one; imediately after that, the ReputationSystem->download method is called which does all the reputation score increasing and decreasing.

We rand a 50 days simulation and collected some data which we think is of fair interest so we created three charts based on this simulation. Next we present the graphical

interpretation of the data obtained (charts) and offer some observation.



The picture above represents how the number of users varies along a period of fifty days. Although it doesn't say much about the system, it does show that even with a small daily increase in user number (a random number from 0 to 10) it quicky starts to pose scalability issues. Within a full year, it will reach such a high number of users that to maintain a robust, functional service the solution of adding another web server and a front end to redirect request to the servers has to be taken into perspective.



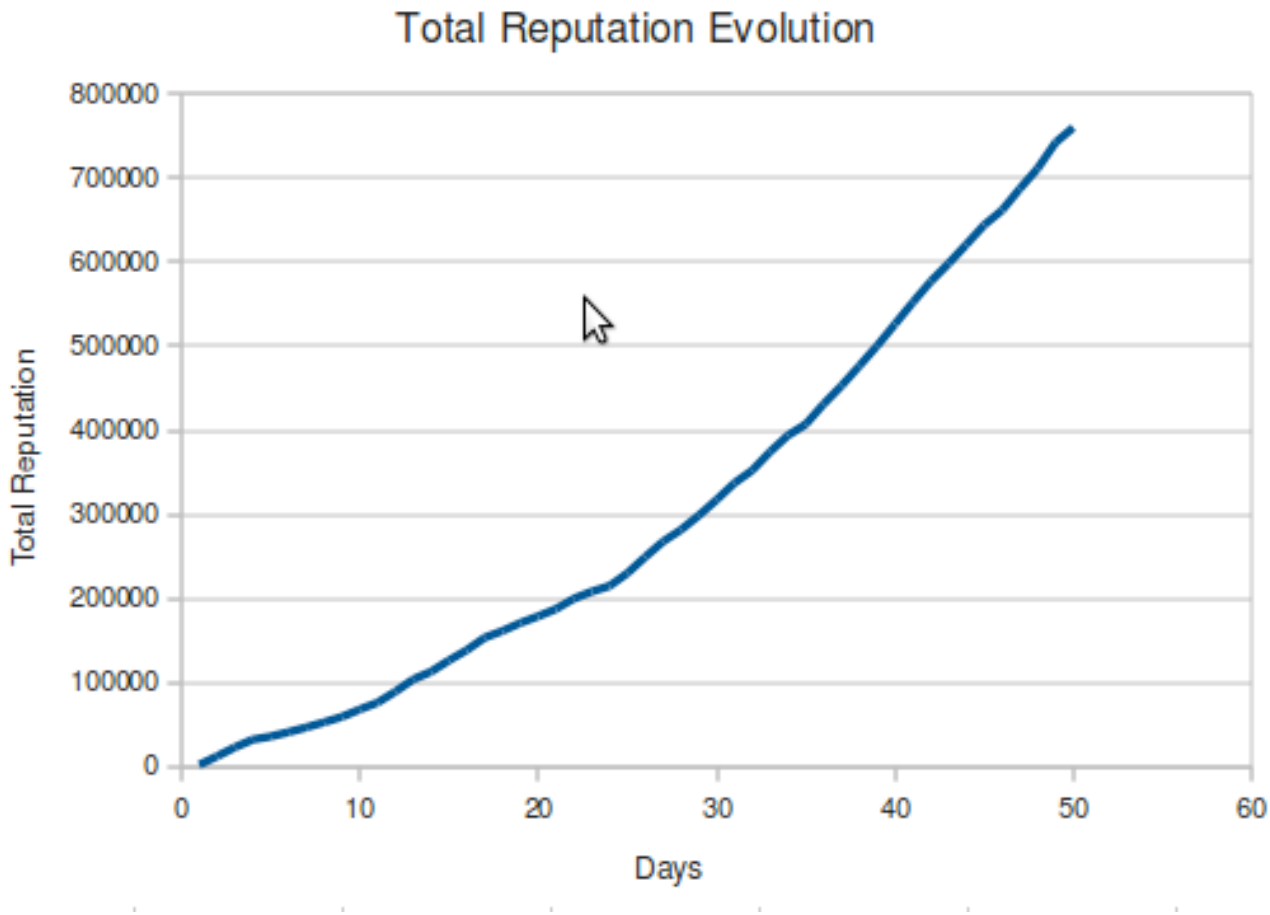
The chart above motivates out previous conclusion, but now focuses on the storage aspect. Scalability on the storage layer is provided by BlobSeer, which we take for granted, however it would be interesting to see the limits of the storage layer with a series of test that focus on latency.

*

Our final chart show the time dynamic of the total reputation score, that is the sum of the reputation score of all current users. It's interesting to combine the data from the first chart where we have, at the end of day fifty, about 270 users. Here, we have a total reputation score of about 760,000 , and by direct divison we see that the average reputation score is an unexpected 2814.

This can mean two things, either our simulation is faulty, in the sense that the users designed do not expose a behaviour similar enough to real life users, or we have to chage

the way our reputation score evolves. It is simply a too big increase in reputation over a short period of time and the system will be over populated with power users. Some one could argue here that the average reputation score is not relevant, because users may have very different reputations, and it's possible and probable that there are some users with very high score, the vast majority with medium score, and those who have low score. The script we implemented for the simulation also has a *verbose* run mode (easily activated by changing a few parameters to true) which shows all the users and their score at the end of each day. The conclusion is that old users develop very high scores, and new users start to do just that; anyway, we belive it's not realistic that not a single user had his score lower than 900.



To conclude this chapter, we belive that further testing is highly needed to find new ways to enahance the performance of the system. A different approach would be to write a script that would associate a thread to each user and let the user decide when and what

action to execute, rather than making all do their actions in a predifined order.

Future Work

In this chapter we present the features that are missing and would be a very welcomed addition to the system or features that are present but would very much benefit from an improvement.

The web interface need a search button to aid the user to find what he needes. If at first it's not a problem, later on, there most certainly will be (we showed in the previous chapter that in fiflty days the system may store as much as 10,500 files). Forcing the user to scroll down such a huge page to look at the files is a sure way to loose that user's respect for the system, not to metion how long ot would take to load the actual page.

Also an interface upgrade would be a graphical mechanism that allows the upload of multiple files at once. If a user wants to upload a few dozens of files which are all in the same folder on his local machine, it's only natural to provide away to load all the files at once.

Since we brough file upload into discussion, it would be worth mentioning here that a more versatile file transfer protocol be used to upload the file(s) to the server than HTTP. Despite its great features, it has overhead that slows down file transfer considerably.

Content fraude prevention through some sort of feedback posting is undoubtedly more a necessity than a bonus feature. A malicious action of uploading a file with a very interesting file name but with a bogus content will dissapoint a lot of enthusiastic downloaders, not to mention illegaly increasing the malicious user's reputation score.

One last idea is to make the reputation algorithm itself adjustable, just like the policies which are to be applied. In this way, and administrator may change, for example, how many points to substract from a user's reputation when he doesn't log on for more than week.

References

[1] Marcus Öjes, *Trust and Reputation Simulations in Peer-to-Peer Network*

[2] Minaxi Gupta, Paul Judge, Mostafa Ammar, *A Reputation System for Peer-to-Peer Networks*

[3] Ernesto Damiani, Sabrina De Capitani di Vimercati, Stefano Paraboschi, Pierangela Samarati, Fabio Violante , *A Reputation-based Approach for Choosing Reliable Resources in Peer-to-Peer Networks*

[4] Adam Wierzbicki, *Trust, Reputation and Fairness in Online Auctions*

[5] Marco Lazzari, *An Experiment On The Weakness Of Reputation Algorithms Used In Professional Social Networks: The Case Of Naymz*

[6] Judith Hurwitz, Robin Bloor, Marcia Kaufman, Dr. Fern Halper , *Cloud Computing For Dummies*

[7] Timothy Boronczyk, Elizabeth Naramore, Jason Gerner, Yann Le Scouarnec, Jeremy Stolz, Michael K. Glass , *Beginning PHP6, Apache, MySQL Web Development*

[8] Danny Goodman, Michael Morrison, *JavaScript Bible 5th Edition*

Appendix. Code snippets

The appendix contains excerpts of code from the most important scripts and classes.

check_file.php

```
<?php
     session_start();
     require_once("DBconnection.class.php");
     require_once("functions.lib.php");
     // Where the file is going to be placed
     $target_path = "../Uploads/";
     // Add the original filename to our target path. Result
is "uploads/filename.extension"
     $target_path = $target_path .
basename( $_FILES['uploadfile']['name']);
     if(move_uploaded_file($_FILES['uploadfile']['tmp_name'],
$target_path)) {
          //connect to data base
          $DB = new DBconnection('localhost', 'bs_user',
'blobseer');
          $connection = $DB->connect('BS_users');
          if($connection != CONNECTION_OK) {
               header("Location: index.php?status=errorDB");
          }
          else {
               $link = $DB->getLink();
               $query = "SELECT id FROM User_Data WHERE
username='" . $_SESSION['MYSESSION'] . "'";
               $result = mysql_query($query);
               $rows = mysql_fetch_assoc($result);
```

```
            $id = $rows['id'];
            $filename = basename( $_FILES['uploadfile']
['name']);
            $blob_id = write_file_BlobSeer($filename);

            if($blob_id != -1) {

                $query = "INSERT INTO File_Data(" .
                    "filename, filesize, filetype,
userid, blobid) VALUES(".
                    "'" . $filename . "'," .
                    $_FILES['uploadfile']['size'] . "," .
                    "'" . $_FILES['uploadfile']['type'] .
"'," . $rows['id'] . ", " . $blob_id . ")";

                $result = mysql_query($query);
                // increase the current user's reputation
for uploading
                increase_reputation($id,
$_FILES['uploadfile']['size']);
                //update the current user's last action
datetime
                update_lastaction($id);
                header("Location: index.php?status=done");

            }
            else {
                header("Location: index.php?
status=errorBLOB");
                    }
        }
```

```
    }
        else {
         header("Location: index.php?status=errorU");
    }
?>
```

adddonwload.php

```
    <?php
    require_once("functions.lib.php");
    require_once("DBconnection.class.php");
    //echo " add my file and current user to the downloads
table";
    if(isset($_GET['filename']) and isset($_GET['username']))
{
         $filename = $_GET['filename'];
         $username = $_GET['username'];
         $filename = unescape_ampersand($filename);
    }
    //connect to data base
    $DB = new DBconnection('localhost', 'bs_user',
'blobseer');
    $connection = $DB->connect('BS_users');
    if($connection != CONNECTION_OK) {
         echo "Database connection problem, nothing could be
done, sorry!";
    }
    else {
         $link = $DB->getLink();
         $query = "SELECT ud.id userid, fd.id fileid,
fd.filesize size, fd.blobid blob FROM User_Data ud, File_Data
```

```
fd WHERE username='" . $username . "' " .

"AND filename='" . $filename . "'";
         $result = mysql_query($query);
         $rows = mysql_fetch_assoc($result);
         $userid   = $rows['userid'];
         $fileid   = $rows['fileid'];
         $filesize = $rows['size'];
         $blobid   = $rows['blob'];

         $query = "INSERT INTO Downloads_Data (userid,
fileid) VALUES(" . $userid . ", " . $fileid . ")";

         mysql_query($query);
         //update the current user's last action datetime
         update_lastaction($userid);
           read_file_BlobSeer($fileid, $filename, $filesize,
$blobid)
         $file = "../Uploads/" . $filename;
         //decrease the current user's reputation for
downloading
         decrease_reputation($userid, filesize($file));

         //increase the file owner's reputation, evidently,
only if the user downloading is the owner
         increase_reputation_indirectly($userid, $filename);
         header("Content-type: application/force-download");
         header("Content-Transfer-Encoding: Binary");
         header("Content-length: " . filesize($file));
         header('Content-disposition: attachment; filename="'
. basename($file) . '"');
```

```
        readfile("$file");
    }
?>
```

checkUserFrequency.php

```
<?php
    require_once("../Account/DBconnection.class.php");
    //connect to data base
    $DB = new DBconnection('localhost', 'bs_user',
'blobseer');
    $connection = $DB->connect('BS_users');
    if($connection != CONNECTION_OK) {
        echo "Database connection problem, nothing could be
done, sorry!";
    }
    else {
        $query = "UPDATE User_Data SET
reputation=reputation-5 WHERE lastaction < (now() - INTERVAL
7 day)";
        $result = mysql_query($query);
    }
?>
```

readfile.cpp

```
#include <iostream>
#include "client/object_handler.hpp"
#include "common/debug.hpp"
```

```
#include <cstdlib>
using namespace std;

int main(int argc, char **argv) {

    unsigned int result,
                 i=0,
                 id;
    unsigned long int size;
    char *file_name,
         *file_buffer;
    FILE *pFile;

    if (argc != 5 || sscanf(argv[2], "%u", &id) != 1 ||
sscanf(argv[4], "%lu", &size) != 1) {
     cout << "Usage: readfile <config_file> <id> <filename>
<filesize>." << endl;
     return 1;
    }
    // retrive the filename
    file_name = (char*)malloc((1 + strlen(argv[3])) *
sizeof(char) );
    strcpy(file_name, argv[3]);
    // open the file
    pFile = fopen (file_name, "wb");
    if (pFile == NULL) {
     fputs ("File Error \\n", stdout);
     fputs (file_name, stdout);
        exit (1);
    }
```

```
    // alloc the file buffer
    file_buffer = (char*)malloc(size*sizeof(char));

    // create the blob
    object_handler *my_mem;
    my_mem = new object_handler(string(argv[1]));

    if (!my_mem->get_latest(id)) {
     cout << "Could not alloc latest version, write test
aborting" << endl;
     return 1;
    }
    else
     cout << "Blob created successfully."
              << endl
              << "\tversion: "   << my_mem->get_version() <<
endl
              << "\tsize: "      << my_mem->get_size() << endl
              << "\tpage size: " << my_mem->get_page_size() <<
endl
              << "\tid: "        << my_mem->get_id() << endl;


    // read back from the blob
    if (!my_mem->read(0, size, file_buffer))
     cout << "Could not read (" << size << ")" << endl;

    // write buffer to file
    fwrite(file_buffer, sizeof(char), size, pFile);
    delete my_mem;
    fclose(pFile);
```

```
    delete file_name;
    delete file_buffer;
    return 0;
}
```

writefile.cpp

```
#include <iostream>
#include "client/object_handler.hpp"
#include "common/debug.hpp"
#include <cstdlib>
using namespace std;

int main(int argc, char **argv) {
    unsigned int page_size,
                        replica_count;
    int blob_id;
    unsigned long int size;
    char *file_name,
         *file_buffer;
    FILE *pFile;

    if (argc != 5 || sscanf(argv[2], "%u", &page_size) != 1
|| sscanf(argv[3], "%u", &replica_count) != 1) {
     cout << "Usage: writefile <config_file> <page_size>
<replica_count> <filename>."     << endl;
     return 1;
    }

    // retrive the filename
    file_name = (char*)malloc((1 + strlen(argv[4])) *
```

```
sizeof(char) );
    strcpy(file_name, argv[4]);
    //cout << "the filename is : " << file_name << endl;
    // open the file
    pFile = fopen (file_name , "rb");
    if (pFile == NULL) {
       fputs ("File Error", stdout);
        return 2;
    }

    // retrive the filesize
    struct stat st;
    stat(file_name, &st);
    size = st.st_size;
    //cout << "file size is : " << size << endl;
    // alloc the file buffer
    file_buffer = (char*)malloc(size*sizeof(char));

    // copy the file into the buffer
    fread (file_buffer, sizeof(char), size, pFile);

    // create the blob
    object_handler *my_mem;
    my_mem = new object_handler(string(argv[1]));

    blob_id = my_mem->create(size, replica_count);
    if (!blob_id) {
     cout << "Error: could not create the blob!" << endl;
     return 4;
    }
    cout << "Blob created successfully."
```

```
          << endl
          << "\tversion: "   << my_mem->get_version() << endl
          << "\tsize: "      << my_mem->get_size() << endl
          << "\tpage size: " << my_mem->get_page_size() <<
endl
          << "\tid: "        << my_mem->get_id() << endl;

    // write to the blob
    if (!my_mem->write(0, size, file_buffer)) {
     cout << "\n\t !! Could not write (" << size << ") !!" <<
endl;
     return 5;
    }
    delete my_mem;
    fclose(pFile);
    delete file_name;
    delete file_buffer;
    blob_id = my_mem->get_id();
    printf("\n%i\n", blob_id + 10000);
    return 0;
}
```