Social Scribe
Our Personal Information At Our Service

Tiago Alexandre Rodrigues Garcia

Master Dissertation in
Information and Software Engineering

Jury
Chairman: Prof. Mário Rui Gomes
Adviser: Prof. Daniel Jorge Viegas Gonçalves
Members: Prof. Bruno Emanuel da Graça Martins

November 2011
Acknowledgments

First of all, I want to thank from the bottom of my heart to my parents, Fernando and Natália Garcia, for keeping supporting and pushing me to go further and do something great with my life. It took a lot of hard work, but I could never have achieved everything I did without having your faith and your sermons to guide me towards the right path. I dedicate this to you.

I want to thank to all my family for always worrying about me and my work, and keeping supporting me.

Thanks to my girlfriend, Inês Carreira, for having patience to withstand my occasional lack of time for her and supporting my work having in mind that it is all for a better future.

A big thank you to all the friends I've had walking down this path with me these years, Marta Teixeira, Mauro Silva, João Pedro Costa, Filipe Leandro, Luís Lemos, Nuno Bento, Bruno Antunes, among others whose names don’t come to mind, but whose help will never be forgotten.

To the never forgotten Miguel Branco and João Aires, you know the two of you have contributed a lot for me to be able to come this far. There wasn’t one single course project I haven’t been with at least one of you two by my side to help me when I needed and to push me forward when I thought I had reached my limit. My biggest and sincere thanks go to you.

I am especially grateful to all the users that took the time and patience to test the system. I know the indexing process can be long and yet you never complained.

Last but definitely not least, thank you Professor Daniel Gonçalves, for advising me and keeping me on track throughout this sometimes painful journey.
Resumo

Hoje em dia, os utilizadores têm a sua informação digital espalhada pela Internet. Antes, tínhamos toda a nossa informação pessoal nos nossos computadores. Com o "boom" da Internet e o aumento do uso de email, começamos a guardar muita informação também nas nossas contas de email. É possível gerir duas fontes, mas entretanto chegaram as redes sociais e outros serviços na forma de aplicações na "cloud" com o propósito de armazenar e partilhar informação online. De repente a nossa informação passou a estar todo o lado.

Apresentamos o Social Scribe, uma aplicação com o propósito de ajudar os utilizadores a encontrar a sua informação independentemente da localização. O Social Scribe é uma aplicação baseada em plugins construída com uma arquitectura flexível para tornar a indexação de novas fontes fácil. O utilizador escolhe que informação indexar e o Social Scribe, usando os seus plugins, obtém-na e indexa-a, mantendo-a num índice único e integrado. Também foi desenvolvida uma ferramenta de recuperação de informação pessoal para, usando o índice, encontrar a sua informação de modo a ajudá-lo a geri-la.

A avaliação feita à ferramenta provou a correcta indexação da informação e que, integrando as várias fontes de informação, pode rápida e facilmente ajudar os utilizadores a obter a sua informação de qualquer lado. Os resultados mostram que quase 100% da informação encontrada foi indexada. Menos de 1% dos anexos de email e dos ficheiros falharam o processo de indexação, e apenas por não terem, nesses casos, um parser para o tipo de ficheiro.

Palavras-Chave

Gestão de Informação Pessoal, Recuperação de Informação Pessoal, Indexação de Informação, Informação Social
Abstract

Nowadays, users have digital information scattered all over the Internet. Before we had our personal information in our personal computers, all kept on the hard drive. With the Internet boom and the growing use of email, we started to also have a lot of information in our email accounts. Two sources are manageable, but then social networks came along, and other services were created as cloud applications with the purpose of storing and sharing personal information on the Internet. All of a sudden, our personal information was all over the place.

We present Social Scribe, an application with the purpose of helping users find the information, no matter what’s its location. Social Scribe is a plugin-based application built with a flexible architecture to make developing the indexing of new sources an easy task. The user chooses what information to index and Social Scribe, using its plugins, fetches and indexes it, keeping it on a unified and integrated index. Furthermore, a personal information retrieval tool was developed along with Social Scribe to, using the built index, find information the user needs to help him or her manage it.

The evaluation performed on the tool showed that information is correctly indexed and that, by integrating the several sources of information, it can help users obtain information from everywhere fast and easily. Results show almost 100% of the found items make it to the index. Less than 1% of email messages and files fail to be indexed and it’s only attachments and files for which the source has no implemented parser.

Keywords

Personal Information Management, Personal Information Retrieval, Information Indexing, Social Information
Contents

1 Introduction 1
  1.1 Contributions ...................................................... 4
  1.2 Document Structure ............................................... 4

2 Related Work 5
  2.1 Lifestreams ......................................................... 5
  2.2 TimeScape ............................................................ 7
  2.3 Haystack ............................................................. 8
  2.4 MyLifeBits ........................................................... 9
  2.5 Stuff I’ve Seen ..................................................... 10
  2.6 Semex ................................................................. 11
  2.7 Connections ......................................................... 13
  2.8 Apple Spotlight .................................................. 14
  2.9 Google Desktop ................................................... 15
  2.10 Beagle++ ........................................................... 16
  2.11 Phlat ................................................................. 17
  2.12 Quill ................................................................. 19
  2.13 Jourknow ........................................................... 20
  2.14 SeeTriever .......................................................... 22
  2.15 DEVONthink ...................................................... 23
  2.16 YouPivot ............................................................ 24
  2.17 Discussion .......................................................... 25

3 Architecture 31
  3.1 Scribe ................................................................. 31
    3.1.1 Scribe’s Architecture .......................................... 31
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.1.2</td>
<td>Scribe</td>
<td>33</td>
</tr>
<tr>
<td>3.1.3</td>
<td>Plugins</td>
<td>35</td>
</tr>
<tr>
<td>3.1.4</td>
<td>Nibs</td>
<td>48</td>
</tr>
<tr>
<td>3.1.5</td>
<td>Nobs</td>
<td>51</td>
</tr>
<tr>
<td>3.1.6</td>
<td>Accesspoint</td>
<td>54</td>
</tr>
<tr>
<td>3.1.7</td>
<td>Parchment</td>
<td>54</td>
</tr>
<tr>
<td>3.1.8</td>
<td>Database</td>
<td>55</td>
</tr>
<tr>
<td>3.2</td>
<td>Changes to Scribe</td>
<td>55</td>
</tr>
<tr>
<td>3.2.1</td>
<td>File Indexer</td>
<td>55</td>
</tr>
<tr>
<td>3.2.2</td>
<td>Email Indexing Process</td>
<td>58</td>
</tr>
<tr>
<td>3.2.3</td>
<td>Other changes</td>
<td>59</td>
</tr>
<tr>
<td>4.1</td>
<td>Social Plugins</td>
<td>60</td>
</tr>
<tr>
<td>4.1.1</td>
<td>Twitter</td>
<td>61</td>
</tr>
<tr>
<td>4.1.2</td>
<td>Facebook</td>
<td>61</td>
</tr>
<tr>
<td>4.2</td>
<td>Social Nib</td>
<td>64</td>
</tr>
<tr>
<td>4.3</td>
<td>Social Scribe’s front-end</td>
<td>68</td>
</tr>
<tr>
<td>4.4</td>
<td>Social Scribe’s Retrieval Tool</td>
<td>68</td>
</tr>
<tr>
<td>4.4.1</td>
<td>Architecture</td>
<td>74</td>
</tr>
<tr>
<td>5.1</td>
<td>Testing the Indexing Process</td>
<td>76</td>
</tr>
<tr>
<td>5.2</td>
<td>Testing the Retrieval Tool</td>
<td>77</td>
</tr>
<tr>
<td>5.3</td>
<td>Tests Results</td>
<td>78</td>
</tr>
<tr>
<td>5.3.1</td>
<td>Indexing process</td>
<td>79</td>
</tr>
<tr>
<td>5.3.2</td>
<td>Retrieval Tool</td>
<td>88</td>
</tr>
<tr>
<td>6</td>
<td>Conclusions and Future Work</td>
<td>94</td>
</tr>
</tbody>
</table>
List of Figures

1.1 PC hard disk capacity (in GB). The plot is logarithmic, so the fitted line corresponds
to exponential growth. Data from http://en.wikipedia.org/wiki/Hard_disk_drive. . . 2
1.2 Social Scribe gets information from several sources, prepares it and returns a consis-
tent and integrated. ................................................................. 3

2.1 Lifestreams user interface. ....................................................... 6
2.2 A timeline view of TimeScape. .................................................... 7
2.3 Haystack user interface. ............................................................ 8
2.4 MyLifeBits timeline view. ......................................................... 9
2.5 Stuff I’ve Seen interface with the top view. ................................. 11
2.6 A screenshot of Semex interface ............................................... 12
2.7 A search with the Apple Spotlight. ............................................. 14
2.8 A search with the Google Desktop ”Quick Find” interface. ............... 15
2.9 The Beagle search interface. ..................................................... 17
2.10 Phlat’s UI ............................................................................ 18
2.11 The Quill interface. ............................................................... 20
2.12 The Jourknow interface. .......................................................... 21
2.13 DEVONthink search interface. .................................................. 23
2.14 YouPivot screenshot showing an example of sympathetic highlighting. ...... 24

3.1 Depiction of Scribe’s architecture. ............................................. 32
3.2 Depiction of Scribe’s layer architecture. ....................................... 33
3.3 Flowchart of Scribe module execution. ........................................ 34
3.4 Flowchart of the One Time File Indexer plugin execution. ................. 36
3.5 Flowchart of the IMAP 4 plugin execution. .................................... 38
3.6 Flowchart of the POP 3 plugin execution. ...................................... 40
3.7 Flowchart of the SMTP plugin execution. ........................................ 42
3.8 Flowchart of the Windows Mail plugin execution. ........................ 45
3.9 Flowchart of the Outlook plugin execution. ................................. 46
3.10 Flowchart of the Thunderbird plugin execution. ......................... 47
3.11 Flowchart of the execution of the index_file function of the filenib. 49
3.12 Flowchart of the execution of the index_email_document function of the mailnib. 51
3.13 Flowchart of the execution of the index_attachment function of the mailnib. 52
3.14 Flowchart of the execution of the index_email function of the mailnib. 53
3.15 Flowchart of the preparation by the File Indexer of a file to be indexed by the filenib. 56
3.16 Flowchart of the File Indexer plugin. ......................................... 57

4.1 Social Scribe’s logo. ................................................................. 60
4.2 Flowchart of the execution of the Twitter plugin. ....................... 62
4.3 Flowchart of the execution of the Facebook plugin. .................... 63
4.4 Flowchart of the execution of the functions used to prepare social items to be indexed. 66
4.5 Flowchart of the execution of the index_socialitem function. ....... 67
4.6 Screenshot of Social Scribe’s front-end. .................................... 69
4.7 Screenshot of Social Scribe’s retrieval tool. We can here see the textbox where terms are entered for search and the look of the results. .......... 70
4.8 Screenshot of Social Scribe’s retrieval tool. We can see the result’s paging here. ... 71
4.9 Screenshot of Social Scribe’s retrieval tool. We can see here an open result. ... 71
4.10 Screenshot of Social Scribe’s time of indexing report. ................. 71
4.11 Screenshot of Social Scribe’s indexed amounts report .................. 72
4.12 Screenshot of Social Scribe’s creation timeline report ................. 72
4.13 Screenshot of Social Scribe’s creators graphs report. ................. 73
4.14 Screenshot of an easier example to analyze of Social Scribe’s creators graphs report. 74
4.15 Depiction of Social Scribe’s retrieval tool’s architecture. ............ 75

5.1 Average and standard deviation for the time of indexing (in seconds) values. ... 80
5.2 Individual times of indexing, by user, per source. ....................... 81
5.3 Individual numbers of items found, by user, per source. .............. 82
5.4 Individual numbers of items indexed, by user, per source. .......... 83
5.5 Average and standard deviation for the number of found items values. .... 84
5.6 Average and standard deviation for the number of indexed items values.

5.7 Average and standard deviation for the percentage of indexed items considering the found items.

5.8 Average and standard deviation for the time (in seconds) of indexing per item.

5.9 Average and standard deviation for the times of execution of each task.

5.10 Average and standard deviation for the errors while performing each task.

5.11 Average and standard deviation for each answer to the SUS test.
List of Tables

2.1 First table with the summary of the features and characteristics of the systems presented on the previous section. 27

2.2 Second table with the summary of the features and characteristics of the systems presented on the previous section. 28

5.1 Average and standard deviation for the time of indexing values in seconds. 79

5.2 Average and standard deviation for the number of found items values. 80

5.3 Average and standard deviation for the number of indexed items values. 84

5.4 Average and standard deviation for the percentage of indexed items considering the found items. 85

5.5 Average and standard deviation for the percentage of indexed items considering the found items. 87

5.6 Average and standard deviation for the times of execution of each task. 89

5.7 Average and standard deviation for the errors while performing each task. 91

5.8 Average and standard deviation for each answer to the SUS test. 92
Chapter 1

Introduction

These days, we, as computer users, stopped having our personal digital information sitting at one location at our disposal in small amounts and easy to find. First of all, the main storage location (hard drive) kept growing exponentially in terms of storage space for many years, as we can see in Figure 1.1. This allowed also a big growth in terms of used storage space, since users experienced an increase in stored digital items. With the appearance of digital photographic and video cameras, the number of video and photo items on a person’s computer increased immensely. The possibility to download movies also contributed for the growing in video items on a person’s computer. More than that, as time went by and the digital video quality kept improving, each item kept growing in used storage space. The downloading of music using software like iTunes also played a big part on the increasing used storage space in hard drives across time. All that, plus the growing use of peer-to-peer networks to share big amounts of files across the world, together with the growing digitalization of documents, replacing the classic paper copies with digital versions using formats like .pdf, resulted on spacious hard drives filled with lots of information that is many times hard to browse and find what an user wants without the help of a tool.

Then, the growth of email came in a strong and fast manner. Email information became, like the hard drive information, too much to find a particular item, even when a good organization policy is applied. While six years ago Hotmail offered a 2MB account for a user to store emails, today Google offers its users over 7,5 GB in space on Gmail accounts. Actually, in current times, the use of email is close to irrational and numbers prove that people consider it an indispensable tool in their lives for either business or personal affairs. According to numbers released by Royal Pidgin concerning the 2010 use of email, already excluding the 89% considered spam, there were over 32 billion email messages sent every day. It is estimated that the number of email users is about 1,88 billion around the world. With a few calculations is easy to understand that the average person gets around 17 email messages everyday, which makes about 6000 messages in the whole year. This number is an average, so there are a lot of people that probably get much more than that.

Finally, cloud computing and social networking came along allowing users to do new things but having their personal information scattered all over the place, physically and digitally. Users were given the opportunity to share photos using Flickr and video using Youtube, amongst other cloud applications, and embraced it. Each of these systems has a great amount of users, with Flickr having around 32.000.000\(^1\) and YouTube having possibly more. YouTube videos are greatly used in social actions\(^2\), having at least 17 million YouTube users have associated their YouTube accounts with other social networks’ accounts. At least 12 million people share YouTube videos on social networks. 100 million people perform social actions on YouTube every week. All this shows how

\(^1\)http://en.wikipedia.org/wiki/List_of_social_networking_websites
\(^2\)http://www.youtube.com/t/press_statistics
Figure 1.1: PC hard disk capacity (in GB). The plot is logarithmic, so the fitted line corresponds to exponential growth. Data from http://en.wikipedia.org/wiki/Hard_disk_drive.

some cloud applications are really widely used. And we haven’t even discussed the social networking world we currently live in.

People, as time went by and technology allowed it became surprisingly and significantly attached to computers, the Internet and social networks to communicate, meet people, share personal information, etc. For example, two of the most used social networking websites in the world are Facebook, which currently has a collection of more than 800 million active users\textsuperscript{3}, and Twitter, that has over 200 million accounts created\textsuperscript{4}. Two different applications with different purposes, but still used so much that lots of used have lots of information on their accounts.

Twitter allows only microbloging, being used more for communicating messages than anything else. Still a lot of information is created on Twitter and currently a billion tweets are written every week\textsuperscript{5}, which means that an average user tweets 5 messages per week. Although this is an average value, it is not hard to find some accounts with daily tweet values of two digits. It can be a lot of information per user.

Facebook on the other hand is more complex and allows much more than Twitter. More than creating messages, it allows to share personal information like videos and photos with friends and the community and ultimately comment that content. The average user produces 90 pieces of content each month, which is a lot and only shows that nowadays personal information comes in all shapes and forms and from anywhere to everywhere.

All these facts that came along with the evolution of computer science, the Internet and technology, got our personal information a lot bigger in size and everywhere. Managing all this information became a nearly impossible task because finding what we are looking for became almost as hard as finding a needle in a haystack. Thankfully, it is not exactly needed to browse every source of information we have to find what we are looking for, because most sources (operating systems,

\textsuperscript{3}http://www.facebook.com/press/info.php?statistics
\textsuperscript{4}http://en.wikipedia.org/wiki/List_of_virtual_communities_with_more_than_100_million_users
\textsuperscript{5}http://blog.kissmetrics.com/twitter-statistics/
email clients, social networks, etc.) have search tools to ease the information retrieval process. But still, now we are required to go source by source trying to find this information.

Figure 1.2: Social Scribe gets information from several sources, prepares it and returns a consistent and integrated.

The questions below are examples of problems that to be solved, nowadays, a user needs to go source by source and search it, if possible, or browse it.

- "I know I got this message from that friend of mine, but was it via email, Twitter or Facebook?"
- "I know I have this photo stored somewhere, but is it on my hard drive, as an email attachment, on a Facebook album or on my Flickr?"
• “I know I stored that video somewhere, but was it on my hard drive or on YouTube?”

Information retrieval, when done right, can be really hard work and take too much of the users’ time. Wouldn’t it be great if this problem could be fought? Social Scribe’s main purpose is to integrate users’ personal information from everywhere in one local index.

1.1 Contributions

We developed a solution to fight information’s fragmentation (one user’s personal information scattered through various information locations and cloud applications) by the creation of an unified index. Our work resulted in:

• An improved and more stable version of the information indexer Scribe, previously created to index email and hard drive information.

• Creation of a front-end user interface to allow the user to enter and change all the information needed for the indexer to start and to make Scribe more usable and user friendly.

• Development of social plugins to allow information from social networks to be indexed.

• Social Scribe, a tool that builds an unified, consistent and integrated index for a given user’s personal information (Figure 1.2).

• Retrieval tool that uses the information in the Social Scribe’s index to find information the user is looking for hence saving the immense time browsing would take.

1.2 Document Structure

This document’s structure is really simple. The first thing to do was verify if the idea was new and what was done in the area, so a survey of the state of the art is done and presented on Chapter 2. After that, the solution’s architecture is presented on Chapter 3, presenting specially the original Scribe, whose structure and architecture was used to develop Social Scribe. The next topic discussed is the Social Scribe solution itself, in Chapter 4, including changes made to the original Scribe. And finally, before presenting conclusions and discussing future work on Chapter 6, the evaluation process performed on the Social Scribe solution along with the obtained results is presented and discussed on Chapter 5.
Chapter 2

Related Work

On this section the projects that are relevant to this theme will be presented. Some older, some more recent, but all of them have something in common: make a user’s life easier when it comes to manage one’s digital personal information, specially on retrieval activities.

Throughout this research there was some common ground among the many papers read. Many of them mention Vannevar Bush’s vision\cite{Bush} when in 1945 he first introduced the Memex, a system where a user would store everything (books, documents, etc.) and that would also possess a very powerful searching system to find this information really fast. Bush actually predicted (or maybe inspired) many kinds of technology like hypertext, personal computers, speech recognition or even the Internet along with online encyclopedias. For many people, Memex was the starting point to the development of this kind of systems, but since that time many work has been done on that area. They do differ on other aspects, as we’ll see in the following sections.

Throughout this state-of-the-art section, the work considered relevant will be presented in chronological order of release, so we can see the evolution.

2.1 Lifestreams

More than just a system or tool, Lifestreams\cite{Lifestreams} is presented as an alternative to the desktop metaphor. It uses a time-ordered stream as a storage model and stream filters to perform tasks of organization, location summarization and monitorization of any incoming information.

Lifestreams is based on the following ideals:

1. **Storage should be transparent** - Naming files is unneeded overhead and users should not be forced to attribute names to documents.

2. **Directories are inadequate as an organizing device** - Systems force users to categorize documents and files to put them on their respective directories.

3. **Archiving should be automatic** - Archiving is hard for some users that even consider throw out information instead of archiving it.

4. **The system should provide sophisticated logic for summarizing or compressing a large group of related documents of which the user wants a concise overview**

5. **Computers should make ”reminding” convenient** - Users should be able to use reminders without the need of external applications.

6. **Personal data should be accessible anywhere and compatibility should be automatic** - Regardless of the user’s device or the data location it should be accessible and viewable.
The documents are organized on a time-ordered stream that has every document a user creates. At the tail of the stream are the older documents, the tail is the past. Moving away from the tail we start getting to the present. There are the documents the user is working at the time and new e-mails received. Beyond the present we get to the future. This is a very interesting feature. Documents in the future represent things the user will need, like reminders, to-do lists, etc.

The system's basic operations are new, clone, transfer, find and summarize.

"New" creates a new document while "clone" duplicates an existing one. After the creation, the document is added to the stream.

"Transfer" also duplicates a document but the duplicate is placed on a different stream. Sending an e-mail is transferring a document to another user's stream, printing is transferring a document to the printer, etc.

"Find" creates a substream that is a subset of the user's stream but including only the files that match the query.

Finally, the "summarize" operation takes a substream and presents an overview of it. Depending on what the documents represent the "summarize" operation may have as an output a chart or graphic, a to-do list, etc.

**Strengths and Weaknesses**

The use of chronology as a storage model can be seen as strength if we think that time is a natural guide to experience. That can be why in other systems results of a search are more often sorted by date. The fact that the stream is time-ordered adds historical context to a document.

A great weakness could be the adaptation time for users that are not familiar with this metaphor. Even though the metaphor could possibly be better, users are very used to the traditional Desktop
metaphor.

2.2 TimeScape

TimeScape[15] is a system that was created as an approach to time-machine computing. The application is a time-machine for the desktop environment. It is like a whole new metaphor.

In the environment provided by this system there are no folders. Everything is placed directly on the desktop being possible to spatially organize the information, like when grouping related objects. TimeScape provides time-traveling of the desktop through the manipulation of a time-travel dial, represented like a normal operative system clock. When that clock is manipulated the desktop time changes and the desktop from the time chosen is presented. Event-based time navigation is provided through "go back" or "go forward" commands that makes the system travel to the last or next event that altered the desktop, and the selection of an object and traveling to when it was created or deleted.

Users can remove files by putting them on the trashcan, but they aren’t permanently removed from the hard drive. Files are indefinitely retained to allow desktop restoration when time-traveling. There is however a way to tell the system explicitly to permanently remove an object.

Traveling to the future is supported and creating schedules in the future creates reminders.

The information is displayed using a timeline view where the user can change the scale to view more or less documents.

The system allows the creation of post-it’s to attach to items, like annotations. This is a great support for the search features because it associates metadata to those items that may not have text support to enhance the search, like images. To search for something, keywords are introduced followed by pressing the forward or the back button to travel through time until the system finds a result.

Strengths and Weaknesses

A problem is that even though deletion of an object doesn’t delete it from the hard drive to allow presenting the desktops from the past this means that disk space must be massive on some cases. It’s true that in the last years hard drive space has got bigger and bigger, but also did file sizes. The option of complete deletion on demand helps but if used to much destroys the purpose of the application. On the other hand it needs to be used some times often to clean disk space. This brings another problem: cleaning the hard drive. Many people clean their hard drives sporadically but deleting several files at once. This may be hard when the files are not only all over the place but also "all over the time".
Also, as far as search goes, it does not seem a good method to keep advancing or going back through time, result by result. This way it could take forever to find the result that the user is interested in. It looks more like a way of browsing than searching.

Nonetheless, travel through time seems like a possible good way of browsing, because there are studies on temporal locality that show that it is stronger than spatial.

2.3 Haystack

The idea behind the Haystack[1, 14] project is that every user has his own way to work and manipulate his information. Each person has different preferences when it comes to:

- Which information should be stored, viewed and retrieved
- What relationships or attributes are worth storing to help find information later
- How relationships between information objects should be presented and manipulated when viewing them and navigating through the information space
- How information should be gathered into coherent workspaces in order to complete a given task

If we think of a storage system like a library (kind of standard), Haystack is like our own private bookshelf, organized our own way. When not purposely organized, there is a "latest" read kind of organization. This system was also created to adapt to its user, instead of the other way around.

Even though it is said that Haystack is not a project about information retrieval per se[1] search is very important because is the key activity for information management, and because of that the

![Haystack user interface.](image)

Figure 2.3: Haystack user interface.
developers focused on making users see this activity as a no-overhead activity that is done regularly as part of the navigation. To make this activity better ways of performing fuzzy browsing were also researched.

**Strengths and Weaknesses**

The notion of "view prescriptions" is a great strength because not only do they describe how different types of information should be presented, but the fact that they’re customizable is a great example of how the system adapts to the users. The use of RDF brings several advantages to Haystack, like the fact that it can be used without any schemata for storage and the use of URI provides a useful naming scheme that is independent of the location.

One weakness of this system is UI ambiguity. When clicking a given point on the screen is sometimes hard to know which object a user is addressing, because that point is contained in several nestings of hierarchically displayed objects. Certain decisions and assumptions were made to try to solve the problem. Another possible weakness is the amount of power users are given. The system does not provide much protection against users, allowing them to perform operations that could destroy the information.

### 2.4 MyLifeBits

This system’s creators say themselves that this is a project to fulfill the Memex[3] vision. MyLifeBits[9] was built on four principles:

1. Collections and search must replace hierarchy for organization
2. Many visualizations should be supported
3. Annotations are critical to non-text media and must be made easy
4. Authoring should be via transclusion

![Figure 2.4: MyLifeBits timeline view.](image)
"The MyLifeBits system is designed to store and manage a lifetime’s worth of everything".[10] The developers found that a single hierarchy is too constraining because objects may belong to several categories. Computer file names mangle several concepts so to solve this problem it became allowed to have an item assigned to more than one category, allowing hierarchy but not enforcing it. Annotations are very important because text-based search is very hard, if not impossible, on certain objects (like images and videos) when they are not annotated. Stories are also easily created with the use of Interactive Story By Query (ISBQ) that lets users make queries and drag and drop the results to a story.

A very important notion on this system is "story". Stories are the most valuable form of annotation because they’re something easier for users to remember. Objects can be linked to create a story, and this way users easily understand the context of the objects by following the links. Commentaries on the content are also linked to the objects in question.

**Strengths and Weaknesses**

A big strength about this system is that it was created with the belief that the way to view the media should not be restricted and several ways to view information were implemented. Also making annotations can be boring, so many features were implemented to make it as easy as possible. Objects can be gang annotated (one annotation is linked to several objects). There is gang collection, to add several objects at the same time to one collection. Writing can be tiring too, so audio annotations are also possible and the Microsoft Speech SDK is used to convert it to text.

The greatest weakness of this system is the fact that copyright keeps an user from keeping absolutely everything that we might consider useful. It is too bad that we can’t store every book we’ve ever read, because the system could ”read” and help the users remember some of the things in it.

### 2.5 Stuff I’ve Seen

Stuff I’ve Seen (SIS)[7] is a system whose identity is also captured by Vannevar Bush’s vision of Memex[3].

This system was created to facilitate information re-use and that was achieved in two ways:

1. The system provides a unified index for everything that the user has seen on the personal computer (e-mail, web pages, documents, videos, music, etc.). This helps to avoid the organization of information, which is really helpful considering that some different types of information require different hierarchy organizations.

2. The fact that it is personal information and it was seen before means that rich contextual cues and properties, like author, time of creation, thumbnails, etc. can be used to search, display and order the information.

The system has five main components:

- **Gatherer** - Specifies the interface to different content sources in their native format

- **Filter** - Takes different file formats, decoding them and outputting a character stream to be processed.

- **Tokenizer** - Breaks the character stream into tokens.
• **Indexer** - Builds a standard inverted index structure.

• **Retriever** - Query language to access stored information.

The interface has two different versions: the Top View and the Side View. The Top View has the query text box and the filters (in the form of checkboxes or comboboxes) on the top of the window. The Side View has these on the side of the window and the filter were simplified, they are viewed serially, like in a hierarchy.

**Strengths and Weaknesses**

A great strength of this system is that, because of the fact that it works from a local index query results can be returned very quickly and a query is launched every time any filtering widgets are manipulated or when the user presses return. This way a user can start a search broadly and keep refining it. Another strength relates to the fact that usually short queries suffice in SIS because results are acquired fast and users can quickly filter and sort them.

A possible weakness could be the fact that rank sorting didn’t have much effect on users, they preferred by date. This could be because date is just more appropriate or maybe the ranking algorithm isn’t as good as it can be. Another possible weakness is related with the fact that not many queries resulted on file openings. This could mean that searches failed or that the results were used in another way.

**2.6 Semex**

Semex[6] was created on top of the idea that a successful personal information management tool presents a logical view of the information based on meaningful objects and associations, this way the system adapts to the user.

To provide this logical view, described by a domain model, an association database was constructed to complement the files data storage with a database that stores the associations and instances. Semex has three sub-modules:
• **Domain Management Module** - Consists of the components of the domain model manager used to learn from the user’s previous browsing experiences to help in domain-model evolution.

• **Data Collection Module** - The components of this module extract data from several sources, reconciles multiple references to one object and stores the results on the association database. Instances are indexed to get fast access.

• **Data Analysis Module** - This module contains the components that offer a user the interface to browse and search information, and that analyze information and activities and triggers notifications when certain events occur.

When information is extracted from its sources the result is references. When those references refer to the same object in the real world, the reconciliation algorithm is used to conciliate them into only one entity. If the reconciliation is well done, the use of associations is stronger.

Semex also supports on-the-fly integration with external data sources. It establishes a relationship between the Semex data model and external data source.

Finally, we have a key to PIM, the personal information search. Semex provides several search modes. The first is the known, usually simple, keyword search. Semex not only finds results containing these words, but also explore associations between instances to find more interesting results. Then the results are returned and ordered according to a ranking that combines relevance, usage and significance scores. The second search mode is finding association chains. If a user wants to find out how the first heard of an author, paper, or something, he can follow an association chain and possibly find it on an email or something. The third and last is finding patterns in external data that consists of when opening a webpage, document, etc. Semex can search for similar instances in the association database.

**Strengths and Weaknesses**

The concern about reconciliation is very important, because without it would be hard to find everything about a single identity when it is referred by several references. An effective reconciliation helps a more effective search in some occasions.

This reconciliation idea has a negative side. If it is not completely effective, a user may think that an object is associated with every reference that exists on the computer when in fact, when searching or browsing, he might be missing some important ones.
2.7 Connections

Connections[16] is a tool to perform search on file systems. The method used combines the traditional content-based with context information obtained from user activity. The system identifies temporal relations between files through the tracing of system calls. These relationships are used to expand the results and reorder them. This method is an improvement not only for reducing false-positives (better recall), but also false-negatives (better precision).

It’s true that content analysis is very useful but there are limitations:

1. A text document is easy to analyze, but the same is not true to multimedia files.
2. Content analysis alone overlooks context, which is very important to a user when organizing files.

So, the work done on this system was specially to improve system searching through context. Like was said before, this tool identifies temporal relations between files tracing the activity on the file system. This system performs search by obtaining a first set of files through conventional content-based search and then extends them with files related.

For the user, Connections is just like any other search tool for the file system. It looks like the ordinary content-based search tools, separated from the file system, that to a given set of keywords returns a set of results ordered by a rank. The beginning of the process is pretty much like the other tools, processing the keywords and getting the same results as a content-based tool. These results are the input to the Relation-Graph that returns a few more documents related to the previous results. The final hits are ranked and presented to the user.

These relationships between documents are identified and stored by two components:

1. **Tracer** - Traces the activities of the file system so Connections can identify relationships between files.
2. **Relation-Graph** - Stores the relationships between files.

Each user has his relation-graph since the same file may have different contextual meanings for different users.

The context-based work of the system has three algorithms. The first one identifies the contextual relationships using file traces and creates the relation-graph. The second algorithm is the one that receives a set of content-based results and finds related files in the relation-graph. The third algorithm ranks the set of results.

**Strengths and Weaknesses**

It’s really important that different users can use the system and have it adapted for each one. The point of using context is for to enhance the search according to a certain user’s usage of the file system and documents in it. If it was not possible to have multiple users, a different user using the systems would be a drawback, since the creation of context would be harder, and the resulting Relation-Graph could hinder the search for any of the users.

One weakness is related to the difficulty in ignoring system calls not made by applications controlled by the user. The tracing can fail to be accurate if the system calls caused by system processes can’t be accurately identified by the Tracer.
2.8 Apple Spotlight

Spotlight\(^1\) is a desktop search application developed by Apple. It is tightly integrated with the file system that is responsible for the preparation for the search by making sure that every time a file is manipulated (created, moved, copied, etc.) it is properly indexed, while working on the background. This is after an initial indexation when the user first logs into the system. Then, Spotlight starts the indexation on every file on the hard disks and on external hard drives, among others. When the system identifies modification on the files of external hard drives, the volume is re-indexed from scratch.

The technologies that Spotlight uses are:

- A database integrated in the file system with a metadata store and content index
- API’s to query the metadata store and content index.
- Plugins to populate the metadata store and content index with the files’ information.
- Plugin API for programmers to provide information to be indexed for custom file formats.

The Spotlight Store is the file system-level database responsible for the storage of metadata and indexing content. The content index uses an evolved version of the Search kit used on Mac OS X 10.3 Panther, but a lot faster. As far as the metadata store goes, it was designed to handle the need of metadata. There is one content index and one metadata store per file system.

A search is performed by creating queries. When creating a query, the search is based on three things: file embedded metadata, file system attributes and file’s text content.

Several file formats importers come usually with the operating system, but if a developer’s application has its own format, a plugin can be created and provided to Spotlight to help it understand the files.

Spotlight can be used and accessed in different ways from a different number of places. On the top-right of the menu bar there’s an icon to open a text field to enter queries. The Finder file manager also has a text field that can be used to query. On both these fields cases results appear just as the user types something.

Strengths and Weaknesses

A great thing about this system is that not only it is available for users to use, but also for the developers. A programmer can develop an application that uses the search technology provided by Spotlight to find data they want to find on the desktop and use it.

Even though it is a good application it doesn’t add much to what already existed. It doesn’t do much more than other commercial applications like Google Desktop.

2.9 Google Desktop

Google Desktop\(^2\) is an application developed by Google that allows desktop search. It also has other features, like a sidebar with several gadgets. Searching with Google Desktop is basically the same as searching with the Google web search engine, with the exception of the indexed documents being on the user’s computer.

After the installation of the application the system starts indexing all the files on the desktop once the computer is idle for more than 30 seconds to avoid disturbances on the computer’s performances. The system will continue to index new objects as they are added, like new documents, e-mails, web pages, etc. The application only indexes up to 100 000 files per drive on the installation. If a drive has more than that they won’t all be indexed, but Google Desktop will index them in real-time as they’re opened or moved. As far as e-mail goes, there are also plugins for the application to allow the indexation of contents of other email databases different that the functions already built-in the application.

Even though a simple keyword search can be performed just like on web search, advanced search is possible. On the advanced search some filters can be applied to the search to get better results according to what the user wants, like type of the files or date.

Google Desktop is installed on the computer to index the files but at the beginning was used on the web browser. To use advance search it still is. Nonetheless there was an interesting feature included on the application called "Quick Find". This feature consists of a simple UI with a text box where we can write something and as we write, instantaneous results show up. To make it appear or disappear we just double touch the ‘Ctrl’ key. This can be used for more than documents

search. It can be also used to launch applications that are installed on the computer.

**Strengths and Weaknesses**

One of the system’s strengths is the fact that it creates some copies of documents on the cache each time we view them and stores them on the hard drive. So if one user accidentally erases a document from the computer, it is still possible to retrieve it without having to do all the work from scratch. Also the way indexation is handled is really good, because it tries to not interfere with the user’s activity.

A thing that may lack is the possibility of advanced search on ”Quick Find”. This forces a user to open a browser if it’s being difficult to find what he is looking for. This simple option would save a lot of trouble for the user.

### 2.10 Beagle++

The Beagle open source project for Linux is a regular desktop search application, like Google Desktop is for Windows. Beagle++[4] is a prototype built on top of Beagle with the addition of a couple of modules:

1. **Metadata Generators** - To manage the creation of contextual information on the documents of the desktop.

2. **Ranking Module** - Responsible for calculating ratings of documents so that the results can be shown in an order of ”most relevant ones first”.

This project was developed having in mind that semantic information from user activity and having knowledge of contexts and links between files would allow a better ranking between files that result from a search.

This extended desktop search system’s most interesting characteristic is the ability to generate metadata and index files on the moment changes are made on the file system. The kernel provides notifications that can be used to trigger actions to respond to a given event. The main components of Beagle++ are Beagled++ (responsible for the main functions, like indexing) and Best++ (responsible for the UI, uses sockets to communicate with Beagled++).

There are different metadata generators to different types and contexts of files and events. The generators are built upon a RDFS ontology that describes RDF metadata for that specific context. The metadata can be extracted directly from properties, like ”subject” or ”sender”, or be created based on association rules and background knowledge. Four contexts were implemented: Files, Web Cache, Emails and Publications (extracts the titles of identified papers and builds the annotations when matching them with entries from the CiteSeer publications database).

Each user has his own context network, so document rankings are different between users. This ranking’s computation depends on the links between files and their metadata.

**Strengths and Weaknesses**

Usually the sorting of results according to a certain rank is not the best approach. People have an easier time remember of a certain document usually for when it was created, or last time it was edited or viewed, which should make date a better sorting category. Nevertheless, if a proper ranking algorithm is developed and ranks documents depending on a user activity and context, the results may be really positive. Google is good because their ranking is good, and most of the time what you want is on the top five results.
Still, something yet to be improved is related to the results ranking. The result’s relevance is not influenced by the extension of the retrieved set, where obviously, from a ranking point of view, similar objects should be weighted less than original ones.

2.11 Phlat

The first system presented is Phlat[5], a result of work done at Microsoft Research. Phlat was created specifically with the purpose of helping users search their personal information on their machines. The storage and indexing systems were already developed as Phlat operates on top of Windows Desktop Search, which indexes desktop information. The challenge was to find a way of taking advantage of the fact that personal information is familiar to the user to make its search easier, so they developed a user interface to accomplish that goal.

As depicted on figure 2.10, the interface is far from simple, with several areas and ways on inputing information. A simple interface would probably be closer to Google’s\(^3\). However, this may help an experienced user perform a search action in a much more effective way.

The biggest area on the interface is the ”Results Area” where the results are displayed and where the sorting is explicit and can be changed. According to the sorting they can be separated into categories. On the left side there are two other areas: the ”Query Area” and the ”Filter Area”. The ”Query Area” has a textbox to write terms to search and the ”Filter Tiles” that represent the filters being applied. Below is the ”Filter Area” that contains a series of expandable areas that have a kind of specifications of these areas to use as filters.

\(^3\)www.google.com
The way the system is used is by searching using all the features presented on the previous paragraph. By writing some keywords on the "Query Area"’s text box and/or selecting some of the various filters, a user begins a search.

When the system began being developed there were some goals to be achieved: extend traditional search. Have more than simple text search. They believed a good search system could reduce a lot the need of organization on information storage and that as a basis for this good search would also be user-generated metadata.

To achieve the goals established, the researchers found that a series of crucial design points should be followed:

1. **Unify text entry and filtering.** Filtering was very encouraged and, even without text, forms queries.

2. **Current search criteria must be visible and salient at all times.**

3. **Provide rapid query iteration.** Dynamic queries provide constant visual updates.

4. **Allow iteration based on recognition.** Users can use results and their properties to redefine queries.

5. **Allow for abstraction across property values.** If a user is looking for a picture he doesn’t have to remember the format. He just has to choose the picture filter.

6. **Tag UI must support both tagging and filtering.**

7. **Integrate with common file system/mail operations.** Users have the possibility to use simple operations like open, copy, etc.

**Strengths and Weaknesses**
A big strength of Phlat is the inclusion of filters. The evaluation that was made to the usage of the system[5] showed that 47% of the queries involved filters and some didn’t even have text entry. Another big strength is the interaction based on recognition feature, because it’s easier to remember something through recognition. The UI is very well achieved in general.

The system also had a few weaknesses. First, because it runs on top of a search engine independent from the mail and file systems changes to an object can sometimes take too much time to be propagated to the index. Second, the interface is far too complex for an unexperienced user. Also it doesn’t map many representations of people into one underlying ”person” so when looking for people it is possible to find different designations for the same person. Another weakness is related with the metadata. When we reply an email, tags don’t travel with it, and a file copied to a non-NTFS system loses its tags too. Finally, the fact that tagging is only possible through Phlat is also a weakness.

2.12 Quill

Quill[11] is a very interesting application with a different way to search for documents.

Quill is a system that basically retrieves a user’s documents by using stories that they write around those documents. Everybody tells stories at some points in their lives. The truth is that storytelling is something that comes naturally in the human species. People may change the way they tell stories, but it is something they do every day.

Stories were analyzed and the system’s creators found an order or frequency of elements most used by users. Quill’s creators defined a set of qualitative guidelines for the design of narrative-based document retrieval interfaces like:

- There is no need to consider different stories when it comes to document age.
- Dialoging with the users prevents them from digressing and help to elicit more information. Most stories have close to the same structure that can be used to guide the storytelling process, regardless of the user.
- When stories are told, much of the information in them depends of the context, and because of that it could be useful to understand the user’s activities and environment.
- Sometimes there are stories about related documents but they shouldn’t be confused with the main story.
- It’s very rare to see users mention stories in the world and occurring around them.

The interface is actually simple. On the left side elements suggested by the system appear for the user to choose them and incrementally create the story. For each element a user can define if it occurred or not, or if he can’t remember.

When searching a document, the entering of a new element to the story creates a new set of inference rules. The documents identified are assigned a score and the sum of those scores result on a documents ranking. The ones with highest rankings are shown.

Strengths and Weaknesses

The stories told using the system are very similar to those told to humans, according to an evaluation made. This is very good because this means the system is very user friendly. It’s like telling a story to your kids or a friend.
A great weakness can be that when describing a document using a story, a distinctive feature may be omitted preventing it to be found among several candidates. The retrieval rate can also not be the best sometimes. Some aspects like incorrect indexing, missing information on the knowledge base, etc. might hinder the retrieval process.

![Quill interface](image)

*Figure 2.11: The Quill interface.*

### 2.13 Jourknow

Jourknow[17] is a system that was created with the objective of making a bridge between two things: simple and fast text input and low-level context automatically captured, and effective information retrieval and presentation.

The basis of this idea is about the value of text. A simple and old recording technology that makes recording information simple itself. Text in its free form allows us to not worry about formats and schemas.

Jourknow supports text input to allow the capture of structured information for later retrieval. It provides several facilities like information entry by typing scraps of text, inclusion of structured information in the text and its extraction, among other things.

The system’s design allowed the achievement of the goal of making the bridge between two things previously referred through:

1. **Unconstrained text entry** - Free text in a small widget invoked by a keystroke.
2. **Entity recognition** - Named identities are spotted and associated with existing identities.
3. **Structure recognition** - Simple subject-oriented grammars are used to parse certain structure information.
4. **Context capture** - User activity is monitorized and recorded.
5. Application Integration - Text can be aligned with structured applications.
6. First-class text fragments - The input is a big collection of text scraps.
7. Structured Retrieval - Search and browsing based on tags, entities and relations.

In Jourknow a user can make notes using an unrestrictive text input. The set of all the notes and their content is called codex. To make the notes' addition easier, hotkeys are provided to interact with Jourknow while other applications have focus. The addition of tags to a note categorizes it and it's done using the '@' symbol.

The system provides two mechanisms to express structured information in a way understandable for it: using a simplified language or pidgin, or a lightweight triple syntax based on Notation3.

There is also the concept of subtext which is an instantiation of a structure described in the codex.

Information describing the circumstances under which a given note was created is called context and its purpose is to help the re-retrieval of that information.

The retrieval mechanism is one of incremental search and a combination of faceted browsing and keyword search all in the same simple interface. Notes can be filtered based on something on their content or subsets of notes that meet a certain criteria can be selected.
Strengths and Weaknesses

The text-based approach is interestingly simple. It’s basic but when used in the right way is really familiar and user-friendly.

A weakness is the difficulty of resolving ambiguous names when recognizing entities. When the system tries to find a match there may exist more than one. The way Jourknow tries to deal with this problem is by trying to find an exact match. If it doesn’t, it provides a list of near misses. If the user doesn’t choose any of the presented choices, the system creates a new subtext item.

2.14 SeeTriever

SeeTriever[12] is a personal document retrieval and classification system that when indexing focuses on what is presented to the user in the user interface. Associating the visible text surrounding a document in time allows the system to gather information about on what task was the document being used. The idea is to try a different approach when compared to the usual content-based systems and when it comes to document recall this context-based system achieved substantial improvements. It’s important because when recalling a previously used document it’s usual for a user to think of the task that occurred when that happened.

Like said before, the application captures and indexes only text displayed at the user interface level to the user. It also uses temporal locality to create a mapping between text snippets and files used during them. This way the index becomes a three-level mapping from terms to snippets and snippets to documents.

Usually there’s a difference between a user interface activity and the complementary activity in the file system. The use of the user interface layer to capture is more attractive due to being much coupled with user activity, involving common activities that aren’t likely to change (like reading) and the contents exposed being those the user interacted with. Five aspects where user interface layer diverge from the file layer were found: location, composition, presentation, interaction and temporability.

Two applications were implemented: Document Retrieval and Context Tagging.

Document retrieval is implemented by SeeTriever through combination of traditional indexation (using Indri or Google Desktop) with the context graph. The index maintenance is achieved by the addition of new snippets as they are seen by the user (if they haven’t been seen before). The query is processed using the index first to find relevant results, and the use of the context graph afterwards allows the discovery of related documents.

Context tagging consists on the creation of a textual summary of a file using the information on snippets related to it. The goal is to use information of content that surround a given file while it is used to identify relevant terms for it.

Strengths and Weaknesses

The main strength of this system is probably the way this idea tries to focus on only relevant information based on the principle of ”if the user didn’t bother to see, it’s probably not important”. This way less information is indexed and searches become more efficient while saving hard drive space.

Although this new approach is refreshing and addresses problems like the existence of too much information, it can fail if we are trying to find information we know we have but we never saw it on that computer, but have read it somewhere else and transfered it recently.
2.15 DEVONthink

DEVONthink\textsuperscript{4} is a commercial information assistant designed to help the user keep disparate pieces of information in order and help him manage them.

This system stores one’s documents, scanned papers, etc. in one place and allows a user to access live web pages from within it to review and/or extract more information. With this system, a user can annotate PDF documents using standard annotations, cross-reference documents and edit images. With a built-in interactive web interface, knowledge sharing is simpler.

With the help of built-in Artificial Intelligence document management gets easier:

- Documents can be classified and filed automatically
- Through finding relations between documents it can present similar ones
- When text is selected similar documents can be found
- The most relevant topics can be automatically extracted
- The search function is powerful: It finds related words, similar spelled words, broadens or narrows search terms and has advanced boolean operators.
- Summaries are created based on the user’s data
- Data navigation is easier with the powerful concordance
- Documents can be automatically grouped by content.


![Figure 2.13: DEVONthink search interface.](image-url)
DEVONthink also provides integration with other systems. Integration with certain web browsers allows sending web pages and search results to a certain database. With certain scanning applications papers can be scanned and automatically filed as searchable PDF’s. With a few email apps a user’s email can be directly archived. Lots of file formats can be stored and viewed and the databases can be searched using Spotlight.

**Strengths and Weaknesses**

This system has a number of positive features, like for instance the way it can be integrated or integrate other applications to make the user’s life easier. Also the search having so much automated features thanks to the artificial intelligence is a strength.

Still it has a few weaknesses like using the system outside of Mac computers is possible but harder. As a commercial application it is hard to know more about it since information about how it works is not available.

### 2.16 YouPivot

YouPivot[13] is a personal information retrieval tool that tries to improve the retrieval process using contextual search.

![YouPivot screenshot](image_url)

*Figure 2.14: YouPivot screenshot showing an example of sympathetic highlighting.*
This tool presents a new interaction technique called pivoting that allows users to search for a given item using information not necessarily semantically related but contextually related to it. For example, this technique allows a user to find a PDF document using information of a music he or she was listening to while working on it. This example uses listening to music as a related activity, but it can be other things like the user’s physical location, interrupting phone calls, etc. These temporally related activities are called contextual cues by the tool’s creators. Studies show that we, humans, use contextual cues as a key component of the human memory, as we think of context when remembering things we might have forgotten. So, this approach may help users retrieve items the same way they remember events. At the heart of this system is the ability to allow a user to shift the point of view while using an operative system, making it possible to check related activities and evaluate the strength of their relation.

This tool allows a number of tasks to be performed:

- **Search and Pivoting** - Users may search for an item contextually related with what they want to find (like a song) and then, using pivoting, get files related in a given pivot period.

- **Temporal Context** - Users may browse related activities that happened not only at the same time, but also a short while before or after the one chosen as main activity.

- **Visualization of Activity** - Not only activity relations can be seen, but also detail information about a given activity.

- **Ambient Context Display and Filtering** - Colors and icons allows a user to see what activities dominated in terms of time usage and then information can be filtered using, for example, keywords presented that are related to activities.

- **Creation of TimeMarks** - Bookmarking 20 websites seen in a period of time while investigating some information can be arduous, so YouPivot allows time marking which marks a given time period and all activities in it.

### Strengths and Weaknesses

A big YouPivot strength, besides the idea itself, is the fact that it is based in the cloud. This means that as long as the user has an Internet browsing device and Internet connection, the application can be used.

However, the fact that it registers the activities on the device is not complete. A user may be working on something while listening to a song that comes from a music video on TV and when searching it later will not find it, because where the music came from is something the user does not remember.

### 2.17 Discussion

This section will present a brief analysis of the related work mentioned on the previous section. Some systems are very different, but there are also some very similar. In this section, with the help of Tables 2.1 and 2.2 the differences and similarities will be discussed. Two tables are used because one was not enough to fit all the information.

First, we should explain each of the columns of these tables. On Table 2.1 the first column is named *Commercial*. It allows to indicate if a given system is built for commercial use or for research. The next column is titled *Indexation* and exists because some systems only allow search, but others are also responsible for the indexation. The third column of Table 2.1 allows to indicate what sources do the systems index and/or search for. The *Annotations* column’s purpose is to
fill according to a system’s ability to annotate files with tags or descriptions, allowing the user to generate metadata to enhance the searches’ efficiency. The *Multiple Views* column indicates if information can be viewed in multiple ways. Finally, the last column of Table 2.1 has the purpose of showing what kind of search modes the system supplies its users.

Looking now at table 2.2 it is possible to verify that the first column, the one called *Organization*, exists to define how the system organizes the files, if it does at all. The *Context* column states whether or not the system uses context to enhance search. The next column is important because some systems that use context to enhance search are based on user activity, so some systems might provide ways of having different contexts depending on the user. The *Entities* column is checked if there are entities on the system. Certain systems store entities information they extract, like involving people in a query might require having the person’s entity represented. The last column is referring to when a system extracts information from metadata and creates entities, if they do reconcile different references to the same entity.

Of the sixteen analyzed applications, only 3 were commercial and didn’t have any scientific papers written about them. The problem with this fact is basically the lack of some information about the systems, especially how they really work inside.

Almost every system is responsible for the information indexation. The exception is Phlat that works on top of Windows Desktop Search. It’s true that it can be considered that all of the others index the information but they don’t do it all the same way. Some, like Haystack, Quill and Beagle++ decided to use RDF and RDF Schemas to create Knowledge Bases or Semantic Networks. Others, like MyLifeBits and Semex create associations between different entities and store them on databases along with index files. These approaches are less efficient than a traditional one using inverted indexes. Nonetheless, using these ideas is probably the right track to follow in order to improve the quality of the results provided by searching features.

If we look at how many formats do a given system support, almost every system support the general files on a PC even if only indexing the text in it. Apple Spotlight provides an API to allow the indexation of custom file formats belonging to applications developed by someone for their own work. Beagle++ allows search on Citeseer for scientific publications. Some applications like Phlat, Stuff I’ve Seen and Google Desktop index email and web cache too. YouPivot indexes every user activity, local or web. Another application different than the rest is Jourknow that basically the information it searches for is the one previously entered by the user, since it is an application that allows text information to be entered and structured like text annotations.

Is interesting that only a few of these systems allow annotations, when it is such a simple way for users to enter extra information that may enhance search. Besides Jourknow only four applications support this in some way. Timescape allows the creation of post-its associated with items to work as annotations. Phlat allows tagging which is a type of annotations and also enhance the search process. MyLifeBits belief when it comes to annotations is that for some types of information (like multimedia) it is crucial. The Beagle++ has RDF annotations that it generates to have more metadata to work with.

Out of the analyzed work five of them had more than one way of viewing information. MyLifeBits had four views implemented to visualize information (detail, thumbnail, timeline, and clustered-time), Stuff I’ve Seen had two simple search views (Top View and Side View), Haystack was built with the idea of a personalized way of viewing for each person and Timescape, besides its desktop view based on time, provides a timeline view of information. Most of the applications (like Phlat) allow changing the property by which the results are ordered. This is not a set of different views but it changes the visualization and possibly presents the results in a way that would make the user more satisfied. YouPivot presents the classical list of results as well as sympathetic highlighting.
Table 2.1: First table with the summary of the features and characteristics of the systems presented on the previous section.

<table>
<thead>
<tr>
<th>Application</th>
<th>Commercial</th>
<th>Indexation</th>
<th>Sources</th>
<th>Annotations</th>
<th>Multiple Views</th>
<th>Search Modes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Phlat</td>
<td>x</td>
<td>x</td>
<td>Documents, Multimedia, Email</td>
<td></td>
<td>x</td>
<td>Keyword and/or Filtering</td>
</tr>
<tr>
<td>MyLifeBits</td>
<td>x</td>
<td></td>
<td>Everything</td>
<td></td>
<td>x</td>
<td>Keyword</td>
</tr>
<tr>
<td>Stuff I’ve Seen</td>
<td>x</td>
<td></td>
<td>Everything (Email, Files, Web Pages)</td>
<td>x</td>
<td></td>
<td>Keyword, Filtering (Fuzzy is possible)</td>
</tr>
<tr>
<td>Haystack</td>
<td>x</td>
<td></td>
<td>Everything</td>
<td></td>
<td></td>
<td>Keyword (Fuzzy is possible)</td>
</tr>
<tr>
<td>Lifestreams</td>
<td>x</td>
<td></td>
<td>Everything</td>
<td>x</td>
<td>x</td>
<td>Text query</td>
</tr>
<tr>
<td>Quill</td>
<td>x</td>
<td></td>
<td>Everything</td>
<td>x</td>
<td>x</td>
<td>Story Telling</td>
</tr>
<tr>
<td>SeeTriever</td>
<td>x</td>
<td></td>
<td>Text Snippets from Documents</td>
<td>x</td>
<td>x</td>
<td>Keyword</td>
</tr>
<tr>
<td>Connections</td>
<td>x</td>
<td></td>
<td>Everything</td>
<td></td>
<td></td>
<td>Keyword</td>
</tr>
<tr>
<td>Beagle++</td>
<td>x</td>
<td></td>
<td>Everything including Web Cache and Email. Publications at Citeseer</td>
<td></td>
<td>x</td>
<td>Keyword</td>
</tr>
<tr>
<td>Jourknow</td>
<td>x</td>
<td></td>
<td>Information previously entered</td>
<td></td>
<td></td>
<td>Keyword</td>
</tr>
<tr>
<td>Google Desktop</td>
<td></td>
<td></td>
<td>Everything including Email and Web</td>
<td>x</td>
<td>x</td>
<td>Keyword and Filtering</td>
</tr>
<tr>
<td>Apple Spotlight</td>
<td></td>
<td></td>
<td>Everything including Email and Personalized Formats</td>
<td></td>
<td>x</td>
<td>Keyword</td>
</tr>
<tr>
<td>DEVONthink</td>
<td></td>
<td></td>
<td>Everything</td>
<td></td>
<td></td>
<td>Keyword and/or Filtering (Fuzzy is possible)</td>
</tr>
<tr>
<td>Semex</td>
<td>x</td>
<td></td>
<td>Everything including Email</td>
<td>x</td>
<td>x</td>
<td>Keyword, Association Search, Search for Similar</td>
</tr>
<tr>
<td>TimeScape</td>
<td>x</td>
<td></td>
<td>Everything</td>
<td></td>
<td></td>
<td>Looking for terms through time</td>
</tr>
<tr>
<td>YouPivot</td>
<td>x</td>
<td></td>
<td>User activity (Web and Local)</td>
<td>x</td>
<td></td>
<td>Keywords, Pivoting</td>
</tr>
</tbody>
</table>
Table 2.2: Second table with the summary of the features and characteristics of the systems presented on the previous section.

<table>
<thead>
<tr>
<th>Application</th>
<th>Organization</th>
<th>User Context</th>
<th>Multiple Users</th>
<th>Entities</th>
<th>Reference Reconciliation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Phlat</td>
<td>OS responsibility</td>
<td>✗</td>
<td>✗</td>
<td>✓</td>
<td>✗</td>
</tr>
<tr>
<td>MyLifeBits</td>
<td>Collections</td>
<td>✓</td>
<td>✗</td>
<td>✓</td>
<td>✗</td>
</tr>
<tr>
<td>Stuff I’ve Seen</td>
<td>OS responsibility</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
</tr>
<tr>
<td>Haystack</td>
<td>OS responsibility</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
</tr>
<tr>
<td>Lifestreams</td>
<td>Stream of files</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
</tr>
<tr>
<td>Quill</td>
<td>OS responsibility</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
</tr>
<tr>
<td>SeeTrieve</td>
<td>OS responsibility</td>
<td>✓</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
</tr>
<tr>
<td>Connections</td>
<td>OS responsibility</td>
<td>✓</td>
<td>✓</td>
<td>✗</td>
<td>✗</td>
</tr>
<tr>
<td>Beagle++</td>
<td>OS responsibility</td>
<td>✓</td>
<td>✓</td>
<td>✗</td>
<td>✗</td>
</tr>
<tr>
<td>Jourknow</td>
<td>OS responsibility</td>
<td>✓</td>
<td>✗</td>
<td>✓</td>
<td>✗</td>
</tr>
<tr>
<td>Google Desktop</td>
<td>OS responsibility</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
</tr>
<tr>
<td>Apple Spotlight</td>
<td>OS responsibility</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
</tr>
<tr>
<td>DEVONthink</td>
<td>System is responsible for storing</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
</tr>
<tr>
<td>Semex</td>
<td>OS responsibility</td>
<td>✓</td>
<td>✗</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>TimeScape</td>
<td>PC is a time machine</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
</tr>
<tr>
<td>YouPivot</td>
<td>OS responsability</td>
<td>✓</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
</tr>
</tbody>
</table>
Almost all of the fifteen systems discussed do keyword search even though only a few do nothing more than that as far as search and retrieval go. The great exception is Quill that retrieves a document based on a story told about it. Even though a few systems only do keyword search (MyLifeBits, Haystack, SeeTriever, Connections, Beagle++, Jourknow, Apple Spotlight and TimeScape) there are also some that have the possibility or even encourage filtering, like Phlat, Google Desktop and DEVONthink. Of the applications that only allow keyword search some go a little further than the simple and traditional one. Haystack can return some fuzzy results while TimeScape, given some keywords can keep advancing or going back on time as commanded by the user. DEVONthink and Stuff I’ve Seen also allow Fuzzy like Haystack. Semex not only allows keyword, but it also allows association search by going through association chains, like getting the last email of a great sequence of them and finding the person who sent the first just by following the association chain, and to search for similar to find instances similar to a given one. Lifestreams accept a sentence to create a substream with some characteristics and does it. YouPivot uses pivoting, which means that contextual relation between activities are used to find information.

Most of these applications don’t change the way a user organizes documents, leaving this to how the operative system does. They still, most of the time, organize the extra information they may generate or require in a way decided by the system architects and designers, for example, Semex stores the associations between entities on a database. On the other hand, some systems change the way the organization of the files is made. Sometimes it is needed (MyLifeBits), other times is the purpose of the application (TimeScape, Lifestreams). MyLifeBits uses the idea of collection, not enforcing hierarchy like usually the OS’ do. TimeScape turns the PC into a time-machine having only the items of the desktop that a user can delete and ”travel back in time” to find them. Lifestreams doesn’t require organization. Files are chronologically ordered in a stream of files.

Some of these systems (MyLifeBits, SeeTriever, Connections, Beagle++, Jourknow and Semex) tried to enhance the information retrieval by using context. Even if Stuff I’ve Seen is not one of these systems, the creators seem to think that using file properties is, in a way, using context, because people have seen the information before, therefore they should know some of the properties and it may help with the search. MyLifeBits has done it using context through the collections organization and links between items that create stories. SeeTriever only indexes information actually seen by the user by monitoring the user activity. Connections creates associations between files based on user activity by tracing system calls. Beagle++ also uses relations between files but uses this information to compute a better rank of files when searching for items, like Semex, while Jourknow captures automatically low-level context. YouPivot uses the user’s point of view in the system to create temporal relations between activities in order to find information even when not semantically related.

Of these applications that use user context and activity to enhance retrieval, only two of them (Connections and Beagle++) considered the idea of different users using the same machine. Connections has a relation-graph for each user of the system and Beagle++ creates a different contextual network for each user.

Of all of the applications presented, four (Phlat, MyLifeBits, Jourknow and Semex) were considered to have entities represented that not only represent documents and files. They may also represent other things, like people. These entities can many times be represented by different names or attributes. When it comes to people, someone named John Smith can be represented not only by this name, but also by J. Smith, John S., john.smith@domain.com or even a phone number. When a system extracts information from different places and finds different references for the same identity it should be able to see that and reconcile these references. Of the four applications mentioned to represent information in entities only Semex is designed to allow this reference reconciliation.

As was possible to verify throughout this chapter, the fifteen systems described in the previous
one were very similar in some points and very different in others. All of them have a little something
that in some way is related or similar to what Social Scribe is supposed to become. Nonetheless,
Social Scribe idea is to bring a major feature that none of these systems have. The information
these systems index is from several sources of information, but one of the biggest and trendiest
sources these days is social networks and cloud applications. Social Scribe presents a solution to
the indexing of such information using the Scribe tool, allowing for an easier and more effective
personal information retrieval.
Chapter 3

Architecture

This chapter is dedicated to the description of the underlying architecture of the solution developed which is basically the architecture of the Scribe tool, the original indexing system used as a starting point for the development of Social Scribe. This is true because the development of Social Scribe was done without any relevant changes on the underlying architecture. Even though (a somehow big) part of the work done while developing the solution consisted on changing and tweaking Scribe, this was achieved keeping the structure of the architecture intact.

3.1 Scribe

As said before, Scribe is a solution developed to index personal information. In this section its architecture will be discussed.

First, we will talk a little bit about the history of the tool in order to understand some things about the state of its source code when the development stage of Social Scribe started.

Scribe was initially developed in INESC-ID by several researchers from that institution between the years of 2005 and 2006 and was known as Scroll at the time. Its purpose was to serve as the indexer of Quill[11], a solution already described on Chapter 2, more specifically on Section 2.12. Since then, many students working on their projects and dissertations that needed indexing for their solutions have used Scribe. Most of the people that had the need for Scribe used it adding something to it or changing something in it, to have it fill the need in question. Needless to say that as time passed by, the impact on the quality of the source code caused its decrease and some bugs, born out of adding or changing features without testing every use case, emerged.

3.1.1 Scribe’s Architecture

The system’s architecture is simpler to understand using visual aid, so it is depicted in Figure 3.1. The most important components are represented in Figure 3.1 and, as we can observe, are:

- Scribe
- Plugins
- Nibs
- Nobs
- Accesspoint
- Parchment
In Figure 3.2 we can see Scribe’s architecture represented as a layered architecture. Even though the information in it can be taken from Figure 3.1, this picture makes it easier to understand how the communication between different modules is organized, being done mostly by layers. As we can see, the only communication done between two non consecutive layers is the one between Nibs, which create an abstraction layer above the plugins that implement functions to be used by the several plugins of the same source (e.g. email), and AccessPoint. To perform simple actions that sometime require access to the Database, there is no need to implement a new Nob function (Nobs create an abstraction layer between the indexer and the database and implement functions to access and manipulate it’s information). So it may be simpler to access through AccessPoint.

Other than these modules represented in Figure 3.1 and Figure 3.2, there is another component very important for the execution of the indexing tool: the configuration file. This component didn’t need to be depicted in there two images because it contains information needed for the execution of almost all the other components.

On the following subsections each of these components will be presented and discussed in a deeper way.
3.1.2 Scribe

This module is the main one, responsible for launching the indexing process. Figure 3.3 depicts the flow of the process when launched.

Before explaining this module more thoroughly through its execution flow, we should understand the configuration file.

Configuration File

This component is used in many of the others, so it makes sense that it is discussed in the beginning of Scribe’s presentation.

The configuration file as a module is more than a file with properties and values associated to them. To make the usage of the file easier, two functions were developed that made possible to manipulate it using an existing library. These actions are simple, but necessary and include loading the configuration file and saving it according to the changes made by the application. Using external libraries other two actions can be made: reading or writing on a configuration file property.

Scribe’s Flow of Execution

When the process is launched, the first step is to load the configuration file. This is done because the information in the information in the configuration file will be needed for two different parts of the process. The information needed is in two different properties and contain what is the directory the index will be placed in and what are the plugins that are supposed to be launched.

After the information is obtained, the first thing to do is verify the existence of the directory where the index is supposed to be placed. If the directory exists, the plugins are launched each on its own thread, which means every plugin runs at the same time. If the directory does not exist, before starting the plugins supposed to run, the system creates it. To launch the plugins, the information from the configuration file is used. There is a property on the file referring to what plugins to load. This property is a string (like all the other configuration properties) where the plugins to load are written and separated by commas. This string is parsed and the plugins are loaded.
Figure 3.3: Flowchart of Scribe module execution.
After launching the plugins the program waits for a signal to close the program, usually using the key combination of 'Ctrl+C'.

### 3.1.3 Plugins

Now we will discuss the plugins package existent in Scribe. Each plugin is translated into programming code to extract information from a source in a given way, e.g. there are several plugins for email, but each one does it differently (IMAP, POP, etc.). When information is extracted, it is passed to the next layer in the indexing process, the nibs, which will be presenter later on this document.

Every plugin is a subclass of the base plugin and must implement its functions that are necessary to allow each one to be launched in an uniform way.

The plugins implemented in Scribe are:

- One Time File Indexer
- IMAP 4
- POP 3
- SMTP
- Windows Mail
- Outlook
- Thunderbird

As we can see, the plugins implemented allowed only to index two types of information:

- Email
- Hard Drive

In the next couple of subsections, these two types of information will be approached and each plugin will be presented more thoroughly.

### Hard Drive

Of the seven original plugins, only one was dedicated to the process of indexing hard drive information, and it was enough, even though not as complete as an hard drive indexer, as it could be.

Obviously we’re talking about the "One Time File Indexer" plugin. Adequately named, this plugin performed the indexing process on the hard drive only once, when the user ran it for the first time.

On Figure 3.4 a flowchart of this plugin’s execution is depicted.

Even though the Scribe’s One Time File Indexer plugin the hard drive is indexed exactly once, it is protected with a mechanism that allows the plugin to restart from a given crashing point by storing its progress in a file. So the first step of this execution is to verify whether a progress file already exists. If it does not, then the execution is beginning for the first time and so it must be crated and every directory is to be indexed, so they are listed. If such a file exists, then the directories that are yet to be indexed are listed.
Figure 3.4: Flowchart of the One Time File Indexer plugin execution.
After verifying the existence of a progress file, the process gets into a loop where it is verified if there are still directories to be indexed. If not, the process is finished. Otherwise, an unindexed directory is picked up and its files get indexed. This happen by having the process getting into another loop where it is verified if in the current directory there are still files to analyze. If not, it moves on to the next directory to index. If there are still files to index, a file is picked and analyzed to verify if it is indexable. This is done, mostly by verify if the file has an extension for which the program has a way to extract its information and index it. If the file is not considered to be indexable, we move on to the next file to index. Otherwise, we continue its indexing process and verify if the file has keywords to be extracted. For, example, an audio file or an image may be in an indexable format but they may or not have keywords to be extracted from the content, depending on the existing metadata. If there are no words to be extracted, the file gets indexed with the obtained information. Otherwise, before having the file indexed, the plugin get the keyword frequency and has the file indexed using this information too.

The responsibility regarding indexing the file is passed to the filenib, which we will approach later on this paper on Section 3.1.4 that presents Nibs.

Email

As we have previously observed, there were six plugins implemented to extract information from email, being different according to the protocol used, the desktop application used, etc. To remind us of what were the six plugins, we will list them again:

- IMAP 4
- POP 3
- SMTP
- Windows Mail
- Outlook
- Thunderbird

The implementation of more than one way to extract information from email is good, because it makes email indexation more effective, considering that in different conditions it can be done in a different way. For example, IMAP 4 would possibly be the most common way to extract a user’s emails from an email account. However there are a few problems. If the Internet connection breaks for some reason, during the indexing process, it can be a problem, specially concerning the indexing time. Also, the fact that every email message and attachment is downloaded, it can take a very long time, just to obtain the information. So, in a general way, IMAP is a good way, because email service providers usually provide an IMAP server, but if an alternative is possible, it can be better. If a user usually works with a desktop email client like Microsoft Outlook or Mozilla Thunderbird, it becomes faster to extract the information, since there is no longer the need to download it, and also there is no need for Internet connection during the indexing process, as long as the information is updated on the email client.

We will now present each of these plugins to try to help the reader understand them and what they do.

**IMAP 4**
Figure 3.5: Flowchart of the IMAP 4 plugin execution.
IMAP (Internet Message Access Protocol) is an Internet standard protocol for email retrieval and one of the two most used for this purpose, the other being POP (Post Office Protocol). In other words, IMAP is mainly used to transfer email messages from an email server that has to, obviously, support it.

In this subsection we will present the plugin that relies on this protocol to extract information from an email account to index it.

On Figure 3.5 is possible to observe the flow of this plugin’s execution, like it was done in the previously presented plugin, with the aid of a flowchart.

If we look to Figure 3.5, we can see that the first step in this plugin’s execution is to connect to the IMAP server of the email account the user intends to index. This is done by, using login credentials to an email account and an address to the IMAP server of the email server, logging in the user in a given IMAP server with a given account. If the login credentials are wrong or do not match, the user is notified of that fact. Also, if a login problem occurs, for example due to Internet connection problems, the plugin waits a connection retry interval and retries to connect to the given account on the given server.

After connecting to the server using the users account, the program uses the folders specified in the configuration file to know what folders to index. Those folders are listed and the program enters a loop. When the number of unindexed folders becomes zero, the program disconnects from the server and waits a given interval of time to restart all over again to update the information on the index regarding email. However, while there are still unindexed folders, the plugin chooses one and lists the messages in them. Then, the plugin enters a new loop. While there are still messages to download messages the indexing keeps going. If the message has not yet been indexed, an Event is written in the database followed by the actual email indexing done by the mailnib. The Event is what says that a given email has already been indexed. Each email indexing process results in a new event, and when trying to create one that already exists, it is verified that the email was previously indexed. This is possible due to emails having an unique ID.

**POP 3**

POP (Post Office Protocol) is an Internet standard protocol for email retrieval and one of the two most used for this purpose, the other being IMAP (Internet Message Access Protocol). In other words, POP is mainly used to transfer email messages from an email server that has to, obviously, support it.

This subsection will be used to present the plugin that relies on this protocol to extract information from an user’s email account to index it.

Using POP consists in creating a communication channel between two processes and trading messages. Usually is something like an email client communicating with a POP server provided by the email service. For example, the following text represents a communication between a client and a POP Server:

```
S: <wait for connection on TCP port 110>
C: <open connection>
S: +OK POP3 server ready <1896.697170952@dbc.mtview.ca.us>
C: APOP mrose c4c9334bac560ecc979e58001b3e22fb
S: +OK mrose's maildrop has 2 messages (320 octets)
C: STAT
S: +OK 2 320
C: LIST
S: +OK 2 messages (320 octets)
```
Figure 3.6: Flowchart of the POP 3 plugin execution.
So, to obtain emails via POP we just have to emulate this kind of communication in order to get the email messages in the user’s account by making requests and reading the answers.

We can observe in Figure 3.6 the depiction of the plugins execution in the form of a flowchart. By looking at this image we can see that this is a very simple process. First, the local POP Server is launched. This serve will act as the client of the email server, communicating with it in a similar way of the example previously presented. Next, the channel between the two processes, local POP server and email server, is created, making communication between them possible. So, the first thing before starting to transfer messages is to authenticate the user and after that, the process of downloading messages begins. The server is constantly checking for messages, and downloading them to index. After having an unindexed message, the indexing process is the same as in IMAP. If it has not been indexed before, write the Email Event and index the email using the mailnib.
**SMTP**

SMTP (Simple Mail Transfer Protocol) is an Internet standard for email transmission across Internet Protocol (IP) networks. Simple Mail Transfer Protocol is the protocol used when people communicate via email, by establishing how messages are sent from the users email server to the other. SMTP is a text-based protocol, in which an email sender communicates with an email receiver by issuing command strings and supplying the necessary data for the communication over a reliable ordered data stream channel.

![Flowchart of the SMTP plugin execution.](#)

The problem about SMTP is that it is a delivery protocol only and it cannot pull messages from a remote server on demand. It was not designed for that purpose like IMAP or POP, which are designed for retrieving messages and managing mailboxes. However, SMTP has a feature to initiate
mail queue processing on a remote server, being the requesting system, then, capable of receiving any messages destined for it.

Like it was previously said, SMTP is a text protocol, and like present in POP, the communication in servers is done through the exchange of text messages between the local client and the remote server. The next few lines show a SMTP transport example:

```
S: 220 smtp.example.com ESMTP Postfix
C: HELO relay.example.org
S: 250 Hello relay.example.org, I am glad to meet you
C: MAIL FROM:<bob@example.org>
S: 250 Ok
C: RCPT TO:<alice@example.com>
S: 250 Ok
C: RCPT TO:<theboss@example.com>
S: 250 Ok
C: DATA
S: 354 End data with <CR><LF>.<CR><LF>
C: From: "Bob Example" <bob@example.org>
C: To: "Alice Example" <alice@example.com>
C: Cc: theboss@example.com
C: Date: Tue, 15 Jan 2008 16:02:43 -0500
C: Subject: Test message
C:
C: Hello Alice.
C: This is a test message with 5 header fields and 4 lines in the message body.
C: Your friend,
C: Bob
C: .
S: 250 Ok: queued as 12345
C: QUIT
S: 221 Bye
{The server closes the connection}
```

To initiate the previously mentioned mail queue processing on a remote server, two commands can be used: TURN or ETRN. Usually TURN is disabled due to the risk of dequeuing mail to an unauthorized host, so ETRN is the one usually used, even though is not available in all SMTP implementations. Here's an example of this command’s use:

```
C: EHLO mindflip.com
S: 250-arjuna.mindflip.com Hello nakula [206.50.17.90], pleased to meet you
C: ETRN mindflip.com
S: 250 Queuing for node mindflip.com started
```

Figure 3.7 depicts the execution of Scribe’s SMTP plugin. Is very simple, considering Python has a lib that implements a SMTP server. First we launch the proxy server to communicate with remote server and demand the emails and then we connect it with the remote server. Then, while
the server gets email messages, it keeps indexing it in the same way as the previous email plug-
ins. If the message has not been indexed before, the event is written to avoid indexing the same
email twice and then the email is indexed by the Nib responsible for it, which would be the mailnib.

**Windows Mail**

Windows Mail is an email and newsgroup client that was included in Windows Vista. Windows
Mail replaced Outlook Express and was later replaced by Windows Live Mail.

Figure 3.8 depicts the execution of the Windows Mail plugin. If we pay attention we’ll see that
it is very simple and similar to some other email plugins, specially IMAP of those we’ve seen and
approached so far.

Considering Windows Mail is an email client, all email information is in the hard drive. So, using
an external library, the plugin accesses the Windows Live information and extracts the information.
The first step is to list every folder on the email box. Then the program gets into a loop. When
every folder is indexed, the indexing process is considered finished. But while there are still folders
to index, the plugin picks one up and lists the messages in it. While there are still messages to
analyze, the plugins gets one of them. Otherwise, the program moves to the next folder. When a
message gets picked up, the first thing done is to verify if the email has already been indexed. If
so, the plugin moves on to the next message. However, if the email has not been indexed yet, the
event of its indexation is written in the database, followed by the email indexation by the mailnib.

**Outlook**

Microsoft Outlook is a personal information manager. Even though it is mostly used as an email
client, it includes other features such as calendar, task manager, contact manager, note taking, etc.

Just as it has been done in previous section, a flowchart depicts the execution of this plugin in
Figure 3.9.

Like it was explained previously in Windows Mail, as an email client, the information the Outlook
plugin tries to extract is on the hard drive. So also here, it is accessed fairly easily using a lib for
that. The execution is simple to understand. The first thing to do is create an Outlook session
to be able to get the information. After that, The base folder is listed and the program enters a
loop. While there are items in the base folder, for each one the proper handler is selected to have
it indexed. There are three handlers for three types of items:

- **Email** - Email message stored in Outlook’s email client application.
- **Contact** - Email address stored in Outlook’s contact manager.
- **Appointment** - Calendar entry in Outlook.

These three types of items are extracted from the user’s information in Microsoft Outlook and
properly indexed. When every item in the base folder is indexed, the next step is to verify if there
are any child folders and repeat all the process to index them: index every item and every child
folder in them. So all this is done recursively.

**Thunderbird**

Like Microsoft Outlook and Windows Mail, Mozilla Thunderbird is a desktop email client, which
means that the information to be indexed is not in the cloud.
Figure 3.8: Flowchart of the Windows Mail plugin execution.
Figure 3.9: Flowchart of the Outlook plugin execution.
Figure 3.10: Flowchart of the Thunderbird plugin execution.
Figure 3.10 depicts the execution of Scribe's Thunderbird plugin. This plugin, when developed, like the One Time File Indexer, was developed for a One Time email indexing process, and so was built with a way to keep progress stored if the plugin, for some reason, does not finish.

Having said that, the first thing done during the plugins execution is to verify if a progress file exists. If it does not, it is created and the plugin gets all the paths to Thunderbird mailboxes. If it does exist, the plugin gets the mailboxes paths that are still to finish indexing. After knowing what mailboxes are to index, the process enters a loop.

In this loop the process checks if there are still mailboxes to index. When there are none, the indexing process is considered finished. Otherwise, a mailbox is chosen and another loop is entered, but this time it is to index every message in the mailbox. While there are messages to analyze in the mailbox, one is chosen and it is verified if it has already been previously indexed. In case the test results in a positive outcome, the plugin moves on to the next message. If the result is negative, the email is indexed like in previous plugins: write email event and give the information to the mailnib in order to have the email indexed.

3.1.4 Nibs

Nibs, in Scribe, basically consists of the layer below plugins. After plugins to their job, that mostly consists of extracting information, that information is passed to nibs so that they proceed with the indexing process. Scribe had two nibs that we will discuss in a more profound way. They are:

- **Filenib**
- **Mailnib**

The purpose of nibs is to treat information from the same types of source in an uniform way. For example, as we have seen in 3.1.3, there are more than one plugin to index email, six to be exact, considering different protocols to obtain the email messages and different email client to get the messages from. Even though these six plugins existed, in the end the information obtained was the same in all of them. So, it didn’t make sense to have the same code multiplied six times, and with nibs, it doesn’t have to. Nibs provide API’s to index information of the same type in the same way.

On the following subsections, each of the two nibs previously mentioned will be approached.

**Filenib**

Filenib, as its names suggests, is the nib responsible for the indexing of files in a computers file system. This nib has only two functions implemented:

- **index_file**
- **locus**

The **index_file** function was the function responsible for getting the information extracted from files using the One Time File Indexer plugin.

Figure 3.11 depicts the execution of this function, and as we can see is fairly simple. First it gets the information passed by the plugin. Then, using the hash key created for each indexed file, it is verified if the file already exists in the database. If it does, the file path is added to the existing entry if it is different.

This approach was to have less entries in the database, because, since it is a One Time File Indexer, is not like one of the file’s information is going to be updated and its entry will have to
be modified. Nonetheless, this method is flawed. On the database a file has a property *path* and a property *name*, that combined become the file system’s filename. Adding only the path is not enough, since with the same hash key two files may have the same content, but not necessarily the same name, so the retrieval of one of the files may be impossible. More than that, even if a second name is added, it would be impossible to make the right match between path and name with several paths and names. Conclusion, the right option here, even if less efficient concerning space, would be creating a new entry.

Moving on, if the file does not exist in the database, it indexing process continues. First, the keyword frequency is updated with increased frequency for metadata keywords. Then all the properties are prepared for indexing:

- **name** - The name of the file within its directory.
- **extension** - The file’s extension.
- **path** - The path to the file excluding its name.
- **locus** - The ID of the computer where the file is stored.
• **size** - The size of the file.
• **created** - When was the file created.
• **modified** - When was the file last modified.
• **accessed** - When was the file last accessed.

Finally, the file’s entry is written on the database.

The *locus* function is a very simple one. It gives the computer where a file is stored an id, or if it already exists, it gets it.

**Mailnib**

Mailnib is the nib responsible for email information. It implements seven functions to support email indexing, like verifying if an email has already been indexed, index an attachment and, of course, index an email item.

Of these seven functions mentioned to exist, there are three that are complex enough to be justified having them more thoroughly examined and explained:

• **index_email_document**
• **index_attachment**
• **index_email**

The *index_email_document* function is the function responsible for indexing the message body of an email message. Figure 3.12 depicts the execution of the function in a flowchart.

As we can see, the algorithm is very simple. The function that calls *index_email_document* passes it the message information. Like it was a file, the next step is to calculate the keyword frequency. After that all the properties are prepared and the message body is indexed.

The *index_attachment* function is the function responsible for indexing email attachments. Figure 3.13 depicts the execution of the function in a flowchart.

The function that calls *index_attachment* passes it the attachment information. After that the attachment is analyzed to verify if the file is indexable, like with computer files, mostly according to its extension. If the file is not indexable, the function’s job is finished. Otherwise, it is verified if, by any chance, the file has already been indexed. If the result is positive, the function returns. If not, it is verified if the file has keywords. If not, it is indexed with the obtained information. In case it has keywords, the keyword frequency is calculated, all properties are prepared and, finally, the document is indexed.

Finally, The *index_email* function is the function responsible for indexing the whole email message. It prepares the message to be indexed and uses the *index_email_document* and *index_attachment* to have the email message indexed. Figure 3.14 represents the execution of the function in the form of a flowchart.

Basically, when the function gets an email message to index, it first gets their properties. Amongst those properties are the participants in the email message, which may be sender, other receivers, etc. So the next step is to index all these contacts. After that, the email message body is indexed using the *index_email_document* function. If the email has an attachment it also gets indexed using the *index_attachment* function. After all this, the algorithm is finished.
Figure 3.12: Flowchart of the execution of the index_email_document function of the mailnib.

3.1.5 Nobs

Nobs is a module that implements functions to perform as an interface with the database and acts as the layer below the nibs on Scribe’s architecture. There are two nobs implemented:

- **highlevel** - Module that implements functions to deal with higher level concepts like determining keywords.
- **lowlevel** - Module that implements functions that return what the knowledge base considers to be identifiers.

The next two sections will present some functions in the two modules previously mentioned.

**Highlevel**

As it was previously mentioned, the highlevel nob implements functions to deal with higher level concepts like determining keywords, performing the TF-IDF algorithm, etc.

**Lowlevel**
Figure 3.13: Flowchart of the execution of the index_attachment function of the mailnib.

Figure 3.13: Flowchart of the execution of the index_attachment function of the mailnib.
Figure 3.14: Flowchart of the execution of the `index_email` function of the mailnib.
Considering the lowlevel nob, as it was already said, it is a nob that implements functions that return what the knowledge base considers to be identifiers. It implements functions to obtain items properties, get identifiers for items of properties, delete an instance, etc.

3.1.6 Accesspoint

Accesspoint is a singleton that gathers the database and all the configurations. It allows to plugins running simultaneously being able to write on the database without risking corrupting the data. It implements functions to:

- Obtain the database object.
- Synchronize the database in memory with the one written persistently on the hard drive.
- Exit and close the database in a clean way.

3.1.7 Parchment

The Parchment module contains all the database logic. It implements the classes of each level of the database:

- Parchment
- Entity
- Property and its subclasses Pattern Property and Numeric Property.

The parchment class is the database which consists in entities. Then, entities have properties in them.

The entities in Scribe are four:

- Document
- Person
- Locus
- Event

The Document entity represents the usual indexable information, like emails and hard drive files. The Person entity, represents people, like email contacts or file creators. The Locus entity identifies a computer. Finally, Events can be a calendar entry or maybe the email event when an email is indexed.

Now, let’s see which properties each of these entities have. These show the systems expressiveness and explain later what is used on social indexing.

First, the Document has the following: name, extension, mimeType, class, path, RFID_ID, created, modified, accessed, size, keywords, creator, title, physicalLocation, deleted, locus.

Then, the Person: names, birthday, emails.
Followed by the Locus: address, description.
And finally, the Event: startTime, endtime, description, type, documentReferences, eventReferences, participants, direction.

The organization of the index is based on inverted indexes. Inverted index is an index data structure storing a mapping from content, like terms or numbers, to their location in a database
file. For example, for the following TX documents, \( T_0 = "it is what it is" \), \( T_1 = "what is it" \) and \( T_2 = "it is a banana" \), the obtained index is:

```
"a":  {2}
"banana":  {2}
"is":  {0, 1, 2}
"it":  {0, 1, 2}
"what":  {0, 1}
```

This index structure is very efficient for term based retrieval. Looking at the example, if what is need is the documents with the word "what", looking for that word on the index gives us immediately the result \( T_0 \) and \( T_1 \).

### 3.1.8 Database

The database consists of basically a parchments instance. Since the persistent index on the hard drive can be very big depending on how much information was indexed, to avoid having all on memory when working with it a mechanism of blocks is implemented. Each block have a given size and there is only a limited number of blocks on memory at a given time. When information from a block on the hard drive is needed and the maximum number of blocks is in memory, one of the blocks is freed from memory using an algorithm based on the least used block and not used for the longest.

### 3.2 Changes to Scribe

In this section the changes that were made in Scribe while the developing of the project will be presented. The biggest changes made were related to improving the "One Time File Indexer" and correcting major bugs related to indexing email information. Other than that, some minor bugs were also corrected and some improvements were made in Scribe.

#### 3.2.1 File Indexer

In this section we will discuss the changes and improvements made to the One Time File Indexer described in the Section 3.1.3. If we recall this plugin, as its name suggests, it indexed the hard drive files exactly one time. This was done using a progress file that at the beginning listed all the directories to index and as the indexing process went on, the file was updated by removing the already indexed files. This was done so that if an unexpected event caused the plugin to stop, the progress was kept to know where to start as the progress went on.

This was changed to make the file indexer plugin more robust. Information on the hard drive is constantly changing, so having only a One Time File Indexer creates a fairly weak index regarding the hard drive information. So a big change made was to drop the "One Time" and create a "File Indexer".

The flow of execution of the new improved File Indexer plugin is depicted in Figure 3.15 using a flow chart.

As we can see the beginning of the process is similar. The steps related with the use of the progress file were removed, because it no longer made sense to have a progress file. The indexing process no longer had a beginning and an end. It only had a beginning. After listing every directory, every file is analyzed and indexed. If the process dies, for some reason, it does begin from scratch, but takes much less time to get to where it stopped, because by looking at Figure 3.15, that depicts
Figure 3.15: Flowchart of the preparation by the File Indexer of a file to be indexed by the filenib.
Figure 3.16: Flowchart of the File Indexer plugin.
the execution of indexing a file, we can see that it is now verified if a file has been indexed before indexing it, and if a given file was in fact indexed, time is saved by not indexing it again.

After indexing every indexable file on the hard drive, the program enters an update cycle where, after waiting a given update interval, the plugin starts to update the information regarding the hard drive items in the indexed. The first step consists of checking for added files to the hard drive and indexing them. To do this a file is created while indexing the files for the first time where information regarding directories’ time of last modification is saved. This is done because in case a directory or file is added to a given directory, its time of modification changes, but not when a file in it is modified. So basically, changes in time of modification in files means a file was added. Since the only information needed regarding directories’ time of last modification is time of modification, this solution is more efficient regarding hard drive space when compared to add entries for directories in the index. This way, this information, that is only used by the File Indexer is kept separate for the index, keeping a smaller index (which improves retrieval time) and occupying less hard drive space with data created by Social Scribe.

When every new file is analyzed and taken care of, the files on the index are updated. To do that, the file entries on the index are obtained and each is analyzed. In case the given file is no longer in the hard drive, it is deleted from the index. If it has been modified the file’s entry is deleted from the database and the file is re-indexed. To do this, the process picks up the file information regarding path and name and gets the file’s current time of modification. This value is then compared to the one on the index, and if it has been changed, the file has been updated. In case a certain exception is raised when trying to obtain the file’s time of modification, it means that the file had been deleted from the hard drive, and that’s how we know when a file’s entry needs to be deleted from the index.

Since we verify if a file has been indexed when indexing it, it’s true that we could have a more cyclic approach to the algorithm and simply, from time to time, restart the indexing process used at the beginning and list every directory and try to index every file. This way we wouldn’t even need to keep a list of the directories’ times of modification. But this would contribute to a decrease in efficiency to the whole process. First because listing every file in the hard drive is slower than listing the file entries in the index. Also, there are less items in the database because, let’s not forget, not all files are considered indexable. So this way, there are less loop iterations. So the reason to have an update algorithm and not repeat the same indexing process from the first indexing loop, is to have a faster update of the database information.

Another change worth mention related to the work done in the process of indexing files is related to a bug discussed in Section 3.1.4 related to the filenib. The fact that when hash duplicates were found while indexing files only the path was added to the entry of which the given file is found to be a duplicate of was already explained to be a flawed method. This is because the file being indexed, while having the same hash key may not have the same name, making the retrieval of that file using the index information impossible. So, the solution is to verify if the file has been indexed resorting to the use of the full file system filename (path and name). And update resorting to the “last time modified” property to update the information on the database. We’ve already discussed that adding the name to the index file entry along with the path was not a valid solution, since it would be impossible to match each path with each name.

### 3.2.2 Email Indexing Process

Regarding the email indexing process there were two major fixes made:

- A bug related to writing on the database
- A bug related to the indexing algorithm on the mailnib
The first one happened was hard to find because the downloading of email messages happened concurrently with the indexing of email messages, so it seemed that the problem occurred after a few emails indexed, when in fact it happened after a few emails were downloaded, but it was as the first email was indexed. The lack of verification if whether the database was empty or not during an action by the mail caused the process to die because the database raised an exception difficult to understand. After several weeks trying to understand the problem, when finally found, was easily corrected.

The second one was related to the execution of the mailnib, where during the parsing of the email’s message body an error on the code caused only a few of the emails to be actually indexed. After indexing an email account, it was verified that only about 10% had been actually indexed. After a lot of time trying to find where the error occurred, it was corrected and all the emails were indexed.

3.2.3 Other changes

Other changes consist basically on some that are difficult to describe because are general and all over the place. These were related to cleaning the Scribe’s code and removing deprecated code that was no longer used. Also some minor bugs that hindered indexing processes were found and corrected.
Chapter 4

Social Scribe

In this chapter, we will approach in a fairly thorough way the solution created in this project: Social Scribe.

![Social Scribe's logo](image)

Figure 4.1: Social Scribe’s logo.

The work included improving Scribe and its components, either by correcting existing bugs or improving the code that has been hindered, has said before in Section 3.1, through several years of making small changes to have a small feature for a given project. Even though improving Scribe was (a big) part of the work done, the most important was making Scribe a tool for indexing social information.

The idea behind the project was to mostly fight the fragmentation of our digital personal information. Before the Internet boomed, all the personal files were in the hard drive. Then we got it in our digital mailboxes. Then, as the Internet grew, cloud applications came along and our personal information became scattered across many different services and applications in physical locations we are not familiar with. Social Scribe comes to allow the possibility of retrieving and possibly managing personal information from different sources in the cloud.

In the next few sections the work done in the development of the solution will be approached and discussed.

4.1 Social Plugins

This subsection will present and discuss the new Social plugins implemented. The word "new" here, is used not only because these plugins are new, but also because we can consider that Scribe already had a, even if little, social component. this if we consider that email can be somewhat considered social information.

For Social Scribe, the indexing of two new sources of information were implemented:

- Twitter
• Facebook

These two sources of social information didn’t get chosen at random, for no particular reason. Nowadays, the two social networks we hear the most about are these two especially Facebook.

The use of plugins to index the user’s information present in these two social networks comes from the already built Scribe’s architecture. It didn’t make any sense to implement the indexing process in any other way, considering that Scribe’s architecture was built and fairly strong.

For each of these two sources a plugin was implemented, and these plugins will now be discussed.

4.1.1 Twitter

Twitter is a social network that allows people to create micro-blogs to communicate. In the context of Social Scribe, Twitter is a source of social information.

To connect to Twitter and get the user information we need to register and application for Twitter developers. This is because Twitter uses OAuth (Open Authorization) which is an open standard for authorization. It allows users to share the personal information (tweets in this case) stored on one site without having to hand out their credentials (username and password in this case). The use of OAuth limits the API in the way we authenticate, however. When we fetch email using IMAP, for example, the approach of using a configuration file to write the credentials in can’t be used, increasing the needs of the tool, e.g. using a front-end approach other than web-based can harm the usability.

Figure 4.2 depicts the execution of the Twitter plugin using a flowchart.

As we can see, the algorithm is fairly simple. After connecting to Twitter, the plugin uses a Twitter lib for Twitter developers to fetch the user’s tweets and replies, which are tweets created by other users directed at a given user, by including in the tweet the expression @<username>. After obtaining all the tweets, they are sent to the socialnib to be indexed. After waiting a given update interval, the index process restarts once more to update the Twitter information.

The plugin fetches every single tweet ever created on the given account, so the information universe is as complete as possible. It’s true that the thought of too much information can cross our minds, but the fact that Twitter is only a couple years old and tweets are limited to 140 characters makes the index size not explode. Besides, if some tweets fail to be indexed the personal information index would be incomplete.

4.1.2 Facebook

Facebook is a social network that was created to connect people and allows its users to post and share messages, videos, photos, links, amongst other things. Like Twitter, in the context of Social Scribe, Facebook is a source of social information.

Just like it was previously explained with Twitter, to connect to Facebook is also necessary to register an application for Facebook developers, because it also uses OAuth. This, while more secure for the cloud application, limits our tool in the ways previously explained in the previous section.

To understand the flow of execution of the Facebook plugin we can look at Figure 4.3 that provides a visual representation of it.

As we can see, the first step of the plugin is to connect to Facebook. Then, by phases, the plugin fetches several types of the user’s Facebook social information and has the social nib indexes them. They are:

• Wall
• Posts
Figure 4.2: Flowchart of the execution of the Twitter plugin.
Figure 4.3: Flowchart of the execution of the Facebook plugin.
• Status messages
• Notes
• Videos
• Photos where the user is tagged
• Photo albums

After indexing the photo albums the plugin gets into a loop where, album by album, it gets the given album’s photos and has the social nib index them. After all the photos from every album get indexed, the plugin waits an update interval to restart the process and update the information on the index.

Connecting to Facebook and manipulating the information is done using a lib for Facebook developers. The information obtained from Facebook comes structured in a JSON (JavaScript Object Notation) with a data field and a possible paging field used by Facebook to limit the amount of information transferred at a time. So, having said that, Facebook, if a given piece of demanded information is too big, sends a piece of it and url’s to the next and/or previous fragment of information.

For the same reason as Twitter, every social item created since the opening of the given Facebook account is created to have a complete universe of personal information.

4.2 Social Nib

Social Nib is the nib that was created and given the responsibility to index the social information obtained from social information sources like the previously mentioned Twitter and Facebook.

The functions this nib implements are:

• index_tweets - This function gets a list of tweets from the Twitter plugin and, for each of them, extracts information and have it indexed by the index_socialitem function.

• index_fbposts - This function gets either a list of Facebook wall items, Facebook posts or Facebook status messages from the Facebook plugin and, for each of them, extracts information and have it indexed by the index_socialitem function.

• index_fbphotos - This function gets a list of Facebook photo items from the Facebook plugin and, for each of them, extracts information and have it indexed by the index_socialitem function.

• index_fbalbums - This function gets a list of Facebook album items from the Facebook plugin and, for each of them, extracts information and have it indexed by the index_socialitem function.

• index_fbnnotes - This function gets a list of Facebook notes from the Facebook plugin and, for each of them, extracts information and have it indexed by the index_socialitem function.

• index_fbvideos - This function gets a list of Facebook video items uploaded by the user from the Facebook plugin and, for each of them, extracts information and have it indexed by the index_socialitem function.

• index_socialitem - This function gets a social item’s information from one of the other functions implemented in the socialnib and use it to index the mentioned social item.
Figure 4.4 depicts the execution of the functions used to prepare social items to be indexed (all the above except \texttt{index\_socialitem}). The algorithm is simple. After getting the social items, it enters a loop where for each it verifies if it is already indexed. If not, it prepares the properties for indexing and it calls \texttt{index\_socialitem}. When every item on the list is analyzed and indexed, the process finishes.

Figure 4.5 illustrates the execution of the function \texttt{index\_socialitem}. As we can see, after getting the item properties it calculates the keyword frequency. The next step is index the items authors as Persons and prepare the properties. After that is done, the item is indexed.

Like with other Scribe plugins explained on the previous chapter, the indexing process does not totally start from scratch when the application starts, whether the tool stopped during the indexing process or during an idle time. When an item is being index it is always verified whether the item was already indexed or not. There is no need to verify modified time because these items cannot be modified. After being created, they can only be deleted. Therefore, executing the indexing process for a second time makes no difference, except updating information by indexing unindexed items.

Every item is indexed as a Document only, since it was considered that there was no need to associate it with an Event. This was decided for two main reasons:

1. **It is not necessary** - Considering the information on the Document entity is enough to know if an item has been indexed (using the \texttt{created} property), using the Event entity lost relevance.

2. **Efficiency increase** - Not using Event makes the tool more efficient by losing less time and hard drive space creating a new entity.

When a social item is indexed, only some of the Document entity are needed and, therefore used:

- **path** - Includes a web link to the social item
- **name** - Name or title of the item
- **size** - Size of the item
- **created** - Time of creation of the item
- **creator** - Reference to the Person that created the item
- **extension** - Type of the item (tweet, facebook photo, facebook post, etc.)
- **keywords** - List of keywords present in the item.

Also, a change was made on the Person entity to be able to correctly index the authors. Two fields were added:

- **fb\_id** - The user’s Facebook ID
- **twitter\_id** - The user’s Twitter ID

This way, the information about the user’s social networks’ unique ID’s is indexed. With this change, obviously came a change in the Person indexing function to try to reconcile the entity information based on the social account user’s name and email, to avoid creating separate Person entities for the same actual real life person. The option to only index contacts from the social networks when they contributed with an item considered relevant for indexing was made to not index irrelevant people. Considering limitations from the API to have access to social contacts email addresses, the only information we can get is names, it seems appropriate that Persons are only created to be linked to content in this case, because as contacts, only a name isn’t for much use.
Figure 4.4: Flowchart of the execution of the functions used to prepare social items to be indexed.
Figure 4.5: Flowchart of the execution of the index.socialItem function.
4.3 Social Scribe’s front-end

Something we consider a big improvement in Scribe is the existence of a front end to use it. In Scribe all the information needed was written and edited manually in the configuration file. Before starting Scribe the user would have to open the configuration file, edit the `load` string (plugins to start), edit credentials, server addresses and directories where needed, and the user could launch Scribe. Now, the user has a GUI to enter the information necessary for Social Scribe to start its indexing process.

Considering the front-end was built thinking of Social Scribe, it included the fields necessary for the use of Social Scribe, in this case four plugins:

- Facebook
- Twitter
- Email (IMAP4)
- Hard Drive

To connect to Facebook and Twitter, a button was added that links to a page where the user logs in and authorizes the plugin to access the user’s information on the given social network. If Social Scribe is not connected to one of the social networks, they won’t be indexed. When it comes to the file indexer, the user has a combobox to chose whether the hard drive is supposed to be indexed or not. Also, the indexing root is the user’s home directory, by default, but the user can change it at will. Finally, regarding email information, IMAP was the protocol chosen to allow email indexing (although this can easily be changed) because it’s the protocol most used and every email service provide an IMAP server. If login information is not entered, email will not be indexed. The directory where the index is to be stored is a folder called `index` on the same level as the file from where the Social Scribe was launched by default, but it can be changed by the user.

Below the options of what to index, there is a button to start the indexing process. If information has already been indexed and the user wants to move on to the page of the retrieval tool, there is a link for that.

Regarding the front-end’s back-end, to accomplish this, Scribe became a function rather than a script. This function is called when the order to begin the indexing process is given. the file to launch the tool ceased to be a simple script of Scribe and became a HTML server that implements the functions needed for the execution of the tool.

4.4 Social Scribe’s Retrieval Tool

The main and most important work done on the Social Scribe project is, without a doubt, the work done in indexation, because it’s the main purpose of the project. However, to verify and test the information indexed, a retrieval tool to obtain the information indexed by the indexer plugins was created.

Basically the retrieval system consists of a textbox where the user enters the terms to search. Then the system returns a set of results. Visually we kept the design as simple as possible, trying to avoid the same mistake as Phlat[5] regarding interface simplicity.

Figure 4.7 depicts a set of results to a query. On each result we can see the item’s source from a small logo next to it and some information that maybe important to understand if that is the item the user was looking for. Is possible to also see the textbox in the figure. When too many results are found, they are divided across several pages with a given number each. This can be seen on Figure 4.8.
Connect to Facebook if you want your Facebook information to be indexed.

Connect to Twitter if you want your Twitter information to be indexed.

**Hard-Drive**

Don't index HD

Indexing root: C:\Users\Shadez

Select the correct option to indicate if you want your hard-drive files to be indexed. If you choose to index the hard drive, choose the root directory from which you want to do it.

**Email (IMAP)**

Username: 

Password: 

Server: imap.gmail.com 993

Fill your IMAP information to have your emails indexed.

Save index in: C:\Users\Shadez\Dropbox\Social Scribe\eclipse\src\Scribe\index

Start indexing...

Click here if information is already indexed and you just want to search it.

Figure 4.6: Screenshot of Social Scribe’s front-end.
Figure 4.7: Screenshot of Social Scribe’s retrieval tool. We can here see the textbox where terms are entered for search and the look of the results.

The reasons for choosing this visualization are that it allows to have a lot of results showing only essential information without taking much space. If the user wants more information concerning a particular result, it is possible on demand. It is simple and effective. As the interface was developed, we kept in mind that too much information could confuse the user and provide a poor usability experience.

Since Social Scribe is a HTML server, when the search button is pressed a function on the server is called and starts the search algorithm. First the input is parsed separating the several terms. After that, the index is accessed and every file with the terms in the **keywords** property are obtained. As we have explained before, there is an inverted index and for the field **keywords** of a "Document" is easy to obtain the files with the given keywords. Since terms may be important for other fields like name, title extension or author, these fields are then checked too. After obtaining all the files they are ordered according to two values:

- **How many terms did that result hit**
- **How many times do the given keyword appear on the item**
Figure 4.8: Screenshot of Social Scribe’s retrieval tool. We can see the result’s paging here.

Figure 4.9: Screenshot of Social Scribe’s retrieval tool. We can see here an open result.

Figure 4.10: Screenshot of Social Scribe’s time of indexing report.
Basically, the more terms are present on the item, the better the item is ranked. The more they are repeated in it, the better ranked it is. A better ranking algorithm could have been worked on using methods like vector terms and TF-IDF, but, from what was tested, to calculate TF-IDF caused the application to be very much slower, like thousands or tens of thousands times slower. Since the major purpose of the work is to have an indexing tool and the retrieval tool is just to test the values on the database, it was decided that if the time of retrieval was that hindered, it was not worth it.

Having obtained a set of results, we can open one or more and obtain more detailed information. This is depicted in Figure 4.9. We could show for each result on the set all the information when the result set is returned, but if the set is compacted is easier to have an overview of the results. This way we have a small bar with the most important information and then the bar is opened and more information is shows. As we can see some information consists of url’s. We consider this feature very important because, since the retrieval tool was created to view and navigate through the indexed information, this way the navigation is easier. With one click we can get all the items of a given information source, see the information about a given creator and list his/her items, get all the items created around a given time and also get all the items of a given type. Also, there is a link that opens the location of a given item:

- **Facebook** - A web page of the item is opened.
- **Twitter** - A web page of the twitter is opened.
- **Email** - A web page of the email is opened.
- **Files** - The folder where the file is located is opened on the file system.
Figure 4.13: Screenshot of Social Scribe’s creators graphs report.
Other than the retrieval part of the tool, an important component was implemented: Report. By clicking a button a report on the indexing process is shown. This was done so the user can get a notion of what were the results of the indexing process in a more graphical way.

On the Report, lots of different information can be obtained. For example, we can obtain the time each of the sources took to index the information in the form of a bar chart. The total time is the biggest one, since every plugin runs at the same time. This is depicted in Figure 4.10.

We can also see how many items were indexed, again on the form of a bar chart. This way we can see the size of the database in terms of items, by adding the amounts of items of each source. This is presented on the application can be seen in Figure 4.11.

A timeline depicting the items creation as time went by, based on the property of time of creation each item has, is also presented, as show in Figure 4.12.

Finally, the last information shown on the report regarding the creators on the database and how many items were created on each source. This is depicted in Figure 4.13. Figure 4.14 shows an example easier to explain. Each black bubble represent a creator. The creator is linked to a certain amount of bubbles representing each one source and its size the amount. By the color we can see the source, but by hovering the bubble we can see not only the source, but the exact amount of items also. When there are items created by more than one creator, the creators’ bubbles of the type of the items shared are linked.

4.4.1 Architecture

The retrieval tool’s architecture is simple and depicted on figure 4.15. As we can see, when the user wants to perform a search activity, the information is inserted in the interface module and a retrieval function is used on the server. This functions uses the Nobs and the Parchment modules to get the information and build the data structure used by the interface to present the results to the user. This structure was chosen to use Social Scribe’s architecture as a basis. The server used is the same as the one responsible for the indexing process. When the process starts, one thread is responsible for it while a second one waits for retrieval requests.

The page navigation is made through communication exchanges between the client on the desktop
and the server. When a button is pressed it calls a function, eventually using parameters, on the server that generates the necessary information to information and builds an HTML page.

Next, we'll present the technology used in the development of the retrieval tool.

**Technology used**

Two different technologies were chose for two different purposes: front-end and back-end.

For the back-end, Python was used to keep consistency with everything else developed and since some of the modules needed were implemented in it. A library called cherrypy was used for easily allowing the creation and customization of a HTML server.

The front-end was HTML, like Social Scribe’s front-end. Javascript was used to page the results and keep data structures of the obtained information from retrieval requests. To obtain a more appealing interface, a Javascripts lib was used, JQuery. This allowed more appealing interaction effects like slide opening a result for more detailed information about it.

Figure 4.15: Depiction of Social Scribe’s retrieval tool’s architecture.
Chapter 5

Evaluation

This chapter presents an evaluation made to the proposed solution, Social Scribe, explained how it was made and what results were obtained using three subsections:

- Tests performed to evaluate the indexing process
- Tests performed to evaluate the retrieval tool
- Results obtained from the tests and their analysis

The tests performed had the purpose to evaluate not only how well the tool works but also its usability and if users found it difficult to use.

5.1 Testing the Indexing Process

To test the indexing process, a simple approach was used: each user of the test group launched it and it wrote useful information for statistics on a log file:

- How much time did each source take to index
- How many items were found on each source
- How many items were indexed on each source
- Why have a given item not been indexed
- How many times did a given item been found on different sources
- What size did the index turn out to have

With the obtained information we could verify if there was a limit found for how many items could be correctly indexed without being hurtful for the operating system and verify if there is a point from which the tool works less well. It allows us to have a perception of what the system can do.

User interaction was not tested on the indexing process because it is almost nonexistent since the user only enters information for the indexing process, like index location and email credentials, and also connects the tool to some sources (social networks).
5.2 Testing the Retrieval Tool

The test conducted on the retrieval tool focused more on user interaction and user experience than
the ones conducted on the indexing process because this tool’s success is much more coupled with
usability.

The test consisted of two main parts:

- **Usability tasks** - The user was asked to perform a set of tasks to verify how well the system
could be used.

- **Satisfaction Questionnaire** - The user was asked to fill a questionnaire to verify how he
or she felt like while using the system and to get some extra feedback concerning what the
user thought about it and what could be done to improve it.

The major adversity of evaluating PIM (Personal Information Management) tools are that,
considering it is personal information, i.e. information only the user can relate to, the collection
of items and the results are completely unknown to the developer/tester. So we must rely on the
user to choose what to look for and give feedback to confirm the objectives were achieved.

Before performing the task, the users were given a brief period of time, around five minutes, to
explore the system and understand how it works. Then they were asked to perform the following
set of tasks:

1. Think of a file existing on your hard drive and perform a search introducing the keywords
   you think to be relevant. Then, open the file location.

2. Obtain every file of the same filetype as the result chosen.

3. Think of an email existing on your email account and perform a search introducing the
   keywords you think to be relevant. Then, open the email location.

4. Think of an item on your Facebook account and perform a search introducing the keywords
   you think to be relevant. Then, open the item location.

5. Obtain every item indexed from your Facebook account.

6. Think of a tweet on your Twitter account and perform a search introducing the keywords
   you think to be relevant. Then, open the item location.

7. Of the last set of results obtained, pick one and get all the items created by that item’s
   creator.

Obviously some tasks were not possible to be performed depending on the user indexing the
needed source (Twitter). The purpose of these tasks was to exercise the several parts of the system
and observe the users’ reactions to it.

Regarding the Satisfaction Questionnaire, it was composed by a set of thirteen questions. The
first ten were sentences where the user was asked to check a number on a scale from one to five,
where one meant the user strongly disagreed with the sentence and five meant the user strongly
agreed with it. After that the user was asked to answer three essay questions where opinions could
be expressed more freely.

The first set of ten sentences consisted of the following:

1. I think that I would like to use this system frequently
2. I found the system unnecessarily complex
3. I thought the system was easy to use
4. I think that I would need the support of a technical person to be able to use this system
5. I found the various functions in this system were well integrated
6. I thought there was too much inconsistency in this system
7. I would imagine that most people would learn to use this system very quickly
8. I found the system very cumbersome to use
9. I felt very confident using the system
10. I needed to learn a lot of things before I could get going with this system

This is the SUS questionnaire (System Usability Scale)[2], a standard usability test that users answer after using a system and that has an algorithm to obtain a quantitative test result from 0 to 100. This is a good way to evaluate a system from its usability point of view, since we get a numeric result easy to analyze.

The essay questions after the SUS are:

1. Enumerate what you consider to be the biggest advantages of the system.
2. Enumerate what you consider to be the biggest advantages of the system.
3. Leave a few comments that you consider might be helpful to improve the system.

The purpose of these questions are to let the user have a space to give a free opinion and help us understand what can be done to improve the system.

5.3 Tests Results

In this section the results of the tests performed on both the indexing process and the retrieval tool will be presented and discussed.

The user group is characterized by the following characteristics:

- 15 users
- Approximately 73% male users and 27% female users
- Only approximately 27% of the users had Twitter accounts
- Age average of 23 years old (18 to 28)
- University students
- Experienced computer user’s
5.3.1 Indexing process

Regarding the indexing process, several measures were obtained and now the values will be presented and discussed. These measures are:

- Time of indexing per source
- Number of found items per source
- Number of indexed items per source
- Percentage of indexed items per source
- Average time of indexing of each item per source
- Size of index
- Number of items duplicated on different sources

A very important thing to mention is the fact that every user that tested the tool have indexed their hard drive, email and Facebook information. However, of the 15 users in the test group, only 5 of them indexed Twitter information. This is because it was found to be difficult to obtain Twitter users among the people willing to test the tool, considering that, specially the indexing process, requires the tester follow it more closely.

Also, to make the results analysis easier, specially when it comes to understanding standard deviation values, individual user times of indexation, number of items found and number of items indexed, per source, are represented on 5.2, 5.3 and 5.4, respectively.

Time of indexing per source

Figure 5.1 presents a chart representing the average time (seconds) of total indexing per source and error bars representing the standard deviation for the same parameter. Table 5.1 also presents the values for these parameters.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>HD</th>
<th>Email</th>
<th>Facebook</th>
<th>Twitter</th>
</tr>
</thead>
<tbody>
<tr>
<td>Average</td>
<td>1863,73</td>
<td>10005,7</td>
<td>146,27</td>
<td>4,5</td>
</tr>
<tr>
<td>Std Dev</td>
<td>896,29</td>
<td>7438,51</td>
<td>127,04</td>
<td>0,58</td>
</tr>
</tbody>
</table>

Table 5.1: Average and standard deviation for the time of indexing values in seconds.

Clearly email is the source that usually takes more time to index. This can easily be explained by the fact that, not only do users have usually more items on the email box to index than on Facebook or Twitter but also because everything has to be downloaded to be indexed, including attachments, which can be considerably big and take some time to download. Needless to say that, since items are downloaded from the Internet, not only the amount of items is determinant for the indexing time, but also the Internet connection and the user’s email server’s bandwidth. Because of the download time, even if the amount of hard drive items can be significantly larger than email’s, to index hard drive no Internet connection is needed and items don’t have to be downloaded, so hard drive indexing is usually a faster process.

If we look at the data for standard deviation, some results can seem somehow odd for being to high. It can be verified that to each source separately, the time of indexing is very much coupled with the amount of items. The big standard deviation regarding time of indexing on some sources...
Figure 5.1: Average and standard deviation for the time of indexing (in seconds) values.

can be explained by the fact that among the population of users that tested the system, the disparity between the amount of items each had was big. For example, regarding email, the user with the lowest amount of emails had 42 and 10 attachments, while the one with the biggest amount of items had 5632 emails with 321 attachments.

**Number of found items per source**

If we look at Figure 5.5 we can see a graphic representing the average amount of items found per source with error bars representing the standard deviation for the same parameter. Table 5.2 also presents the values for these parameters.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>HD</th>
<th>Email</th>
<th>Facebook</th>
<th>Twitter</th>
</tr>
</thead>
<tbody>
<tr>
<td>Average</td>
<td>18909.13</td>
<td>2709.27</td>
<td>530.33</td>
<td>21.75</td>
</tr>
<tr>
<td>Std Dev</td>
<td>8740.79</td>
<td>2046.21</td>
<td>710.75</td>
<td>7.14</td>
</tr>
</tbody>
</table>

Table 5.2: Average and standard deviation for the number of found items values.

So, regarding the number of items found on each source we have a clear winner. As expected, the hard drive is the source with the biggest amount of items. This shows users keep a lot more items in their hard drives. Even though email boxes are much bigger nowadays than about 10 years ago, when a Hotmail user had a 2MB limit inbox, compared to the approximate 7 GB a Gmail user is given these days, it still is nothing compared to how easily someone can get a 1 TB hard drive nowadays. Obviously, there are more reasons than that. It’s true email can be used for storage by attaching items to a message and emailing it to ourselves, but obviously hard drive is
Figure 5.2: Individual times of indexing, by user, per source.
Figure 5.3: Individual numbers of items found, by user, per source.
Figure 5.4: Individual numbers of items indexed, by user, per source.
much better for storage, as it is designed for it while email is designed for communication. It is not much of a surprise either the fact that there are much less items on users’ social networks accounts, considering the fact that nothing is usually stored on social networks, since they’re usually used for their main purpose which is connect people and share information.

Once again the big difference among the amounts of information of the several users caused the values of the standard deviation to be abnormally big on some cases, with the focus being again on email in terms of percentage of the average. The reasons for this have already been explained.

**Number of indexed items per source**

On Figure 5.6 we can see again a chart, now representing the average amount of items indexed per source and error bars representing the standard deviation for the same parameter. Table 5.3 also presents the values for these parameters.

![Figure 5.5: Average and standard deviation for the number of found items values.](image)

<table>
<thead>
<tr>
<th>Parameter</th>
<th>HD</th>
<th>Email</th>
<th>Email Attachments</th>
<th>Facebook</th>
<th>Twitter</th>
</tr>
</thead>
<tbody>
<tr>
<td>Average</td>
<td>18742.6</td>
<td>2707.53</td>
<td>150.2</td>
<td>471.13</td>
<td>21.75</td>
</tr>
<tr>
<td>Std Dev</td>
<td>8778.67</td>
<td>2044.81</td>
<td>113.26</td>
<td>658.3</td>
<td>7.14</td>
</tr>
</tbody>
</table>

Table 5.3: Average and standard deviation for the number of indexed items values.

The number of indexed items is obviously equal or smaller than the number of items found and previously presented. This number alone is not as interesting as when combining it with the previous reveals how much of the found items have not been indexed. An interesting value is presented here, though: the number of indexed email attachments. The number of email attachments found was not
presented, because when the email indexing plugin lists the found email messages before starting the indexing process it is not know which have attachments and how many. However, since the attachments are indexed using the same tools the file indexer plugin uses, the percentage of failure and the reasons should be approximately the same.

However, the standard deviation is almost as big as the average, and the reason is, as previously mentioned, some users have a relatively small email box, in terms of amount of emails, because, according to them, they delete most of their emails unless they consider the content to be of major importance. Therefore, less emails means less attachments, at least if the difference between number of files is big.

**Percentage of indexed items per source**

Figure 5.7 depicts a graphic representing the average of the percentage of indexed files per source considering the number of found items and error bars representing the standard deviation for the same parameter. Table 5.4 also presents the values for these parameters.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>HD</th>
<th>Email</th>
<th>Facebook</th>
<th>Twitter</th>
</tr>
</thead>
<tbody>
<tr>
<td>Average</td>
<td>0.99</td>
<td>1</td>
<td>0.8</td>
<td>1</td>
</tr>
<tr>
<td>Std Dev</td>
<td>0.02</td>
<td>0</td>
<td>0.13</td>
<td>0</td>
</tr>
</tbody>
</table>

Table 5.4: Average and standard deviation for the percentage of indexed items considering the found items.

The data and visual information is curious. At a first glance, we would think that there is no
failure whatsoever when indexing email but there are some problems when indexing the hard drive and specially Facebook items. This information is not completely truthful. Regarding email (not including attachments) and twitter it is true that there were no problems found when indexing these two sources. However, the average 80% of items indexed when it comes to found Facebook items is a false value, because when information from Facebook is fetched it is sometimes overlapped. For example, when user’s posts, user’s status messages and user’s wall are fetched the found items are the sum of all, however some of the items on these lists overlap. The same could happen when the list of photos where the user is tagged and the photos on the user’s albums are fetched. So, it was verified that every item on Facebook was indexed, but when a given item fetched had been already indexed it is ignored the second time. Regarding the hard drive, the 1% of information that failed to be indexed were verified to be because the files’ extensions were sometimes of unknown types that could not be indexed for lack of an appropriate parser.

Now, if we look at the standard deviation, the values related to email and Twitter are normal considering there was a 100% value of indexed items from these sources. The hard drive value of standard deviation was low because users usually have a really small portion of files that cannot be indexed, while from Facebook it depends on how much of the information overlaps on the several lists obtained to download it.

**Average time of indexing of each item per source**

On Figure 5.8 we can see a chart representing the average time (in seconds) each item took to index per source and error bars representing the standard deviation for the same parameter. Table 5.5 also presents the values for these parameters.

As we can see, the time a hard drive file takes to index is the lowest of every source. A probable
Figure 5.8: Average and standard deviation for the time (in seconds) of indexing per item.

Table 5.5: Average and standard deviation for the percentage of indexed items considering the found items.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>HD</th>
<th>Email</th>
<th>Facebook</th>
<th>Twitter</th>
</tr>
</thead>
<tbody>
<tr>
<td>Average</td>
<td>0,12</td>
<td>5,38</td>
<td>0,53</td>
<td>0,22</td>
</tr>
<tr>
<td>Std Dev</td>
<td>0,08</td>
<td>7,67</td>
<td>0,27</td>
<td>0,04</td>
</tr>
</tbody>
</table>

explanation for this values lies on the fact that hard drive is the only source that did not rely or needed Internet connection to work and information did not need to be downloaded for the indexing process to execute. It has however, percentage wise, a high standard deviation, only to be surpassed by email. This is because the time an item takes to index varies a lot depending on the size of the file and how rich it is in keywords. On the other hand, email is the source from which an item (including attachments) takes the longest to index. This happens probably because, not only items have to be downloaded to be indexed but also an email item has a higher probability of being bigger when compared to tweets and Facebook items. Specially when we consider attachments that in some cases, like Gmail, can be up to 20 MB, which can take a considerable amount of time to download and index. The standard deviation for the indexing time per item is also the really high, percentage wise, when looking at email. This is understandable since each user on the test group has a different Internet connection and therefore different download speeds. Also, the percentage of email box storage space occupied by attachments varies a lot, and so some users may have less items to download but a lot of attachments while others have a lot of items but few attachments. This leads to a big difference in time of indexing per item.
Size of index

Regarding the size of index of each user, the average and standard deviation were also calculated:

- **Average:** 610.93 MB
- **Standard Deviation:** 319.38 MB

Once more, the standard deviation is considerably high and nearly half the average. This was somehow expected and with the big disparity in values between users the standard deviation being in general high. So it was expected that the same happened to the size of the index, considering it is a value very much coupled with the number of items indexed.

Number of items duplicated on different sources

Finally, regarding how many files were found duplicated across different sources the same values (average and standard deviation) were calculated:

- **Average:** 5.86
- **Standard Deviation:** 5.08

The low average is explained by the fact that users confirmed that most of the times, when an attachment is received they see it and delete it or if it is to be stored in the hard drive, they delete it from the email account. The high standard deviation is explained, again, by the big differences between the values of this parameter among users. Some users had no duplicates in different sources, however, a couple had relatively higher values, e.g. 16.

Conclusions

From the evaluation of the indexing process several conclusions can be drawn.

First is that standard deviations were found to be considerably high due to how the results from the users’ personal information being very different between each user.

Also, email’s time of indexation, is more dependent of the size in MB of the used email account space than the number of email messages, considering the time of download plays a big part on the process and item’s size being bigger than other items that needed to be downloaded. It also can vary a lot depending on the Internet connection.

5.3.2 Retrieval Tool

Regarding the retrieval tool some values were obtained for evaluation purposes:

- Times of execution of each task
- Errors made while execution each task
- Satisfaction questionnaire results

88
Figure 5.9: Average and standard deviation for the times of execution of each task.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td>Average</td>
<td>31</td>
<td>18.27</td>
<td>25.33</td>
<td>12.27</td>
<td>12.6</td>
<td>11</td>
<td>9.2</td>
</tr>
<tr>
<td>Std Dev</td>
<td>15.26</td>
<td>14.28</td>
<td>12.32</td>
<td>9.22</td>
<td>15.99</td>
<td>1.15</td>
<td>3.19</td>
</tr>
</tbody>
</table>

Table 5.6: Average and standard deviation for the times of execution of each task.

Times of task execution

Figure 5.9 depicts the average and standard deviation for the times of execution of each of the seven tasks users performed to evaluate the retrieval tool. Here we’ll discuss these results.

Let’s remind the tasks:

1. Think of a file existing on your hard drive and perform a search introducing the keywords you think to be relevant. Then, open the file location.

2. Obtain every file of the same filetype as the result chosen.

3. Think of an email existing on your email account and perform a search introducing the keywords you think to be relevant. Then, open the email location.

4. Think of an item on your Facebook account and perform a search introducing the keywords you think to be relevant. Then, open the item location.

5. Obtain every item indexed from your Facebook account.

6. Think of a tweet on your Twitter account and perform a search introducing the keywords you think to be relevant. Then, open the item location.
7. Of the last set of results obtained, pick one and get all the items created by that item’s creator.

The first task was the one that took the most time to perform. The reason is probably due to being the first one and the user not being prepared to taking time to think about what’s on the hard drive and thinking of a god item to search. The standard deviation is considerable because there was a big difference between users regarding the time they took to think of a file to look for. The variation on retrieval time may have played a role in it because depending on the cases there may have been differences of a couple of seconds between different users.

Regarding the second and fifth tasks they got abnormally high averages and standard deviations. This is because there were two users in particular that took a long time to understand that to fulfill the tasks a simple click on a link of the extension and of the source would be enough to obtain the desired results. Because of this, the time average was considerably hindered and the values regarding standard deviation were really big, percentage wise, when comparing to the average.

The time between the first task and tasks number three, four and six got significantly reduced since the users that had more problems with the first task and were more prepared for the others and also understood more what was wanted from them.

When users arrived at the seventh task they already assimilated the use of links and performing it was easy.

**Errors in task execution**

Regarding errors made by users while performing the tasks, Figure 5.10 presents the average and standard deviation for them.

As we can see, most of the tasks were performed error free with the users fully understanding
Table 5.7: Average and standard deviation for the errors while performing each task.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td>Average</td>
<td>0</td>
<td>0,13</td>
<td>0</td>
<td>0</td>
<td>0,13</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Std Dev</td>
<td>0</td>
<td>0,35</td>
<td>0</td>
<td>0</td>
<td>0,35</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

Figure 5.11: Average and standard deviation for each answer to the SUS test.

what was expected and using the right tools and methods to accomplish it. However, there were a couple of users that had a problem with two tasks. Tasks number two and five were already explained on the last section that the right and fastest way to obtain the expected final result was by using links. Both the users were found to commit an error because they had not quite understood the purpose of the links present on a given result and used the search box to obtain the results, even if it took longer and obtained a set of results that included unwanted items. So the task was in fact finished but an error was considered to have been made.

**Satisfaction Questionnaire Results**

Now we’ll present the results from the satisfaction questionnaire. As we’ve said before, this questionnaire is constituted by a first part consisting of a SUS and a second part with essay questions for user opinions.

Regarding the SUS test, Figure 5.11 presents and average and standard deviation for the value each answer on the test. Let us remind the questions:

1. I think that I would like to use this system frequently
2. I found the system unnecessarily complex

91
Table 5.8: Average and standard deviation for each answer to the SUS test.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
</tr>
</thead>
<tbody>
<tr>
<td>Average</td>
<td>4.2</td>
<td>1.27</td>
<td>4.07</td>
<td>1.53</td>
<td>4</td>
<td>1.13</td>
<td>4.47</td>
<td>1.27</td>
<td>3.67</td>
<td>1.53</td>
</tr>
<tr>
<td>Std Dev</td>
<td>1.01</td>
<td>0.46</td>
<td>0.96</td>
<td>0.74</td>
<td>0</td>
<td>0.35</td>
<td>0.74</td>
<td>0.46</td>
<td>1.23</td>
<td>0.52</td>
</tr>
</tbody>
</table>

3. I thought the system was easy to use
4. I think that I would need the support of a technical person to be able to use this system
5. I found the various functions in this system were well integrated
6. I thought there was too much inconsistency in this system
7. I would imagine that most people would learn to use this system very quickly
8. I found the system very cumbersome to use
9. I felt very confident using the system
10. I needed to learn a lot of things before I could get going with this system

Being each sentence given a grade from 1 (totally disagree) to 5 (totally agree), it’s obvious that
the ideal is to have answers as close to 5 as possible for sentences with number 1, 3, 5, 7 and 9, and
answers as close to 1 as possible for the remaining.

We can see in Figure 5.11 that the answers seem good. Of the answers we expect high grades, it’s
easy to notice sentence 7 with the highest average and a low standard deviation, which means the
opinion that the system is easy to learn was consistent. Also we can see that everybody considered
the features of the system well integrated, since everyone graded 4. Also regarding sentence number
9, we can see that is lower than ideal and the standard deviation was somewhat high, which means
that even though some users were indeed confident using the system, a few were not.

Looking at the remaining answers, they were not that bad considering 1 is the minimum and
best possible. We can see that most users agreed that there was no inconsistency in the system,
which is good. Even though the best results were related to sentence number 6, the rest of the
results are very similar, which shows a consistency of big results.

SUS has an algorithm to calculate the SUS grade (1-100) according to the answers of the test.
After obtaining the results for the tests of each user, the average and standard deviation were
calculated:

- **Average**: 84.17
- **Standard Deviation**: 13.18

As we can see the results were positive. 84.17 out of 100 can be considered a good result. However
the standard deviation is somewhat high. This is because a couple of users had a considerably lower
value, of around 70, which is lower than what we consider to be good and far from what the tool
can achieve. This means that there is definitely still work to do.

After the SUS, users were asked to present advantages and disadvantages of the system, and
finally give some opinions they may consider important to improve the system.

The main advantage pointed out was the fact that it fights the fragmentation of information,
which is the main purpose of the system, and allows users to find information from the various
sources. Also it was pointed by some users that the ranking algorithm was not temporal which some considered a positive thing. They ofund the tool in general easy and simple to use, and consistent. A user in particular considered that indexing content and not only names and titles was really important and that the fact that it is web-based brought a familiar component to the tool.

A disadvantage that was pointed out was related to the indexation time, which can be big for users with a big load of items to index. Also the fact that users cannot filter the search by fields like title, name or author was considered a disadvantage by some users. One person found a feedback failure in usability which is easy to correct, that is the fact that the terms used on the search are not shown with the set of results. Finally, a user considered the retrieval time to be slow when compared with some search tools they are familiar with, like Google.

Some comments made by the user were considered interesting and helpful. Besides the option to filter results, some found that there should be an option to return the set of results ordered by time, even though the existence of a ranking algorithm was found an advantage. A couple of users also thought it would be good to have support for other languages besides English and that different color themes should be available.

Conclusions

Of the evaluation of the search tool, some conclusions could be drawn.

First, users adapt and learn easily how to use the tool. This can be verified considering how more efficient they became doing similar tasks throughout the use of the application.

Users revealed a problem with understanding and adjusting to the use of links on the found results, that’s why errors were only made on tasks that involved the use of links to obtain a complete, accurate and expected set of results.

Also, users welcomed well the tool, as we can see from the SUS results. Nonetheless it was revealed when talking to users that some people are reluctant to change and are happy with the existing solutions for each information source.
Chapter 6

Conclusions and Future Work

Throughout the development of this project, many conclusions could be drawn from published works and from the investigation process.

Users nowadays have their personal information scattered all over the place with the growing use of cloud applications and social networks. Nonetheless, they seem to not realize, since most of them claim not to really think about this fact and acknowledge the problem. However, from the investigation we can conclude that users, when presented with a solution to the problem, realize how simple managing their personal information would be, and so admit that if a solution for these problems existed they would possibly embrace it and use it. A solution to try to fight the problem of information fragmentation did not exist, at least documented, before the development of this project, so that’s probably a reason that users, being used to working with what they have, did not know they had such a need until the gap was filled.

We gave the first step in unifying the several sources of personal information by creating Social Scribe. What makes us conclude that users would appreciate the idea in general are the test results that show how they were receptive to the tool and to the idea. We developed a possible solution by creating a unified index where information from the several sources of personal information is kept. Only a few social informations sources were included among the plugins that implement the indexing process, choosing those that are most used\(^1\), specially Facebook, which has an amazing number of acting users (over 800.000.000\(^2\)) and keeps growing.

To accomplish the task an existing indexing tool called Scribe was improved and used as the basis for the development of the Social Scribe. This tool creates a persistent database where entities of three main types are stored:

- **Events**
- **Documents** (which can be any kind of indexable items, from tweets to hard drive files)
- **Persons**

This tool was plugin-based, which means that the architecture was designed to be as flexible as possible in terms of adding sources to be indexed. This means that to index a new source, a developer needs to create a new plugin and, if necessary, create the function to write the organized data on the database.

Tests show that index all the information can be a slow process (more than one user took over 5 hours to complete the indexing process) depending on the amount of items a user needs to be indexed, but this is the first time Social Scribe runs. Since every plugin used by Social Scribe

\(^1\)List of most popular social networks - http://www.ebizmba.com/articles/social-networking-websites

updates the information of its source every given interval of time, the slower part of the indexing process is finished after the first time it runs. The retrieval tool created to test the Social Scribe solution as an indexer also was proven to be usable and easy to learn, helping the user manage all the information, considering it was given a good SUS grade by the users: average of approximately 84 out of 100.

In terms of future work, we believe the first step is to create more plugins to more possible sources of information. An interesting possibility is Google+ that was released in the end of the development of Social Scribe and still lacked an API to fetch the information, but is already one of the most used social networks in the world. Also other services for more specific users could be interesting, like Flickr that has the purpose to share photos.

Also, the retrieval tool is not as good as it can possibly be and so it would also be important to use all the user feedback and suggestions to improve it. This includes allowing time ranking on the set of results, filtering results by fields, etc.

In the end, the evaluation proved the creation of good and robust integrated indexes to allow users to find their fragmented personal information wherever it is.
Bibliography


