Carousel: A Virtual File System With Multiple Categorization and Live Queries

Patrick Dubroy
pdubroy@gmail.com

COMP 4905 – Honours Project
Supervisor: Dr. Dwight Deugo
School of Computer Science
Carleton University

April 8, 2004

Abstract

This report describes the design and implementation of Carousel, a virtual file system and its client application. The virtual file system provides a unified interface for accessing and manipulating resources on IMAP and WebDAV servers. Carousel draws inspiration from the features and implementation of the Be File System (BFS), supporting two features not provided by most traditional file systems: multiple categorization and live queries.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Problem and Motivation

The quantity and complexity of information that computer users are required to process is an increasing problem. While the amount of accessible information has greatly increased, there is still a shortage of software that can help users manage the vast amounts of information at hand. The problem presented by this project is that computer users are unable to manage information in an efficient manner because of fundamental incompatibilities between the user's mental model and the organizational model provided by computer software. In short, people do not think about and manage information the same way that computers do.

1.1.1 Multiple Categorization and Live Queries

The primary method of organizing data on a computer is manipulating files on the file system. The file system is the component of the operating system that allows the user and his programs to save and retrieve data from the hard disk in a simple and efficient manner. This project addresses two ways in which most file systems fail to provide an adequate mapping to the user's mental model. The first problem is limited support for *multiple categorization*. That is, a file must reside in a single directory. For example, an invitation to an office Christmas party would be placed in either the 'work' folder or the 'events' folder, but not in both. However, it seems reasonable that it could belong to both categories, and therefore exist in both folders, without actually taking up twice the physical space on disk. The second idea addressed by this project is the concept of a *live query*, i.e. a folder which contains all files which match certain criteria, and is updated as new files appear that match the criteria. If implemented in a file system, these two features could significantly

reduce the complexity of information processing for many users.

1.1.2 Ubiquitous Access

Another problem faced by many users is that the information is simply not physically accessible when they need it. The widely-used Post Office Protocol (POP) for accessing email is susceptible to this problem. Since POP was designed for offline access, the messages are usually downloaded and removed from the mail server by the mail client before being read by the user. If a user accesses his email from several different computers, his messages will be spread out across systems. The Internet Message Access Protocol (IMAP) was created as a solution to this problem – it allows messages to be stored and manipulated on the server, so that they remain accessible on any computer with a network connection. However, IMAP was designed for email, not as a generic network file system. Many users could benefit from an IMAP-like solution that supports different types of data.

1.2 Goal

The goal of this project was to design and implement a virtual file system (VFS) that supports multiple categorization and live queries, and a proof-of-concept client that could access the file system. A virtual file system is a programming interface which provides the same operations as a file system, but whose purpose is simply to mediate access to other file systems [1]. The file system would support IMAP as a data source, but also provide the capability to store and retrieve files of arbitrary types. The client would connect to the file system through a network interface, and operate much like an IMAP client, with important data remaining on the server.

1.3 Expected Contribution

A user-level application will be implemented that provides the virtual file system, using a high-level language such as Java or Python. Rather than implementing IMAP support directly, a pre-existing IMAP library will be used, such as JavaMail (`http://java.sun.com/products/javamail/`) or Python's built-in IMAP library.

A proof-of-concept client will be implemented that will access the VFS through a well-defined network interface. The client will provide the ability to browse the file system, as well as create, modify, and save

[1] In this project, the term 'file system' is used loosely to refer to any system which provides the ability to read and manipulate objects resembling files and directories.

changes to files. The client will provide an interface for multiple categorization of files, and the creation of live query folders.

1.4 Overview of Chapters

This report describes the implementation of the virtual file system and client application, which are together referred to as Carousel. The second chapter describes how this project relates to the current state of the art in file systems, and reviews three projects which aimed to solve similar problems to those proposed in this project. The third chapter describes two important technical decisions that needed to be made before the implementation could move forward. The fourth chapter contains implementation notes, detailing the technical aspects of the implementation, and describing problems that were encountered and how they were solved. The fifth chapter presents the results of the implementation phase of the project. The final chapter concludes the report, evaluating the final result in light of the stated goals of the project and suggesting further work that could be pursued.

Chapter 2

Related Work

The core ideas presented in this project are not new. The increasing information management burden on users has been addressed by several academic studies and commercial products. First, this chapter examines how the features of traditional file systems compare to those proposed by this project. It describes the approach taken by BFS, an advanced file system which was the inspiration for most of this work. While BFS adequately solves the problems at the file system level, there is a requirement for a higher-level general-purpose solution. This chapter also describes three projects which have taken such an approach, and compares them to the solution presented by Carousel.

2.1 Modern File Systems

The concept of multiple categorization in a file system is not new. Most modern file systems support multiple categorization to some degree, through the ability to create a file-like object which is a pointer to another file. This concept is referred to by several different names: in Windows, a *shortcut*; in Mac OS, an *alias*; and in Unix, a *link*. There are several problems with this approach. First, the semantics vary from platform to platform — the effect of certain operations can be hard to predict. In Windows, if the target file is renamed, moved, or deleted, the shortcut is broken. However, when the user attempts to access the shortcut, the system will automatically search for the document. Unix has two types of links: a *hard link* and a *soft* (or *symbolic*) *link*. In Unix, if the target file is renamed, the link will be broken only if it is a symbolic link. A second problem is that it is usually only possible to create a link to a file on the same physical file system as the original file. That is, it is not generally possible to keep a link on the user's desktop that points to a file on a network server. Clearly, the ability to create a pointer to a file is not an adequate solution to the problem of multiple categorization.

Lack of support for multiple categorization is not the only limitation of most modern file systems. A common information management task is to search for files that match certain criteria. On the World Wide Web, the most common way to find information is to use a search engine such as Google (`http://www.google.com`). BFS, the native file system for the Be Operating System (BeOS), is unlike most file systems in that it provides direct support for complex searches (known as *queries* in BFS).

2.2 BFS

The query mechanism in BFS is based on extended file attributes. Attributes are used to store information about a file when it may not be possible or feasible to store the information in the file itself. For example, an image file might have an attribute which contains the name of the artist who produced it. In general, an attribute consists of a name (such as 'artist') and a value (such as 'Paul Klee'). In BFS, the value can be a string, a number, or raw binary data. Since attributes are important to the query facility, they are also used in some situations where the data *is* stored in the file. For example, email messages have attributes which correspond to the common email header fields: `To:`, `From:`, `Subject:`, etc., although this information is also available in the file itself.

When searching for files in BFS, the user can issue a query to the file system to find files based on the value of certain attributes. To find all emails received from a particular person, the user could issue a query to find all files where the `from` attribute is equal to that person's email addresses. A naive approach would be to examine all files in the file system and check if each one has the matching `from` attribute. However, if file attributes are *indexed*, it is possible to find a list of matching files much more quickly. An *index* is simply a data structure which allows efficient access, based on a specific attribute, to a list of files. Indexing of attributes is central to BFS and its query mechanism, and will be explored further in this paper.

When a file system provides built-in support for efficient searches, it is possible to provide support for live queries as an extension of the general query mechanism. Live queries are supported directly by BFS, and also built upon the extensive use of attributes and indices in the file system. Using user-specified attributes and live queries, BFS supports multiple categorization. For example, a user could assign a `priority` attribute to files, and create a live query containing all files where `priority` is greater than 7. The files would continue to be accessible in their original location, but would also be contained in the live query.

The design and implementation of BFS is well documented in [Giampaolo, 1999], which served as the single most important resource for this project. While the features described above can be very useful, it has become rare for a user to spend most of his time interacting with a single file system. More commonly, data resides in many different places: on a web site, as an attachment to an email on an IMAP server, or

on a network file server. In such an environment, it would be useful to provide these concepts in a higher level application. Several projects have had aims similar to this one: to provide higher level information management features within the confines of a heterogeneous data environment.

2.3 Placeless Documents

One such project was Placeless Documents [Dourish et al., 2000], a research project of the Xerox Palo Alto Research Center. With Placeless Documents, document properties are the primary means of organization, not the physical location of the document. The properties used for organization are those that are significant to the user rather than to the computer. The Placeless Documents project concentrated on three core features: uniform interaction, user-specific properties, and active properties. Uniform interaction is achieved by making document properties the primary interface for all interactions. While some properties may be available to all users, the concept of user-specific properties is also supported. That is, one user may mark a document as 'important' while another user marks the same document as 'irrelevant', and neither user is affected by the other's decision. In this manner, properties are not low-level details about a file on disk, but high-level, user-specific properties about a document as an abstract entity. Finally, Placeless Documents supports *active properties*, which are properties that carry an associated action. For example, a `backup` property can contain code which performs a backup of the contents of the document every night.

Based on these three principles, Placeless Documents provides an interface for storing, categorizing, and retrieving documents in a similar manner to the one proposed in this project. Much like Carousel, Placeless Documents does not store the contents of documents itself, but instead "integrates and unifies existing document repositories" [Dourish et al., 2000]. Each document in the system is associated with an active property called its *content provider*, which is responsible for reading and writing the contents of the document. One class of document provider might be capable of accessing an IMAP server, while another might be able to retrieve documents from the World Wide Web. A group of documents which share certain properties is known as a *collection*. Membership in a collection can be determined dynamically, in a manner similar to the concept of live queries. Placeless Documents uses what the authors call *fluid collections*, which balance the dynamic characteristic of live queries with the ability to make collections user-manipulable. A fluid collection consists of three properties: the query, the inclusion list, and the exclusion list. The query dynamically selects documents for membership in the collection. The inclusion list is a list of documents which the user has indicated should be part of the collection regardless of whether or not they match the query. The exclusion list refers to documents that should explicitly be omitted from the collection, even if they do match the query. For a given collection, any of the three properties can be empty. A collection with

empty inclusion and exclusion lists would be a fully-dynamic collection (exactly the same as a live query), while one with only an inclusion list would be the same as a directory or folder on a conventional file system.

The goals of the Placeless Documents were quite similar to those of this project. The two projects are similar in that they seek to enhance, rather than replace, traditional data and information storage. Other projects have taken this a step further, not only enhancing the user's ability to organize information, but using automated agents to perform tasks on the user's behalf.

2.4 Haystack

To address many of the same problems proposed by this project, the MIT Artificial Intelligence Laboratory has developed an application called Haystack [Huynh et al., 2002]. In June 2003, after 6 years of development, the project was released under a liberal Open Source license. Haystack is a "universal information client" which seeks to address four specific user needs. First, a user should be able to organize his information in any way he chooses, regardless of the underlying object types. Second, the system should not create any artificial distinctions based on the object type or the application that was used to create it. The system should use a homogeneous representation, and allow the user to organize information as he sees fit. Third, the system should allow the user to quickly and easily customize his view of the information based on the current task. Finally, the system should allow the user to delegate some information processing tasks to system agents.

To address the problem of unifying bodies of heterogeneous data, Haystack uses a semi-structured RDF database. RDF (Resource Description Framework) is a standardized language for describing metadata of Internet resources. It was originally developed for the Semantic Web, as a semi-structured data format to facilitate automation. Haystack uses RDF to bring structure to the user data in the same way the Semantic Web aims to bring structure to the World Wide Web.

In order to assist the user in accessing and organizing information, Haystack provides a set of agents that operate on the RDF database. For example, agents fetch email from a POP3 server, attempt to automatically categorize documents, and flag documents as being important for the user to look at. These agents can leverage information given by the user, as well as information retrieved from the Internet, to make decisions on the users behalf. Some of these agents are collaborative agents - agents that can communicate and share information with the agents of other Haystack users. In order to make sense of the data provided by the user and the various agents in the system, Haystack uses a belief layer that makes decisions about the truthfulness and trustworthiness of various components of the system.

Unlike Placeless Documents and Carousel, Haystack seeks to completely change the way people perform

their information management tasks. It is an ambitious project combining research in artificial intelligence, user interface design, and data storage and retrieval. The scope of change, as well as the computation resources required to run the program [1], make it rather impractical as a general-purpose solution for most users. An Open-Source project called Chandler seems to be designed to solve many of the same problems as Haystack, while taking a more evolutionary approach.

2.5 Chandler

Chandler is a personal information manager (PIM) being developed by the Open Source Applications Foundation. Like other PIMs, such as Microsoft Outlook, Chandler will be used to manage email, contacts, and personal scheduling. However, Chandler also aims to be a general-purpose information management tool, which can be used for any type of information that a user may want to manipulate. Chandler is in the very early development stages, so the final product may differ significantly from the current vision. The features described here are based on documents available on the OSAF web site, `http://www.osafoundation.org`.

Like Carousel, one of the goals of Chandler is to allow users to manage their information in a flexible manner. "With Chandler, users will be able to organize diverse kinds of information for their own convenience — not the computer's convenience" [Foundation, 2003b]. Chandler will provide the ability to create context-sensitive views containing many types of data, including email, instant messages, and spreadsheets. One of the key aspects of Chandler's design is the ability to gather information from disparate sources. A user will be able to search through all of his data, whether it resides on the local machine, on another user's machine, or on a server. The user will also be able to save the results of a search for later use. This feature sounds similar to live queries which was implemented in Carousel.

Most of Chandler is being developed in Python, with performance-critical code being developed in C and C++. According to one of the core developers, the design of the data repository "is probably our biggest architectural challenge" [Open Source Applications Foundation, 2003]. It is currently proposed that the data repository be built using the Resource Description Framework (RDF), due to its ability to describe data in a flexible format. Chandler is similar to Haystack in this respect. One way that Chandler differs significantly from Haystack and Carousel is that Chandler aims to be a *platform* for developing information management applications. Chandler will support modules at a variety of levels, and use well-defined Application Programming Interfaces (API), to support extensibility. Programmers will be able to develop plug-ins for Chandler (known as parcels) written in Python. Within a parcel, an end user will be able to create customized views of his data, and instruct agents to perform work on the user's behalf.

[1] The Haystack downloads page at `http://haystack.lcs.mit.edu/downloads.html` strongly recommends a Pentium 4 2GHz, 768MB RAM, and 1GB disk space

Overall, Chandler seems to have similar goals to Haystack, but much more modest. This is not surprising, as Haystack is a long-lived academic project, while OSAF's goal is to produce an end-user application in a reasonable amount of time. In the same way, the goals of Carousel are similar to Chandler's, but the timeline even shorter.

2.6 Summary

This chapter submits that traditional file systems do not provide adequate solutions to the problems presented in this project. They have limited support for multiple categorization, and little or no direct support for efficient searches or live queries. The exception to this statement is BFS, which supports all of these features. While BFS provides an excellent solution at the file system level, there is a need for a more general-purpose solution. Placeless Documents, Haystack, and Chandler are three projects which have attempted to provide similar features in a higher-level application. These projects were examined before the implementation of Carousel took place, so some inspiration was drawn from their approaches to the problems. Table 2.1 summarizes the features offered by each of the approaches described in this chapter.

Table 2.1: Comparison of Features

	User-Defined Attributes	Multiple Categorization	Efficient Searches	Live Queries	System Agents
Typical File Systems	*no*	*limited*	*no*	*no*	*no*
BFS	*yes*	*yes*	*yes*	*yes*	*no*
Placeless Documents	*yes*	*yes*	?	*yes*	*no*
Haystack	*yes*	*yes*	*yes*	*yes*	*yes*
Chandler	*yes*	*yes*	*yes*	*yes*	*yes*
Carousel	*no*	*yes*	*yes*	*yes*	*no*

Chapter 3

Implementation Decisions

Before implementation could begin on Carousel, there were two critical decisions that needed to be made. The first decision was what programming language(s) Carousel would be implemented in. Given the nature of the project — a proof-of-concept and academic exercise — it seemed logical to choose a high level language like Java or Python, rather than C or C++. The second decision was how Carousel would store files of types other than email. At worst, it would be possible to write a simple network file system from scratch. However, given the goals of the project and its relatively short timeline, this was not acceptable. Instead, several popular network protocols were examined to see if one could provide the features necessary to act as a general-purpose network file system for Carousel.

3.1 Programming Language(s)

There are many possible implementation languages for Carousel. For a performance-critical application, C or C++ might be the best choice. When performance is less important than development time and effort, a higher-level language is often more appropriate. Higher-level languages like Java offer features such as garbage collection and array bounds checking, as well as comprehensive libraries, which can save development and debugging time. Recently, scripting languages such as Perl and Python have become more popular as application programming languages, offering many of the same features as Java. Since this project is not performance-critical, the decision was made to use a high-level language for majority of the development.

Due to the complexity of the IMAP protocol, the decision was made to use an existing library rather than implement IMAP support directly. Most high-level languages, including Java, Python, and Perl, have the ability to make use of libraries written in C. In general, however, a library written in a high-level language cannot be used by any other language. Therefore, the choice of implementation language would to some

extent be determined by the availability of IMAP libraries.

The reference IMAP implementation from the University of Washington includes an IMAP library written in C known as C-Client. Since the development language would not be C, using the C-Client library would require extra effort to make the library available to the high-level language. For example, Java requires a Java Native Interface (JNI) wrapper to be written in order to use a C library. However, the JavaMail API, which is part of J2EE and is available as a standalone library, provides IMAP support with much higher level of abstraction than the C-Client library.

Based on anecdotal evidence, many developers claim that they are several times more productive in Java than in C or C++ [Foundation, 2003a]. Similarly, it has been claimed that a developer can be several times more productive in Python than in Java [Ferg, 2003]. There are a few possible explanations as to why a developer might be more productive in Python. Java is a very verbose language, whereas Python syntax is quite compact. Guido van Rossum, the creator of Python, suggests that Python's powerful built-in types are a source of the language's productivity [Venners, 2003].

These reasons were strong factors in the decision to implement Carousel in Python. In addition, Carousel was intended to run either as a desktop application, or as a server application with a web interface, and Python is more likely to be found on a typical web server than Java is. Unlike Java, Python provides IMAP support (imaplib.py) as part of its standard library. However, the level of abstraction provided by this library is much lower than JavaMail, and closer to the C-Client API.

3.2 Protocols

The primary purpose of Carousel is to allow the user to store, retrieve, and organize data on network servers. Specifically, the client should provide a feature set similar to the one provided by the IMAP protocol, but not restricted to the manipulation of email messages. Ideally, an existing protocol could be found which provides features similar to IMAP, but one which can store files of arbitrary type.

3.2.1 IMAP

The Internet Message Access Protocol (IMAP) provides a set of commands for the manipulation of email messages contained in hierarchical folders on a remote server. IMAP allows a client application to create, delete, and rename remote folders (also known as mailboxes in IMAP terminology). Individual messages are not normally delivered via IMAP, but instead through a separate delivery mechanism, such as SMTP. However, IMAP does allow new messages to be saved to a folder, for the purpose of creating drafts of incomplete messages to be sent later. In order to minimize network access, messages can be fetched in whole

or in part, and searches for messages or folders can be performed on the server, without transferring the entire contents of non-matching items. [Crispin, 1996] Since Carousel was designed with exactly the same mode of use as IMAP, that protocol would serve as the baseline by which other protocols would be compared. The challenge was to find a protocol that supported a similar mode of use, but which would allow any type of file to be stored.

3.2.2 FTP

There are several network application protocols that can be used to store and retrieve the contents of a file of arbitrary type. File Transfer Protocol (FTP) is a popular means of exchanging files between computers. The advantage of using FTP is that it is mature and well-tested, having been in use for almost 20 years. However, there are several problems with FTP. First, FTP is a somewhat complex protocol which requires two connections between a client and server. Second, FTP requires a username and password to connect to a server, and the password is sent in plain text. Finally, many network firewalls are configured to block FTP connections in any direction. HTTP, on the other hand, is a much simpler and more widely-accessible protocol.

3.2.3 HTTP

Hypertext Transfer Protocol (HTTP) is the protocol of the World Wide Web. A client (usually a web browser) requests a document by name, such as `http://www.abc.com/hello.html`, and the server responds with the contents of that document. Usually, the file is a Hypertext Markup Language (HTML) document, but the protocol itself is capable of transferring any type of file. The advantage of HTTP is that clients — web browsers like Microsoft Internet Explorer and Netscape Navigator — are virtually ubiquitous, and very few firewalls filter HTTP traffic.

The web is currently considered to be a read-only medium, due to the fact that the first popular web browser, NCSA Mosaic, was not capable of publishing to a web server. However, HTTP provides simple commands for creating and modifying files on a web server. In the mid 1990s, as web site authoring tools began to take advantage of these commands, it became clear that protocol had significant limitations in this respect. The Internet Engineering Task Force (IETF) formed the World Wide Web Distributed Authoring and Versioning (WebDAV) working group to consider these issues. [E. James Whitehead and Wiggins, 1998]

3.2.4 WebDAV

The IETF WebDAV working group came to the conclusion that "the existing capabilities of the WWW have proven inadequate to support efficient, scalable, remote editing, free of overwriting conflicts" [Slein et al., 1998]. As a result, a set of extensions to the HTTP/1.1 protocol were proposed which would provide better support for distributed authoring and versioning on the web. This standard came to be known as WebDAV, after the name of the working group. While WebDAV was primarily designed to allow interoperability between web page authoring tools, it can more generally be used as a network file system, one which operates on entire files. [Stein, 2000]

There are several advantages to using WebDAV as the main protocol for manipulating arbitrary files in Carousel. Its feature set is comparable to IMAP, and both protocols were designed with concurrent access in mind. Since WebDAV is backwards-compatible with HTTP, read-only access to a WebDAV repository is available through a standard web browser. Since network firewalls rarely block HTTP traffic, a WebDAV server should usually be accessible on a network-connected machine. Finally, there is a WebDAV library available for Python which can be used when developing the client interface.

3.3 Summary

Since Carousel is a proof-of-concept and not a performance-critical application, the decision was made to use a high-level language for development. Both Java and Python suit this purpose, but Python was chosen due to its wider availability on average network servers. For this stage of development, IMAP and WebDAV will be the primary protocols used. However, it might be useful to add support for protocols such as FTP at a later date.

With these two decisions made, development of Carousel could proceed. The next section describes the implementation stage, and discusses some problems that were encountered, and how they were solved.

Chapter 4

Implementation Notes

This chapter describes the design and implementation of Carousel. It first presents the high-level design of the system, which arose out of some initial exploratory programming. The implementation of multiple categorization, searching, and live queries is explained in depth. As the implementation progressed, several problems were encountered. Given the goals of the project, certain problems were essential, and required resolution. This chapter describes the solutions to such problems, and discusses possible solutions to some problems that were considered non-essential.

4.1 Organization

When designing Carousel, it became clear that there were two distinct components in the system. Most importantly, there is the virtual file system (VFS) component, which provides a uniform programming interface for accessing network resources. Through this interface, a program can perform typical file system operations (read a file, add a new file to a directory, etc.) without knowledge of where or how the file is stored. That is, a file stored on a web server can be manipulated in the same manner as an email on an IMAP server. Both kinds of file are identified by a Uniform Resource Locator (URL)[1]. For example, a file on a web server is identified by the URL like `http://www.gkb.com/report.html`, while an email can be identified by a URL such as `imap://joe@mail.gkb.com/INBOX/;UID=13`. The first component of a URL identifies the protocol used to access the resource. In this example, the protocols are HTTP and IMAP, respectively. Given a request to access the contents of a resource identified by its URL, the VFS component can invoke the appropriate protocol-specific routine to satisfy the request.

[1] The IMAP URL scheme is described in [Newman, 1997] while the format of URLs for the World-Wide Web is described in [Berners-Lee, 1994].

From a Carousel user's point of view, it would be cumbersome to have to remember the URL for each resource in which he is interested. Instead, he would like to refer to his favourite resources by a short, memorable names, such as simply 'Inbox'. He would like to refer to a bookmark as 'Ottawa weather report', rather than `http://www.weather.ca/weather/cities/can/Pages/CAON0512.htm`. This process of converting a name chosen by the user into a URL is a secondary but equally important responsibility of the VFS component.

The VFS component is also responsible for the creation and maintenance of indices. Much like in BFS, attributes are an integral part of Carousel. Every resource in Carousel has a set of attributes. Many attributes are associated with an index, which provides an efficient way to retrieve resources based on the value of that attribute. The VFS component is responsible for maintaining the indices, and using the indices to execute queries on behalf of the user.

The second important component in Carousel is the client of the VFS interface, or simply *the client*. The client is responsible for the user interface, translating user input into a request to the VFS component, and then presenting the results to the user. An example of user interaction with the client would be the client presenting a list of the email messages contained in the user's IMAP inbox. The user might indicate that he wishes to view the contents of the particular message by double-clicking on the message in the list. The client would translate that action into a request for the VFS component to fetch the contents of the message from the IMAP server, and the client would then display the message to the user.

4.2 VFS Component

In the Carousel VFS component, all file system objects are dichotomously divided into two classes: File and Collection. These two classes are subclasses of VNode (short for 'virtual file system node'), which provides some common functionality. For example, both Files and Collections have the attributes `name` and `url`. A File represents any file-like object, i.e. any object which can be read from or written to as a stream of bytes. Email messages on an IMAP server are represented by instance of the Email class, which is a subclass of File. On a WebDAV server, any non-collection resource is represented by an instance of DAVFile, also a subclass of File. In contrast to a File, a Collection does not contain a stream of bytes, but an ordered list of VNodes representing the contents of the Collection [2]. A Collection is much the same as a directory or folder in most file systems. Mailbox and DAVFolder are subclasses of Collection, representing an IMAP mailbox and a WebDAV collection, respectively. Most of the high-level code in Carousel relies only on the behaviour of File

[2] A Collection can be implemented as a special kind of file, one whose byte stream follows a strict structure which describes the children of the Collection. However, in a high-level application such as this one, it is simpler to consider the two types of objects to be separate and distinct.

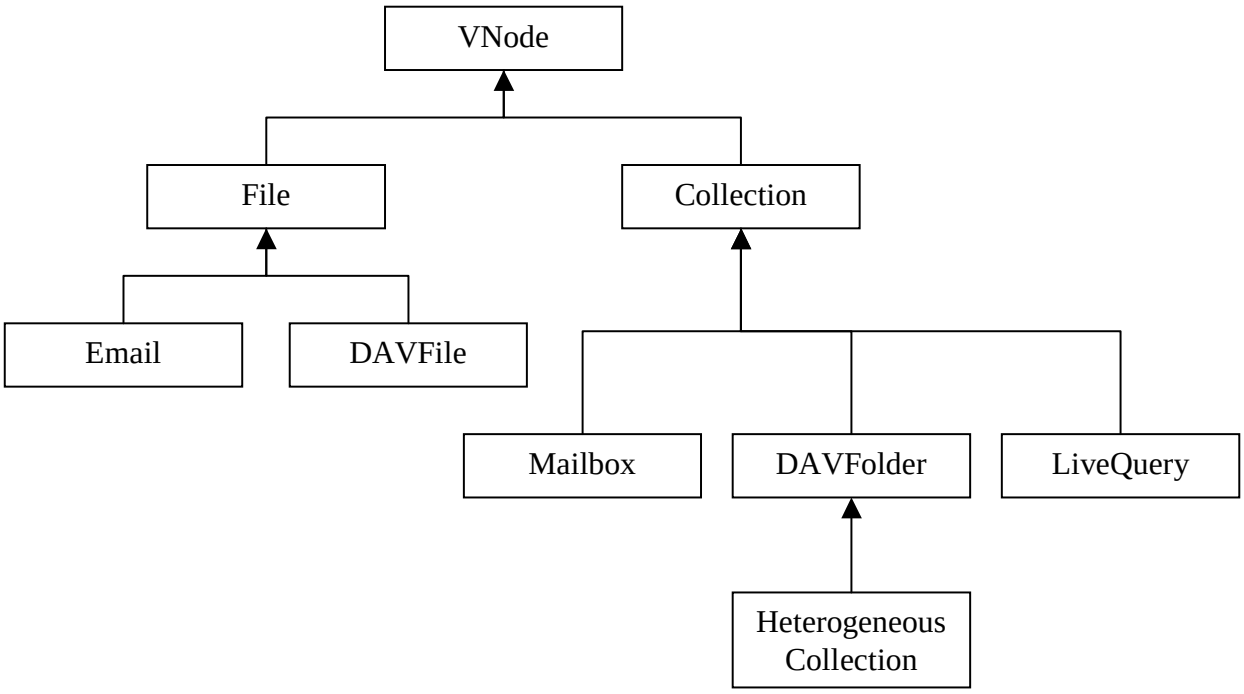

Figure 4.1: VNode class hierarchy

and Collection, and not on any specific subclass, such as Email or Mailbox. This makes it easy to support multiple categorization, and to some extent live queries, by simply implementing the Collection interface.

When the Carousel client is started, a single Collection must be designated as the *root node* (identified by the location `carousel:/`). For a file or collection to be accessible from Carousel, it must be descended from the root node. Any type of Collection could serve as the root node, such as a Mailbox or DAVFolder. However, in order to support multiple categorization, the root node is an instance of HeterogeneousCollection, which is also a subclass of Collection. The current implementation of HeterogeneousCollection is a subclass of DAVFolder, since WebDAV was chosen as the 'generic file system' of Carousel. However, HeterogeneousCollection could also have been implemented as a subclass of some other Collection which provided the same functionality as a DAVFolder. The full VNode class hierarchy is illustrated in Figure 4.1.

4.3 Multiple Categorization

Multiple categorization refers to the ability to have a file or collection contained in more than one parent collection. In Carousel, the relationship between a Collection and its children is one-way. Given a Collection, its list of children can easily be determined, but the parent Collection of any given node is not necessarily well-defined. So multiple categorization is achieved by simply having two or more Collections which contain

common children.

The simplest kind of Collection supporting multiple categorization is HeterogeneousCollection. A HeterogeneousCollection allows children to be added to the collection through the `addchild` method. The root node in Carousel is usually a HeterogeneousCollection. A typical root node might contain a Mailbox named 'Inbox' corresponding to `imap://fred@example.com/INBOX`, an Email named 'To-do' corresponding to `imap://fred@example.com/INBOX/;UID=117`, and a DAVFolder named 'Drop Box' corresponding to `http://example.com/~fred`. The Email is both a child of the root node and a child of the Mailbox named 'Inbox', and can be accessed from either location.

Problems

There are, however, a few problems introduced by the implementation of multiple categorization. When a node has more than one parent, the semantics of the 'delete' operation are unclear, especially since the multiple categorization is only simulated. Specifically, if a file or collection is deleted from its parent, should the item be deleted from the underlying file system (and thus removed from all collections that contained it), or should it simply be removed from the collection in which the operation was performed?

In the current implementation, the effect of a delete operation is well-defined, but it varies depending on the circumstances. If a resource is deleted from within its 'true' parent collection — such as an Email in a Mailbox, or a resource added to a HeterogeneousCollection using the 'new' operation — it will be deleted from the underlying file system. Any links to the resource (i.e. resources added to a HeterogeneousCollection using the 'add' operation) will continue to exist, but will be no longer valid. However, if the resource is deleted from a collection which is only linking to the resource, only the link is affected: it is removed from the collection. The problem with such an approach is that the distinction between the two cases may not be very clear to the user, but the result of the operation is vastly different.

Perhaps there are really two distinct operations: *delete*, which deletes the file from the underlying file system, and *remove*, which removes the item from the collection without deleting it. If this is the case, it complicates the implementation of the collections which correspond directly to a file system collection, such as Mailbox and DAVFolder. These types of collection do not maintain a list of their children, but instead retrieve the children from the underlying file system collection. In order to support the 'remove' operation, they would have to maintain a list of children which are specifically excluded from the collection. In addition, to support the 'delete' operation, collections which *do not* correspond to a physical collection must have a means of determining the physical parent of a given item, so that the item can be deleted from the underlying file system. A simple implementation would be for each node to keep a pointer to its canonical parent, a

collection which corresponds directly to a collection on a file system.

Unity

An extension to the idea of heterogeneous collections that was not implemented in Carousel is the concept of unity. A union is a collection that consists of the union of the children of two or more collections. A useful example would be a collection which is the union of an IMAP Mailbox and a DAVFolder which contains important notes and reminders. When viewing the collection, the notes would be mixed in with the emails, and all of the items could be sorted based on the date.

Implementing unity in collections also presents a small problem: if a collection named 'Alpha' is a union of two collections 'Beta' and 'Gamma', and Beta and Gamma both contain a child named 'delta', which one does the name 'Alpha/delta' refer to? One answer is to say that it is not well-defined, that it may refer to 'Beta/delta' or 'Gamma/delta'. This could be very confusing, and potentially disastrous, if the wrong file were to be modified or deleted. A better answer is that the union collection Alpha detects the duplicate name, and locally renames the two items to 'delta (Beta)' and 'delta (Gamma)', respectively, eliminating the collision. However, this does lead to some unexpected behaviour, in that if Beta contained an item named 'delta', and Alpha is the union of Beta and some other collections, then one would expect Alpha to also contain an item name 'delta'. The consequences in this case are only slightly inconvenient, rather than confusing.

4.4 Searching

Searching (or *querying*) is the act of looking for files which match certain criteria. The list of search results is the answer to a question like "what files have the name 'report.pdf' and were modified since yesterday?". The request "find me all files such that..." is implicit, so the previous example can be shortened to "name is 'report.pdf' and modified since yesterday". This is an example of a *query*, although in Carousel a query is more formally defined. A query is a simply a logical statement which is either true or false for any given node in the file system. The simplest possible query consists of an attribute name, an operator, and a value. For example: `name == "report.pdf"`. The attribute name is `name`, the operator is `==` (equals) and the value is `"report.pdf"`. It is possible to form more complex queries by using the logical operators `and` and `or`, and by using parentheses specify the precedence of those operators. The first example above could be written as: `name == "report.pdf" and modification_date >= 1074769207` [3]. It is not expected that the user

[3] This example uses the POSIX method of representing dates, as seconds elapsed since "the Epoch" (00:00:00 UTC, January 1, 1970)

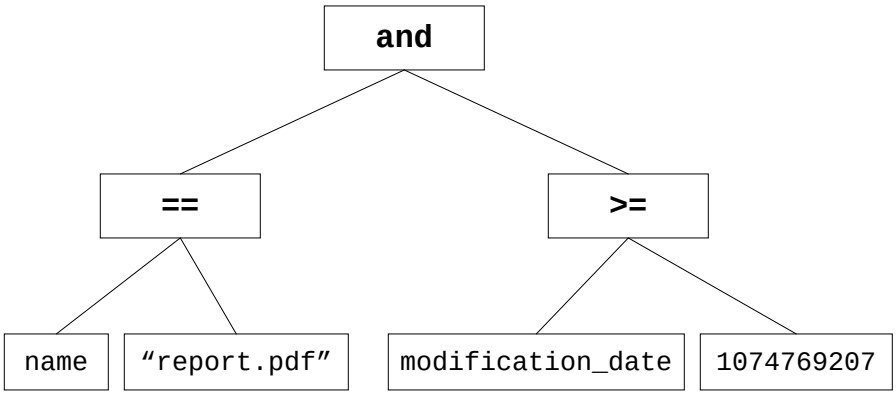

Figure 4.2: Parse tree for the expression `name == "report.pdf" and modification_date >= 1074769207`

would perform searches by writing such queries by hand. Instead, there should be a graphical interface for creating queries, which would be responsible for converting a more user-friendly syntax to the one described here. However, due to time constraints, such an interface was not implemented in Carousel; the user must create a syntactically-correct query manually.

Once the query has been constructed, it must be parsed and verified by the system. Python includes a module in its standard library that provides access to the internal Python parser. The Carousel query grammar is a subset of the Python grammar (i.e. every valid Carousel query is also a valid Python expression), so this module is used to parse the query. If the query is valid, the resulting *parse tree* is obtained. A parse tree is a graph which represents the semantic interpretation of the query. Figure 4.2 shows the parse tree for the query given above. From the parse tree, the system attempts to determine the most efficient way of executing the query.

The general strategy for executing a query is to recursively descend the parse tree and evaluate each internal node. Each internal node is either an `and`, `or`, or *comparison* (such as `==`, `>=`, `>`, etc.) node. If possible, a comparison node is evaluated by using an index. However, just because an index is being used does not mean that the search is being performed efficiently. For example, a query such as:

```
size >= 1 and name == "foo.txt"
```

could be implemented in one of two ways. Clearly using the `size` index is not very beneficial, because almost all files will have a size of at least 1, and we will have to check if each of these files has the name foo.txt. In this case, there is hardly any improvement, if any, over the linear-time exhaustive search method. However, we would expect that a much, much smaller number of files would be named foo.txt. The `size` index could be used to fetch the list of such files in O(log n) time, and each one could be checked for `size >= 1`. This example illustrates the best-case and worst-case scenarios, but in practice, the decision of which index (or indices) to use is more complicated.

It is only necessary to choose which index to use when the node being evaluated is an `and` node. The result of an `or` node is the union of the results of all of its subtrees. If its subtrees are comparison nodes, each one could be evaluated using an index, and the results combined in linear time. However, the result of an `and` node is the *intersection* of the results of all its subtrees. The most efficient way to perform this operation is evaluate the subtree which results in the smallest result set, then evaluate each of the elements in this set against the conditions of the other subtrees. Unfortunately, there is no way of knowing ahead of time which subtree will have the smallest result set. The strategy used by Carousel is to estimate the size of the result set for each subtree, evaluate the subtree with the minimum estimate, and then compare each item in the result set against all of the other subtrees.

4.5 Live Queries

Since Carousel relies on indices for performing queries, it is important that the indices are accurate. In a conventional file system, indices can be kept up to date by examining each file as it is created, and inserting it into the appropriate indices. In Carousel, it is not so straightforward. Carousel is dependent on the mechanism of the underlying file system or protocol, such as WebDAV and IMAP. IMAP does not provide immediate notification when a new email arrives, but the command set does allow a client to easily determine if the contents of a mailbox have changed recently. It is possible (and acceptable) to poll each mailbox at a given interval to determine whether its contents have changed. WebDAV does not provide any such facility — the only way to determine whether files have been added or deleted is to iterate over the entire file system. This is a costly operation that should not be performed frequently. This poses a significant problem for the implementation of live queries. By nature, a live query is expected to be updated almost immediately when new files appear that match the criteria. It would be acceptable if live queries were only updated once every few minutes, since most email client poll for new mail at a similar frequency. However, polling a large WebDAV file system this frequently is not acceptable.

Certain types of v-node are guaranteed not to be modified outside of Carousel, such as a HeterogeneousCollection. In such cases, the v-node can provide notification as soon as it is modified, and the proper actions can be taken in the user interface. If the contents of a HeterogeneousCollection change while the collection being displayed, the display should be updated in accordance with the changes. V-node types that may be modified outside of Carousel, such as an IMAP Mailbox, are said to be *dynamic*. Every node has an attribute named `isDynamic` which contains a boolean value.

A consequence of this approach is that the burden of polling the node for changes is shifted from the VFS component to higher-level code. This is advantageous because it is not necessary to monitor all dynamic

nodes in the system. Higher-level code can decide which nodes (if any) need to be monitored. There are several cases to consider when deciding whether any dynamic nodes need to be monitored. First, if a dynamic collection is currently being displayed, then it should be monitored. If any children are added to or removed from the collection, the display should be updated. Second, if a live query is being displayed which could possibly contain one or more dynamic nodes (or any of their children), then those nodes should be monitored. However, this is really just a special case of the first one: a live query that could possibly contain dynamic nodes is itself dynamic. Monitoring the child nodes is implemented in the `changed` method of the Query class. Finally, when a new query is performed, any dynamic nodes in the system should be updated and matched against the query. Finding all dynamic nodes in the system is a cheap operation, because the list can be retrieved directly from the `isDynamic` index.

In the Carousel client, there is a timer which interrupts the main thread every 30 seconds to perform this node monitoring work. If the current node being displayed is dynamic, the `changed` method of the node is called to determine whether the node has changed. If it has changed, the entire list of children is fetched from the node. This operation is no more expensive than fetching the list of children for the first time. In the case of a live query, it is actually significantly less expensive. Each index is associated with a list of live queries to which the index is pertinent. Each index that is used to evaluate a query is tagged with a pointer to the LiveQuery object. When a new file is created that might match the query, it must first be inserted into the appropriate indices. As each index is updated, every live query that tagged the index is notified that a new node is being added to that index. Live queries are implemented as a subclass of Collection, so the contents of a live query are obtained by calling its `getchildren` method. The first time the method is called, the full query is executed. Thereafter, any new node that matches the query is simply added to the list of children.

A major disadvantage to using the main thread for node monitoring is that the user interface can become unresponsive for short periods of time. Checking if an IMAP mailbox has changed requires a round trip to the server, which can take a few seconds. During this time, the interface is completely locked up. A solution to this problem would be to do dedicate a background thread to node monitoring. This way, the main thread could continue to process input from the user while dynamic nodes are being polled for changes. However, making a program thread-safe is a non-trivial task. Since the client is only intended to be a proof-of-concept, the decision was made to keep it single-threaded, even though it results in poor performance of the user interface.

4.6 Summary

For the most part, the implementation of Carousel proceeded as expected. The design of the VFS component made the implementation of multiple categorization relatively simple. The current implementation *does* pose a problem with the semantics of the delete operation, but further thought is required to provide a proper solution. The implementation of searching was more complicated than expected, but no serious problems were encountered. The most significant problem encountered was the fact that the virtual file system cannot know precisely when a file is created, which causes a problem for live queries. This problem was solved by careful and constrained use of polling. Generally, the design of Carousel was based on BFS, as described in [Giampaolo, 1999]. However, the problems encountered in the Carousel implementation do not apply to BFS. First, since BFS only supports multiple categorization in the context of live queries, there is no ambiguity in the semantics of the 'delete' operation. Second, since BFS is a full-fledged file system, it knows precisely when a file is created or modified. Overall, Carousel provides acceptable solutions to these two problems. The following chapter evaluates the success of Carousel as a whole, and reveals some of its limitations.

Chapter 5

Results

This chapter shows the final outcome of the implementation phase of the project. The Carousel user interface is presented, showing how Carousel supports multiple categorization and live queries. However, the Carousel client is only intended as a proof-of-concept. The most interesting aspect of the implementation was the VFS component. This chapter uses experimental data to demonstrate that the search techniques used by Carousel are significantly faster than the naive method of testing every file in the system.

The main window of the Carousel client consists of two panes (see Figure 5.1). At any given time, the client is either displaying a collection or a file. When displaying a collection, all children of the collection which are themselves collections (identified by a folder icon) are displayed in the left pane. Non-collection children are displayed in a multi-column list in the right pane. Each column in the list corresponds to an attribute of a certain file type. For example, when displaying the contents of an IMAP inbox, the columns contains the `From`, `Date`, and `Subject` attributes of each message. When displaying a non-collection resource, the contents of the left pane are unchanged, and the right pane contains a text control with the contents of the resource.

When the Carousel client is first started, the root node (a collection) is displayed. The location of the currently displayed node is shown by the 'Location' field at the top of the window. All elements in Carousel are relative to the root node, identified by the location `carousel:/`. The user can display the contents of any child resource by double-clicking on the name of the resource. There are two other means of navigating in Carousel. Much like a web browser, items are added to a stack as they are visited, and the user can return to the previous item in the stack using the 'Back' button, located to the left of the 'Location' field. Alternatively, the user may enter a location directly into the 'Location' field to display a resource.

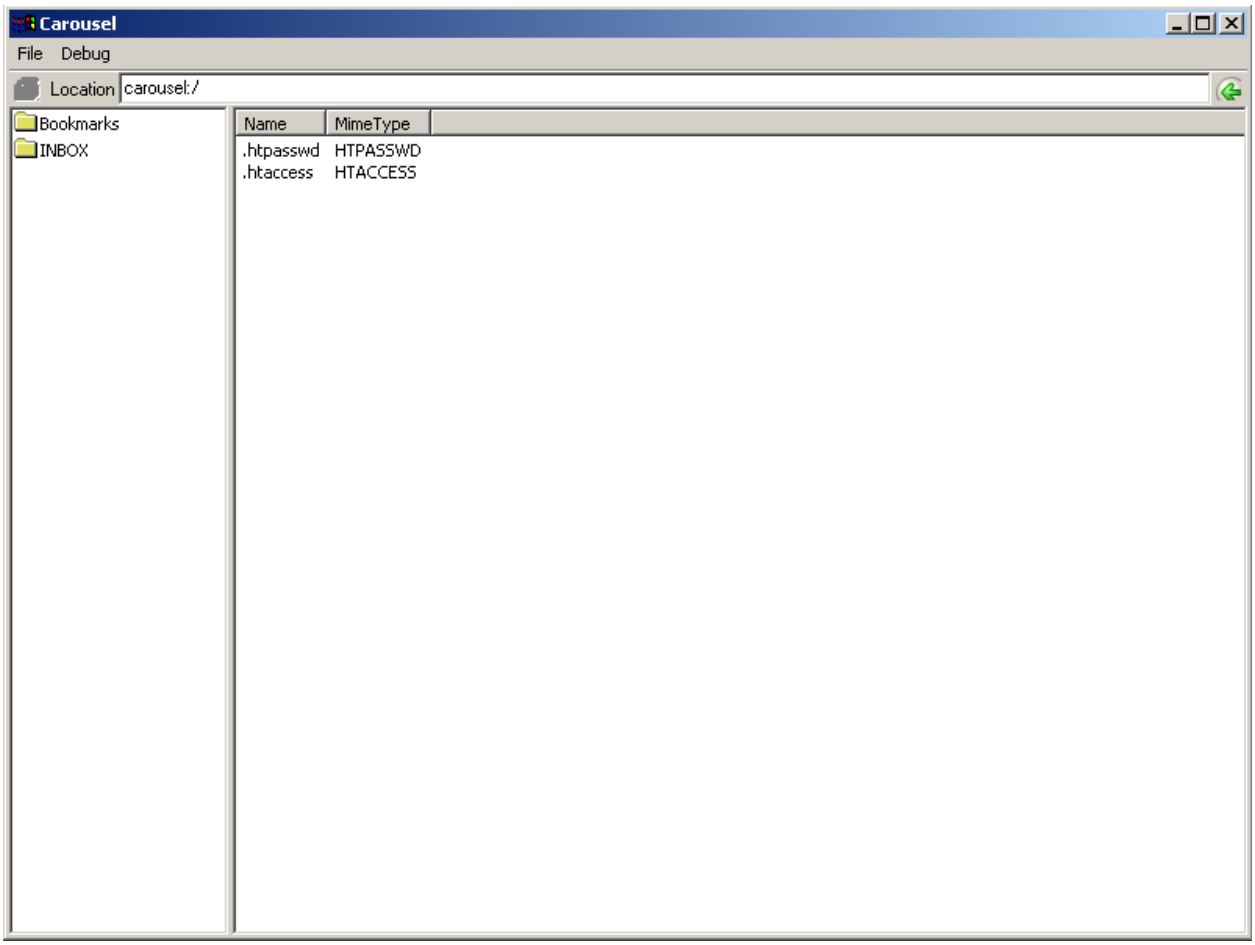

Figure 5.1: The main window of the Carousel client, displaying the root node

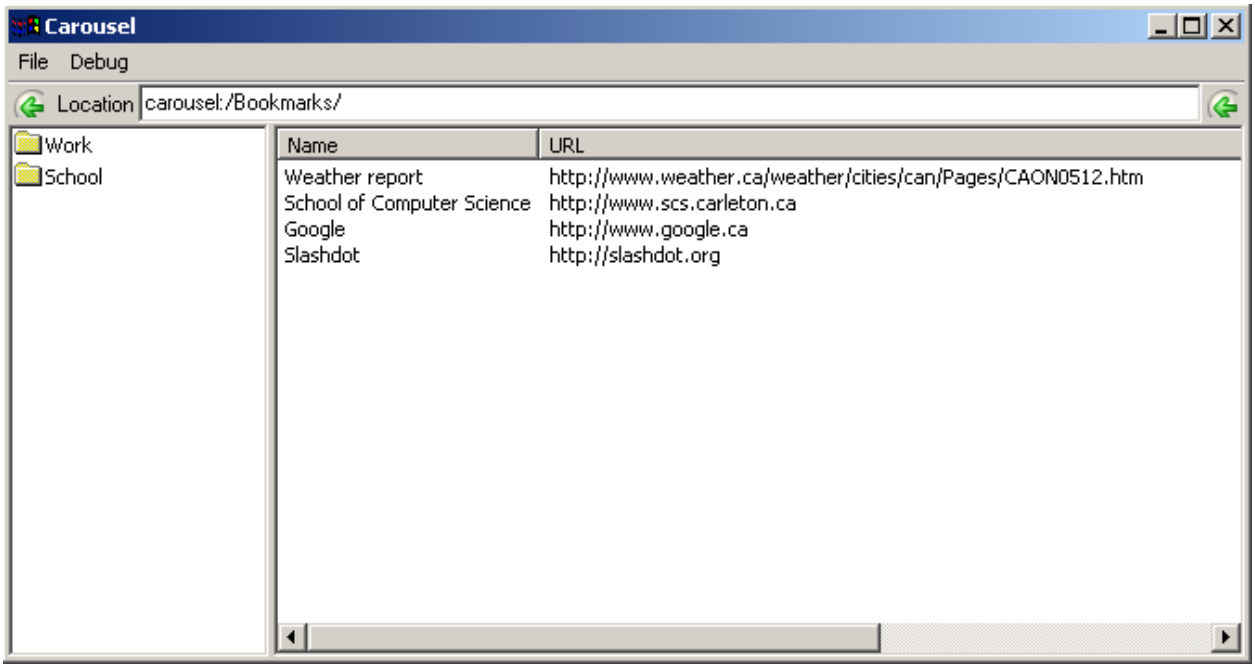

Figure 5.2: A Carousel collection containing web site bookmarks

5.1 Multiple Categorization

Multiple categorization is supported in two ways in Carousel. One method of multiple categorization is through Live Queries, discussed below. The second method is to explicitly add a resource to another collection. A user can add an item to a HeterogeneousCollection by right-clicking on the name of the collection (or, if the collection is currently being displayed, on any white space in the window), selecting the 'Add' submenu, and selecting 'File' or 'Collection'. The user is prompted for a name by which to refer to the resource in this collection, and for the URL of the resource. For example, a file could be named 'Notes from Jim' and be linked to the URL `imap://joe@mail.gkb.com/INBOX/;UID=119`. This is a long URL for the user to type — he could easily make a mistake. For resources which are already accessible from Carousel, the client provides an easier way to perform multiple categorization. First, the user navigates to the resource he wishes to add to another collection (in this case, it might be `carousel:/INBOX/;UID=119`). If the user right-clicks on the item and selects 'Copy URL' from the context menu, the URL for the resource will be copied to the system clipboard. When the user adds the item to the collection and is prompted for the URL, he can simply paste the URL into the text box [1].

Carousel is not limited to linking to resources on servers that are owned or controlled by the user. Figure 5.2 shows how a collection named 'Bookmarks' could be created to hold bookmarks to the user's favourite

[1] On Windows this can be accomplished by pressing Ctrl-V, or by right-clicking in the text field and selecting 'Paste' from the context menu

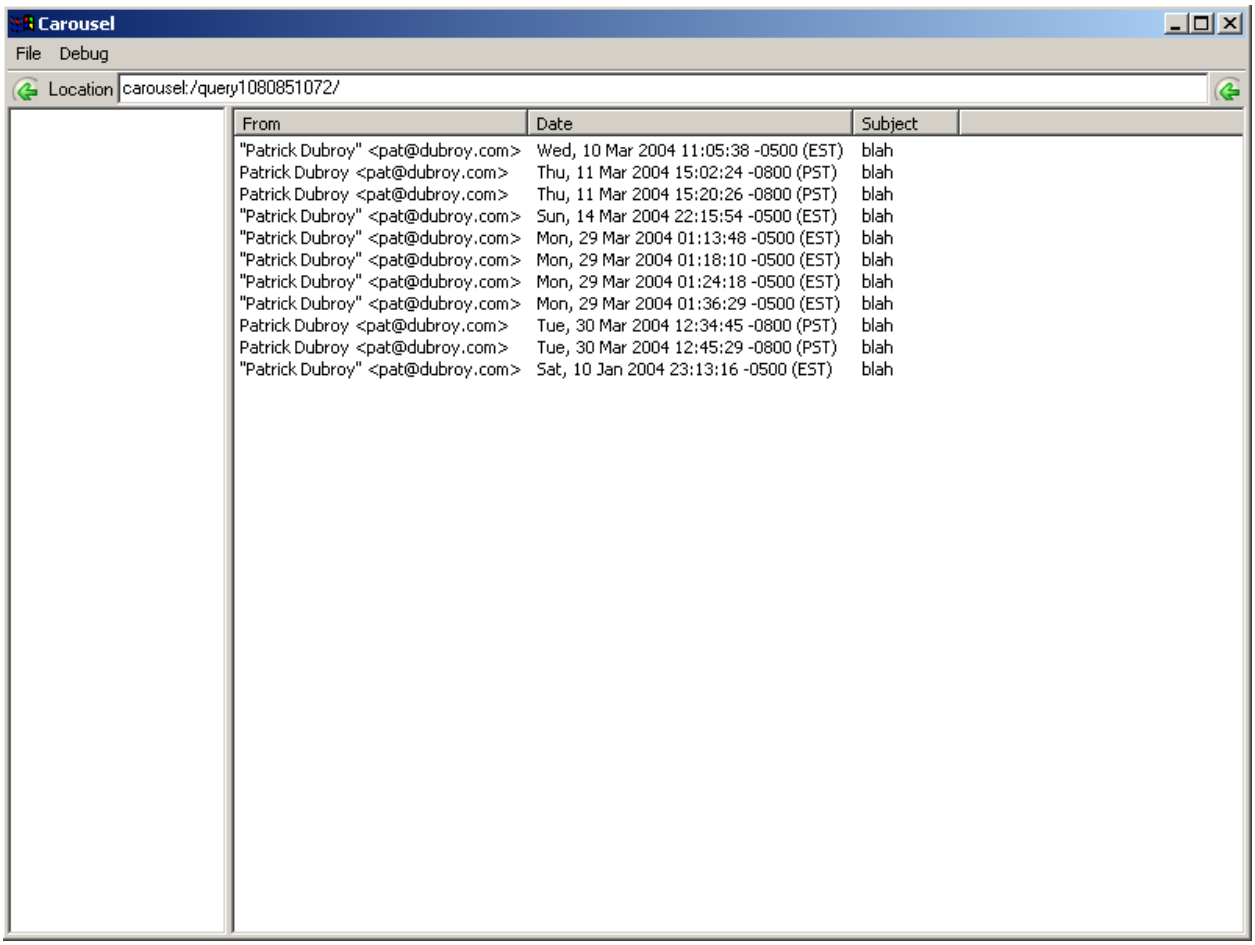

Figure 5.3: A live query containing all files where `Subject == "blah"`

web sites [2]. This is a good example of how Carousel enables the user to interact with all his data in a uniform way — a collection of bookmarks is treated like any other collection. In fact, any data that can be represented as files and collections can be managed from within Carousel. Section 6.1 suggests some possible further work in this area.

5.2 Live Queries

In addition to multiple categorization, the other main feature of Carousel is the ability to perform live queries. In fact, live queries are also a type of multiple categorization, in that a resource accessible through a live query is also accessible through some other collection.

Live queries are created in Carousel by performing a search. A global search can be performed from the

[2] In the current version of the Carousel client, this is not particularly useful, since only the raw HTML source of the web page can be displayed.

'File' menu. This searches all files known to Carousel. When a search is performed, it will be displayed as a collection in the Carousel window. The set of results of any search is a live query: while it is displayed, if new resources appear that match the search terms, they will be appear in the search results. For example, Figure 5.3 shows the results of the query `Subject == "blah"`. As long as this query is being displayed, all dynamic nodes in the system will be monitored for changes. When a new email arrives with a matching `Subject:` header, it is immediately added to the results of the live query.

The query is executed only once, when the search is first performed. As new files appear that match the query, they are simply added to the list. This is an important, because even though Carousel uses techniques to search as efficiently as possible, certain queries can still take a noticeable amount of time to execute. A small pause is not too bad if it only occurs the first time the search is performed, but if it were to occur every time a new file was added to the system, the system would be virtually unusable.

The functionality provided by live queries could be particularly useful in dealing with large amounts of email. A user might have more than one email address that is delivered to the same account. A live query could be used to provide a separate view of each account. Another use of live queries could be to organize messages from many mailing lists. Most mailing lists contain a special tag in the subject line which identifies the message as being from the list, e.g. '[Carousel-devel]'. A live query could be used to keep the messages from a particular mailing list separate. A developer might have a live query containing his to-do list, and could mark bug reports submitted to the mailing list as 'to-do', meaning they would appear in his to-do list alongside other notes and emails.

While the Carousel client is the most tangible product of this project, it is relatively incomplete. However, the goal of the project was not to produce a fully-featured client — the main focus was the implementation of the virtual file system. This section has demonstrated that Carousel VFS provides support for multiple categorization and live queries. The next section discusses the performance of the live query implementation, and shows how it is significantly better than performing static searches.

5.3 Performance

This section contains experimental data which demonstrates that the search techniques used in Carousel are significantly faster than the brute-force method. It also shows that Carousel is capable of evaluating queries in an manner which minimizes the elapsed search time. All tests were performed on the same hardware and in the same execution environment. In all cases, Carousel was invoked with the command line options: `-XbmQuery`, standing for 'benchmark query performance', and `-XnoNodeMonitor`, which disables the timer which interrupts the main thread to perform node monitoring. The root node was an IMAP

mailbox containing 500 messages in the INBOX, and three other folders containing 1000 messages each. In all three tables, Query 1, Query 2, Query 3 refer to the queries `name == "foobar"` (0 matches), `Subject == "blah13"` (1 match in 500 items, 4 matches in 3500 items), and `From == "pat@dubroy.com"` (all items match), respectively.

Table 5.1 contains the results of performing brute-force searches of 3500 items. The searches were performed by iterating over the list of all nodes in the system, and checking if each node matches the query. In this test, Carousel was invoked with the extra command-line argument `-XnoIndices`, which disables the use of indices for searches. The full command line for the test was:

```
python carousel.py -XbmQuery -XnoNodeMonitor -XnoIndices imap://m3083602@mail.dubroy.
com/INBOX
```

Table 5.1: Time To Perform Search Without Indices - 3500 Items

	Execution Time (s)		
Trial #	Query 1	Query 2	Query 3
1	1.124436	1.162670	1.191308
2	1.126441	1.168979	1.179119
3	1.133354	1.167958	1.176590
4	1.124805	1.169128	1.191021
5	1.131907	1.187020	1.181441
Avg	1.128189	1.171151	1.183896

The results of Table 5.1 show very little difference in execution time between the three queries. In all three cases, every item in the system must be compared against the query. Thus, the execution time should scale in direct proportion to the number of items in the system.

Table 5.2 contains the results of searching through only 500 items using Carousel's most efficient search techniques, including the use of indices whenever possible. These tests were performed in order to see how the Carousel's search performance scaled in relation to the number of items being searched. The full command line for the test was:

```
python carousel.py -XbmQuery -XnoNodeMonitor imap://m3083602@mail.dubroy.com/INBOX
```

The results of Table 5.2 show that searching through only 500 items using Carousel's most efficient search techniques is almost 4 orders of magnitude faster than the brute-force technique. Even if the speed of the search scaled in proportion to the number of items in the system, we would expect that searching through 3500 items would still yield much faster execution times than the brute-force technique.

In the final test, all three queries were again executed using the most efficient search techniques, against a list of 3500 items. Three additional queries were also performed. Query 4, `From == "pat@dubroy.com" and`

Table 5.2: Time To Perform Search Using Indices - 500 Items

	Execution Time (s)		
Trial #	Query 1	Query 2	Query 3
1	0.000237	0.000191	0.000232
2	0.000230	0.000196	0.000232
3	0.000230	0.000193	0.000236
4	0.000232	0.000195	0.000234
5	0.000232	0.000215	0.000239
Avg	0.000232	0.000198	0.000234

`Subject == "blah13"`, would test that Carousel would correctly decide to evaluate `Subject == "blah13"` first, because it would result in only four matches while the other half of the query would result in 3500 matches. Query 5, `From == "pat@dubroy.com" and Subject <= "blah100"`, is used for comparison purposes. Due to Carousel's unsophisticated estimation algorithm, this query would actually result in `From == "pat@dubroy.com"` being executed first. Finally, Query 6, `From == "pat@dubroy.com" or Subject == "blah13"`, is used to verify that the time to execute an `or` query is approximately equal to the sum of the times to execute each of the children. The full command line used for the tests in Table 5.3 was:

```
python carousel -XbmQuery -XnoNodeMonitor imap://m3083602@mail.dubroy.com/INBOX
```

Table 5.3: Time To Perform Search Using Indices - 3500 Items

	Execution Time (s)					
Trial #	Query 1	Query 2	Query 3	Query 4	Query 5	Query 6
1	0.000234	0.000194	0.000610	0.002705	1.222425	0.001224
2	0.000233	0.000197	0.000606	0.002755	1.213615	0.001184
3	0.000234	0.000196	0.000613	0.002698	1.227134	0.001169
4	0.000235	0.000199	0.000708	0.002719	1.216647	0.001216
5	0.000236	0.000199	0.000619	0.002624	1.220718	0.001211
Avg	0.000234	0.000197	0.000631	0.002700	1.220107	0.001201

The results of Query 1 and Query 2 in Table 5.3 demonstrate that, for simple queries, the execution time in Carousel does not depend on the number of items in the system. This is evident from the fact the time to search through 3500 items is very close to the time to search through 500 items, as given in Table 5.2. For these two queries, the list of results is exactly equal to the list of items stored under a specific key in the index. Therefore there is no need to iterate over the list of items — it can be directly returned as the result of the query. We would instead expect the execution time of simple queries like these to vary in proportion to the number of keys in the index.

The results of Query 3 seem to show that the execution time does vary in proportion to the number of

items returned from the search. Compared to Table 5.2, Queries 1 and 2 return only 3 more items in this test. However, Query 3 returns 3000 more items, or 7 times the number returned by the same query when performed against 500 items. This explains the fact that the execution time is about 3x greater. However, we still note that for all three queries, the method used by Carousel is almost 4 orders of magnitude faster than the brute-force method.

A more complex query, `From == "pat@dubroy.com" and Subject == "blah13"`, is performed with Query 4. The results of this query, when compared to the results of Query 5 (`From == "pat@dubroy.com" and Subject <= "blah100"`), demonstrate that Carousel will correctly choose to execute the subquery that has fewer results. If `From == "pat@dubroy.com"` were executed first, as in Query 5, Carousel would essentially be reduced to the brute-force search technique, since that subquery matches all 3500 items. In fact, we see that the time to execute Query 5 is quite similar to the brute-force search times in Table 5.1. The decision to execute the subquery `From == "pat@dubroy.com"` first in Query 5 is not the best decision, but it is the 'correct' one, according to Carousel's simple estimation algorithm.

We would expect that the time to execute Query 6 (`From == "pat@dubroy.com" or Subject == "blah13"`), would be approximately equal to the sum of the times to execute the subqueries, since the result of an `or` query is the union of the results of its subqueries. Indeed the results in Table 5.3 demonstrate that this is true in Carousel. The time to execute Query 6 is equal to the sum of the times to execute queries 2 and 3, plus a factor of about 1.5.

The results presented in this section prove that, for some queries, Carousel can execute a search using its indices orders of magnitude faster than with a brute-force method of testing every file in the system. Carousel can perform a search of thousands of files in well under a second of wall clock time, and in some cases, less than a millisecond. Furthermore, it has been shown that the execution time of certain searches is not even dependent on the number of files in the system, suggesting that the techniques might scale to tens or hundreds of thousands of files.

5.4 Limitations

Due to the short timeline for completion of this project, some aspects of the system could not be addressed. The most significant limitation of the system is that the indices are not permanently stored anywhere. That is, each time the client is run, the full list of files in the system must be examined in order to create the indices. This is performed by selecting 'Build Indices' from the 'Debug' menu in the client. The final chapter of the report suggests some possible solutions to the problem of permanent storage of the indices.

Another limitation of Carousel is in the query grammar. Currently, only the `and` and `or` operators can be

used to combine comparisons to form more complex queries, and the *not equal* (`!=`) operator is not supported for comparisons. Ideally, Carousel would provide support for `!=` as well as for `not`, logical negation operator. An example of logical negation is `not (name == "blah" or name == "foo")`. Also, it would be useful to support wildcard characters (or full regular expressions) for string matching.

5.5 Summary

This chapter has described the support of multiple categorization and live queries in Carousel. Some examples have been given of how these features allow users to organize and retrieve data in more intuitive ways. In order to be truly useful, Carousel must provide these features without sacrificing performance. If the system is not responsive, it will be unattractive to users. The chapter has shown that Carousel can perform searches using indices much faster than through brute-force search methods. It has also proven that the search techniques scale to thousands of files, at least.

Chapter 6

Conclusion

The impetus for this project was the belief that the organizational model provided by most computer software does not match the user's mental model. This report identified two particular features that could help reduce the problem: multiple categorization and live queries. As the project progressed, it became clear that supporting live queries was closely related to the ability to perform efficient searches, which became a secondary goal of the project. The template for much of this work was the implementation of the Be File System, described in [Giampaolo, 1999].

The goal of this project was to produce a virtual file system that supported multiple categorization and live queries, and a proof-of-concept client. The file system would be able to access IMAP repositories, but also allow other types of files to be stored and manipulated. The client would interface with the VFS, and allow the user to perform multiple categorization and create live queries, as well as create modify files.

Based on the stated goals of the project, the outcome was a success. The previous chapter demonstrated that the client works as described, and through experimental data, proved that the search techniques used by Carousel can significantly outperform other approaches. However, for Carousel to provide a real solution to the problems inherent in managing large amounts of data, many more aspects need to be addressed.

6.1 Further Work

While the client interface is functional, it needs refinement. In particular, the distinction between the 'add' and 'new' operations and the 'remove' and 'delete' operations might be quite confusing to a user. If necessary, some of the flexibility of the virtual file system could be constrained so as to simplify this aspect of the user interface.

In general, there are many features missing from the user interface that would be required for Carousel to

be suitable for everyday use. For example, more sophisticated display and editing of files would be beneficial. Currently, the entire body of an email is displayed, including all of the headers. Ideally, emails would be displayed in an abbreviated form as in most email clients. Also, the ability to send email would be required before Carousel would even be useful as a replacement for a traditional email client.

Overall, the Carousel virtual file system is a solid base upon which to build a more fully-featured client. As mentioned at the end of Section 5, it would be nice to provide a more sophisticated query facility. Most importantly, Carousel needs some means of permanently storing its indices. They could be stored on the local hard drive, but that would mean that each instance of the Carousel client on separate computers would maintain its own copy of the indices. If one client hadn't connected in quite a while, the user would have to wait a long time for the indices to be updated. Another alternative is to maintain the indices on a network server. This would be the most efficient choice overall, although search performance would be degraded as every search would require a round trip to the server.

Another avenue for further work is in building more powerful abstractions on top of the essential concepts of Carousel. When multiple categorization and live queries are applied to more diverse types of data, many interesting possibility become evident. Consider the idea of storing a contact list in Carousel. A contact might be represented by a file with attributes such as `name`, `address`, `email`. Contacts could be in categories such as 'work', 'school', and 'friends'. If Carousel could recognize that the `From:` header of an email refers to a contact, this categorization could be leveraged to divide a user's email into the same groups.

There are many other ways in which the core concepts of Carousel could be built upon to provide new capabilities. If features such as the ones presented in this section were implemented, Carousel might represent a significant step towards easing the information management burden of many computer users.

Appendix A

Source Code

The Carousel source code is available on the CD accompanying this report.

A.1 Prerequisites

Carousel was developed and tested on Python 2.3 on Windows 2000. Python can be downloaded from `http://www.python.org`. Carousel also requires the wxPython library, which is available from `http://www.wxpython.org`. Carousel was developed with version 2.4 of wxPython, but may work with more recent versions.

A.2 Installation

Carousel does not require any installation per se. Files contained in the main source code directory must be available to the Python interpreter. The simplest approach is to leave all these files in the same directory, and run the Python interpreter from that directory.

A.3 Running

Carousel is run by invoking the Python interpreter on `carousel.py`. Carousel requires at least one command line argument, which is the URL of the root node. For full functionality, this should be a WebDAV folder. For example:

```
> python carousel.py http://www.gkb.com/dav/
```

Carousel is not yet able to perform HTTP authentication, so a WebDAV folder must not require a username and password to access it. To connect to an IMAP server, the username must be specified as part of the URL, e.g. `imap://joe@mail.gkb.com/INBOX`.

A.4 Copyright and License

`moddavlib.py` is Copyright (C) 1998-2000 Guido van Rossum and Copyright (c) 2004 Patrick Dubroy. It is a modified version of `davlib.py`, a Python module written by Greg Stein and available from `http://www.lyra.org/greg/python/davlib.py`. `moddavlib.py` is licensed under the same terms as `davlib.py`.

References

[Berners-Lee, 1994] Berners-Lee, T. (1994). Universal resource identifiers in WWW: A unifying syntax for the expression of names and addresses of objects on the network as used in the World-Wide Web. `http://www.faqs.org/rfcs/rfc1630.html`. Cited 21 March 2004.

[Crispin, 1996] Crispin, M. (1996). Internet Message Access Protocol - version 4rev1. `http://www.ietf.org/rfc/rfc2060.txt`. Cited 29 January 2004.

[Dourish et al., 2000] Dourish, P., Edwards, W. K., LaMarca, A., Lamping, J., Petersen, K., Salisbury, M., Terry, D. B., and Thornton, J. (2000). Extending document management systems with user-specific active properties. *ACM Transactions Information Systems*, 18(2):140–170.

[E. James Whitehead and Wiggins, 1998] E. James Whitehead, J. and Wiggins, M. (1998). WEBDAV: IETF standard for collaborative authoring on the web. `http://www.ics.uci.edu/~ejw/authoring/intro/webdav_intro.pdf`. Cited 29 January 2004.

[Ferg, 2003] Ferg, S. (2003). Python & Java: a side-by-side comparison. `http://www.ferg.org/projects/python_java_side-by-side.html`. Cited 20 January 2004.

[Foundation, 2003a] Foundation, O. S. A. (2003a). Notes from the Chandler demo. `http://wiki.osafoundation.org/twiki/bin/view/Journal/ORAEmergetTechConferenceTranscript20030423`. Cited 20 January 2004.

[Foundation, 2003b] Foundation, O. S. A. (2003b). What's compelling about Chandler: A current perspective. `http://www.osafoundation.org/Chandler_Compelling_Vision.htm`. Cited 14 January 2004.

[Giampaolo, 1999] Giampaolo, D. (1999). *Practical File System Design With The Be File System*. Morgan Kaufmann Publishers, Inc.

[Huynh et al., 2002] Huynh, D., Karger, D. R., and Quan, D. (2002). Haystack: A platform for creating, organizing and visualizing information using RDF. `http://haystack.lcs.mit.edu/papers/sww02.pdf`. Cited 2 February 2004.

[Newman, 1997] Newman, C. (1997). IMAP URL scheme. `http://www.faqs.org/rfcs/rfc2192.html`. Cited 21 March 2004.

[Open Source Applications Foundation, 2003] Open Source Applications Foundation (2003). Chandler architecture. `http://www.osafoundation.org/architecture.htm`. Cited 14 January 2004.

[Slein et al., 1998] Slein, J., Vitali, F., Whitehead, E., and Durand, D. (1998). Requirements for a distributed authoring and versioning protocol for the World Wide Web. `http://citeseer.nj.nec.com/slein98requirements.html`. Cited 28 January 2004.

[Stein, 2000] Stein, G. (2000). DAV frequently asked questions. `www.webdav.org/other/faq.html#Q2`. Cited 29 January 2004.

[Venners, 2003] Venners, B. (2003). Programming at Python speed: A conversation with Guido van Rossum, part III. `http://www.artima.com/intv/speed.html`. Cited 20 January 2004.