Carleton University
95.495* Honours Project
Design and Implementation of an Information Management System

Dave O'Neill
<dave@nexus.carleton.ca>
Student number 177331
Advisor: Dr. M. Weiss

16th April 2001

Contents

List of Figures

List of Algorithms

Chapter 1

Introduction

1.1 Motivation

Every day, humans accumulate many small pieces of information. Some of this information can be easily stored in existing systems such as address books or calendars, both of which have been implemented in software *ad nauseum*. However, other pieces of information are not easily categorized for handling by a domain-specific software tool. For example, what would you use to keep track of movies recommended by a friend or gift ideas for a relative? Probably nothing. Most people[1] store this sort of information the old fashioned way – on paper, either in a notebook or worse yet, on Post-It notes stuck to the inside cover of their day planner or to the edge of their computer monitor. Those who do use an electronic solution frequently use plain ASCII text files, or an email folder to store this sort of data. Searching for and retrieving the information when you need it becomes problematic, if not impossible. Are you likely to remember the name of the Thai restaurant in Toronto your friend recommended 6 months ago? Even if you wrote it down somewhere, can you find it?

The task, then, is to develop a solution to manage small, unrelated bits of cross-domain information, allowing storage, update, searching and retrieval, while making it simple for the user. We will call this system the "Scraps" system.

1.2 Objectives

This project aims to:

- design an API for storage and retrieval of small pieces of information.
- design an XML format for the serialization of this information.
- implement a prototype client and server implementation using the designed API.
- implement serialization of this data for import and export.

In accomplishing these goals, I hope to further my understanding of information management, XML, network communications, and remote procedure calls.

[1] Source: informal survey conducted of 10 students and 10 software professionals in the Ottawa area.

Chapter 2

Methodology

In approaching this design problem, existing solutions that provide solutions to similar problems were examined. Then, the high-level requirements for the system were determined, using the initial concept and the lessons learned from existing solutions as a basis. Following this, potential technologies were evaluated and chosen. These steps are outlined in this chapter.

Once these steps were completed, a final design for the overall system was established. This is described in Chapter 3.

2.1 Evaluation of existing solutions

There are many software solutions that attempt to solve similar problems to the one described here. Enterprise data management solutions are plentiful, for example, however they are several orders of magnitude more complex than needed in this situation. Instead, the evaluation of existing technologies will focus on those solutions that can be used on a personal level.

The five below were chosen based on relatively non-objective criteria — these were the applications that in part inspired the creation of this system — and evaluated to determine their strengths and weaknesses with respect to personal data management:

- The bookmark system used by the Netscape web browser.
- The Radio Userland outliner
- The Palm Pilot memo pad
- Electronic Mail
- The Google search engine

The benefits and drawbacks identified in these programs are summarized below.

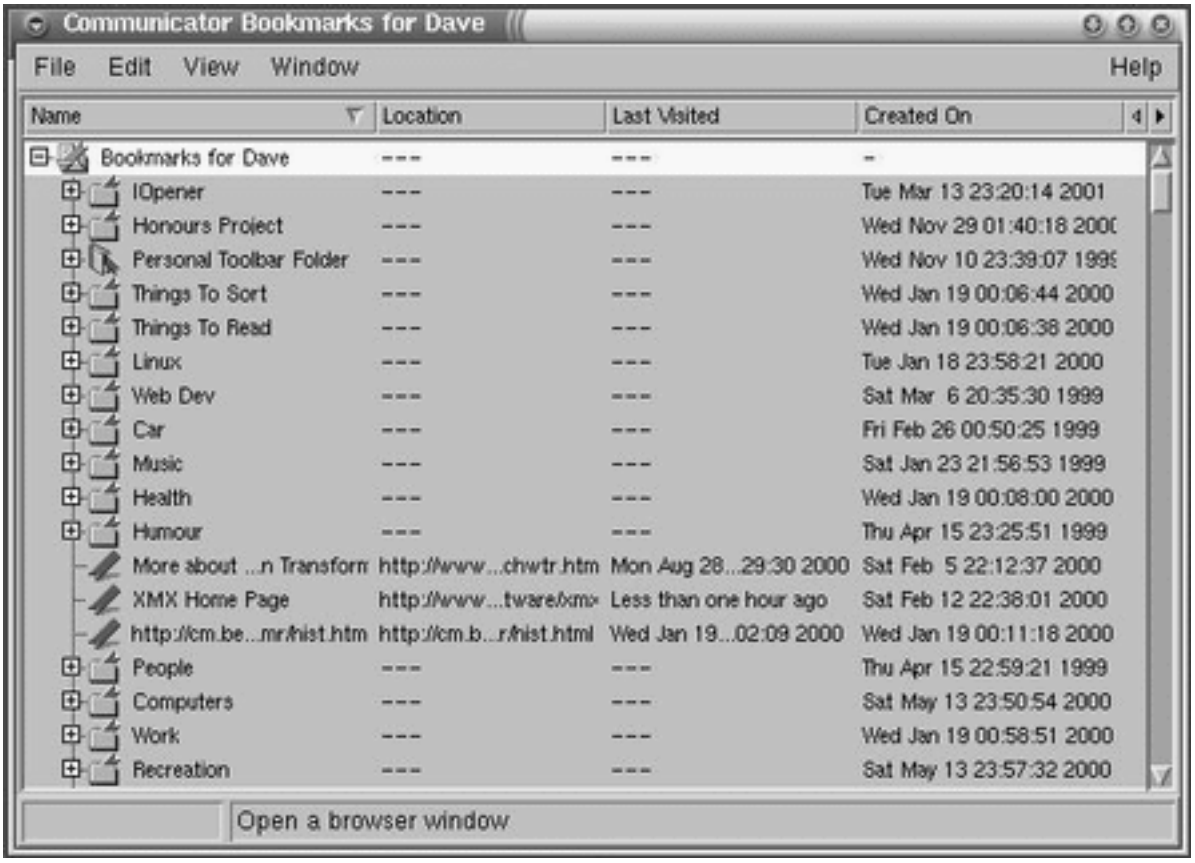

Figure 2.1: The Netscape bookmarks manager

2.1.1 Netscape Bookmarks

The Netscape bookmark tool isn't something typically thought of when you think of an information management tool, but it does manage a particular type of information — HTTP and FTP URLS — for the Netscape web browser. Other internet browsers have similar functionality, however we will focus on Netscape's solution here. The version of Netscape evaluated here was version 4.76 for Linux.

Netscape's bookmark tool certainly isn't the greatest way to manage information — it has induced many software creators to author replacement tools [Sherman, 1999] to provide better functionality — but, it does have its good points in addition to the bad.

On the pro side, Netscape's bookmark system allows for simple, one-touch addition of a bookmark for the page being viewed. Location, title information, and creation date are automatically stored when creating a bookmark this way. It also updates a "Last visited" field each time the link is used, so you can track recently (or not so recently) used information. The bookmark system also allows for the addition of a textual description of the bookmark in addition to the title. Editing a bookmark can be seen in figure 2.2.

A plus for the Netscape system is that it allows a hierarchical arrangement of nesting folders (see figure 2.1), allowing information to be categorized and grouped. It also uses an open, readable format (HTML with Netscape-specific element tags) for data storage, allowing third party apps like the replacement tools mentioned above to

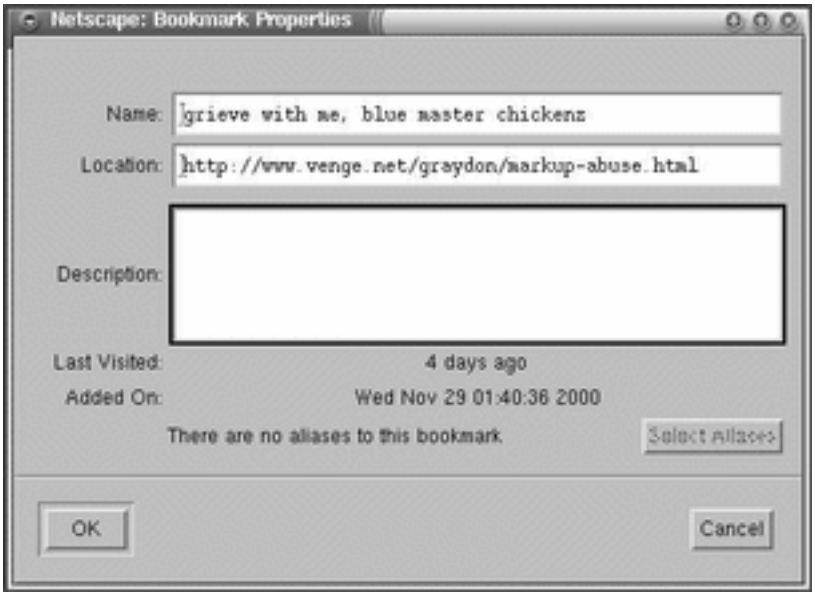

Figure 2.2: Editing a bookmark

manipulate the data while still allowing full access by the browser.

On the con side, the easy-to-use system becomes increasingly difficult to handle as the number of bookmarks rises. Locating the information you want, even with the built-in search tool, can be nearly impossible with hundreds of bookmarks in dozens of categories. As the quantity rises, the ability to properly sort and categorize the bookmarks drops. As well, the search tool does not provide a listing of multiple results. Instead you need to step through the results individually, using the "find next" operation, rather than being able to see all search results at a glance.

The hierarchical arrangement of the bookmarks can be a benefit provided that it can be adequately maintained, but one of its main drawbacks is that it is difficult and time-consuming to have the same information listed under multiple folders. To do this, Netscape provides an "Alias" option, allowing you to have a pointer back to the original bookmark. However, in practice, this is cumbersome to use, and thus seldom is.

2.1.2 Radio Userland Outliner

Radio Userland (`http://radio.userland.org`) is an application and webserver that allows users to customize online content, and to publish their own. In addition to its many other features, it allows creation and editing of content in using its built-in outliner. That outliner's native format is the Outline Processor Markup Language (OPML)[OPML 1.0, Winer, 2001].

The stated goal of OPML is

> [...] to have a transparently simple, self-documenting, extensible and human readable format that's capable of representing a wide variety of data that's easily browsed and edited. As the format evolves this goal will be preserved. It should be possible for a reasonably technical person to fully understand the format with a quick read of a single Web page.

This is extremely similar to our goals, so obviously OPML and its outliner tool bears a closer look.

The main benefit of the outliner is its ability to link in others' OPML code and display it within the context of your own outline hierarchy. The OPML format itself is simple, well-structured, and stores appropriate metadata such as the creator's name and email address, a title, and creation/modification dates. It's also designed with human-readability in mind. An example of OPML can be seen in Algorithm 1.

Algorithm 1 An example of OPML

```
<?xml version="1.0" encoding="ISO-8859-1" ?>
<opml version="1.0">
  <head>
    <title>Sample OPML</title>
    <dateCreated>Sat, 03 Mar 2001 19:42:29 EST</dateCreated>
    <dateModified>Mon, 05 Mar 2001 22:11:39 EST</dateCreated>
    <ownerName>Dave O'Neill</ownerName>
    <ownerEmail>dmo@acm.org</ownerEmail>
  </head>
  <body>
    <outline text="Today's TODO List">
      <outline text="Fly to Mars" />
      <outline text="Download movie listings" type="link"
               url="http://ottawa.film-can.com/avantgo/" />
      <outline text="Eat Cheese" />
    </outline>
  </body>
</opml>
```

The downside to this system is that it follows a traditional outliner model. Content organization is strictly hierarchical, and while this is a good model for outliners, it isn't necessarily the best choice for a general information system.

2.1.3 Palm Pilot Memo Pad

The basic Memo Pad that ships with PalmOS allows a user to create and store brief notes that can later be searched and viewed on the device, or transferred to another PalmOS device or a desktop computer.

Creating and editing a memo is very simple (see figure 2.3). There is only one text area to fill, and your first line in it becomes the memo title. The size of a memo is constrained at 4,000 characters, but since most memos are entered using Palm's Graffiti handwriting recognition, they are unlikely to get this large. Memos can be assigned to a category for grouping purposes.

The main benefits of this tool are its simplicity, and the ability to easily and quickly transfer memos or entire categories of memos to another Palm user via the Palm's built-in infrared data port. Also, the full text of these memos is searchable through the Palm's search capability.

The drawbacks are mostly related to the simple nature of the tool. There is no metadata kept or transmitted with each memo as a memo simply consists of unstructured text, where the first line is the title. Also, there is only one level of categorization allowed with no hierarchies, and memos cannot be listed in multiple categories.

Figure 2.3: Editing a memo on the Palm

2.1.4 Electronic Mail

This "solution" is really a number of solutions, one for each piece of email client software in existance. Many people use their email inbox and folders as a filing system for information they've recieved. Some even send themselves email so they can file a particular piece of information with the rest. The pros and cons of doing this are considered on an aggregate basis, taking into account the general properties of email readers and not focusing on any single reader.

The benefits of doing this are fairly obvious to anyone accustomed to using electronic mail. Information in an incoming email can be immediately filed to the appropriate storage location. It's a format that encodes title, creator and date information, and allows for arbitrary content of any type and size. It also allows the end-user to choose from a wide variety of user interfaces (in this case, mail clients). The tool is also famliar and in most cases used on a daily basis.

The drawbacks of this may not be as obvious. First, once received, email is not a terribly efficient storage medium, as the overhead of the mail protocol in question (generally SMTP) is retained in multiple mail header lines, most of which are not relevant once the message has been received and stored. Examining one of the author's own email folders used for this purpose showed that for an email file containing 5858 lines in 219582 bytes, only 3823 lines totalling 119821 bytes were useful information (including the Subject: and From: headers). The remainder (46% of the raw file size, or 35% of the lines) was header information used in the SMTP protocol.

Second, email clients do not make it terribly easy to import and export individual pieces of information for use on portable or handheld devices. Third, the use of email as an information management device generally precludes allowing third-parties access to the data contained, as much of the surrounding information may be personal or confidential in nature.

2.1.5 Google search engine

Google (`http://www.google.com`) is a search engine designed to accurately index and search the World Wide Web. While not a personal information management device, it is worth considering for its search interface.

The search interface itself is extremely simple. The default search is to AND all of the search terms together. More complex searches are possible, but in fact they are seldom needed. The results of the search are displayed together in ranked order (in stark contrast to most of the search capabilities in existing information management programs).

Its main benefit is that it is extremely accurate — so accurate that they cockily place an "I'm Feeling Lucky" button on the search interface that will take you directly to the top-ranked search result. However, to achieve such accurate results, Google uses a complex page ranking system that could not be duplicated on a personal scale.

2.2 Requirements capture

In examining the features and limitations of the existing solutions, a number of essential requirements for the new system were identified.

It must be simple to use.

The system must be simple for the end-user to learn and use, and it must be simple for a programmer to understand and implement a front-end client or a back-end server component.

It must allow both peer-to-peer and client-server models.

The system will be designed to have both client and server components. However, these components can be combined into a single application that can be run as a "peer" on the network and interact with other clients and servers.

It must be network-enabled.

The system must allow clients and servers to speak to each other over any TCP/IP network.

It must have import and export capabilities.

The system must be able to import from and export to a common data format, to facilitate data interchange with other programs that do not or cannot use the network RPC-based method to communicate.

It must not be limited to any hardware or software platform.

The system shouldn't be tied to a particular hardware or software platform, nor should implementation be confined to one programming language.

It must not be tied to any particular backend database.

The system must not require any particular database backend. Individual implementations can take advantage of SQL, XML structured storage, Berkeley DB, or anything else. However, the design will be such that a specific back-end storage solution is not specified or required.

It must be designed to allow for future enhancements.

The protocol and export format shall be designed with future expansion in mind where possible.

It must support a full set of operations.

The protocol and implementation must include commands for adding, deleting, editing, viewing, searching, importing, and exporting data to and from the system.

It must allow both simple and advanced searches.

A simple way for users to query the system is necessary, as the majority of queries will most likely be multiple keywords AND'ed together. However in ensuring simplicity, the system must not limit the ability to allow powerful advanced searches.

It must store metadata on each piece of information.

The system must store a certain amount of metadata on each piece of information. Metadata elements such as time of last update, name of author, and title information are essential.

It must be a generic solution.

The set of data and metadata required will be defined as generally as possible to allow for the management of many types of potentially disparate data in the same system.

It must be usable without a network.

The application must be able to function in a standalone environment. In a networked environment, it will be able to take advantage of network searches and peer-to-peer data exchange, however it must be fully usable in its own right as a stand-alone personal information management tool.

2.3 Evaluation of potential technologies

In evaluating existing technologies for use, there were three areas of focus; the data export format, the RPC mechanism, and the search syntax.

2.3.1 Data export format

Since an extensible, platform independant standard is needed for data export, the only real option here is XML.

XML (the eXtensible Markup Language) is a markup language intended to represent structured information in a self-describing, text-based format. It was designed to be simple, straightforward, and easy for applications to process. Since it is a text-based format, it can be used across multiple platforms and operating systems without

difficulty. This, and the fact that it is a free, open standard, and that there are several reusable libraries and components that support reading and writing XML made it ideal for this system.

With the choice of XML, the field was narrowed but by no means complete. With XML, there are several other related standards and standards-in-progress. In the interests of simplicity and stability, this project will not use XML schemas or namespaces, it will use only XML 1.0[XML 1.0] and the DTD approach to defining our XML elements.

XML itself does not define a document format, it merely defines the parameters within which one can be created. One must create a DTD, or Document Type Definition, describing the valid content and structure of an XML document. Only then do you have an XML-based document format.

In looking to find a relevant existing XML DTD for this application, none were found. While there are many specifications for XML data formats available, the vast majority of them are unsuited to this system, being either too domain-specific, or too complex. So, it was determined that for this system, a new document specification would need to be created. This is explained further in chapter 3.

2.3.2 RPC Protocol

A remote procedure call (RPC) based protocol, used to execute procedures on a remote machine, is a good way to achieve the network transparency desired for this system without introducing excess complexity to the applications on either end.

Since XML was chosen to be the data export format, it was only natural to consider the XML-based remote procedure call protocols XML-RPC and SOAP as possible choices for our remote communications. Both of these RPC methods work by marshalling the data into XML at call-time, passing the XML across the wire to the remote machine, where it is unmarshalled for the application on the remote end to process the call. Upon return, the process is repeated in the other direction. The preferred method of network transport for both of these protocols is HTTP over a TCP/IP network.

XML-RPC is an RPC protocol developed in mid-1998 by Dave Winer of UserLand. It was designed to be a "just-enough" solution for remote procedure calls over XML.[Winer, 1998] As such, it is relatively lightweight and simple, with few "bells and whistles". The protocol is frozen and no further changes to it are being considered.

SOAP[SOAP 1.1] is a follow-on to XML-RPC, originally created by a collaboration of UserLand, DevelopMentor, and Microsoft. It is now under development by a W3C working group. It contains many more features than XML-RPC, including the use of XML Schemas and namespaces, new data types, different error handling, and a much more complex encoding.

Both of these have good cross-platform and cross-language support, and are in reasonably wide use. At first glance, it would seem that SOAP would be the natural choice for this application, as there is existing support for it, and it is expected by many to supplant XML-RPC for most applications. However, XML-RPC is stable, contains all the features needed for this system, and is not suffering from the creeping featurism that has befallen SOAP.

2.3.3 Search syntax

Knowing that we will be using XML for both export and transport, it would seem to be natural to use an XML-based search syntax. There are several attempts, in varying stages of completion, to define an XML query

language. Of these, the W3C's work-in-progress XQuery [XQuery] appears to be the only real contender for a standard.

The problem with XQuery in this case is that it is not a query language for defining general searches as XML, it is a query language for defining searches *on* XML. As we make no assumptions as to the back-end database implementation, we cannot assume that the database will be in XML. Because of this, XQuery is unsuited to the task at hand. Other search languages such as SQL also have similar drawbacks, being designed to query a certain type of data.

The query description language needs to be structured to meet several needs. The client must be able to translate the appropriate end-user search input into the description language for transmission to the server. The server must be able to translate the query description into the correct format to query the back-end database. Finally, the query description language must be serializable for storage itself, or for export.

Because of these needs, and because we will need to perform a translation on both ends, we will need to develop our own simple search encoding. This is defined in Section 3.3.

Chapter 3

Design

After arriving upon a set of general requirements and some suitable technologies to use, the major design questions still remain. We need to determine exactly what set of data we wish to store and make available. Then, we need to determine how this data will be stored, exchanged, and searched.

3.1 What data do we want?

3.1.1 The Scrap

Since the system is being designed to store small pieces of information, we need to establish how to describe the smallest piece of data in it. We will call this basic unit a *scrap*. In keeping with this theme, we will refer to our local database of scraps as a *collection*, and an XML-exported group of scraps as a *scrapbook*. These terms will be used throughout the remainder of the document.

In determining what sort of data a scrap should hold, three main goals were kept in mind. First, the data set must be complete enough to allow all relevant meta-information about the data to be stored along with it. For example, retaining associated information like a short description, the date created, and the author's name and email address is essential.

Second, it must be generic enough to allow the same scrap specification to be used to describe data from many different domains without waste. Having data fields that are seldom or never used because they are irrelevant to the type of data stored within is to be avoided.

Third, it must be extensible. There may be future uses for this system that have not been envisioned yet, or the design may simply overlook something. The design should not preclude future extensions.

The goals above parallel the larger goals of the Dublin Core Metadata Initiative[Dublin Core, 1999], and so this design will draw upon, and aim for compatibility with, the Dublin Core Element Set.

The data that will be stored in a scrap are:

data The data itself. This can be in any of the allowed types (see below). This specification is content-neutral, so the data field can contain anything that can be safely represented inside an XML document. However, it is intended that only text-based content will be supported for the time being.

data type This describes the type of data present. The purpose of this is to allow for special handling of data depending on type. Currently, we will only support three types:

- **text** This indicates that the data contained in the data field is plain ASCII text. There is currently no support for Unicode.
- **url** This indicates that the data field contains a single URL. This allows references to external content or larger documents to be referenced from within the system. Client programs should allow the URL to be opened in the appropriate application (web browser, FTP client, etc).
- **query** This indicates that the data section contains a stored search. See section 3.3 for details.

title This is a title for the scrap. It will be frequently used by client applications as a title, and as a quick indicator of the content of the scrap. It should be short and descriptive.

description This contains a longer description of the content. It can be more verbose than the title, but should be no more than a summary of the actual data, not a duplication.

keywords Each scrap will be associated with one or more keywords. Keywords should be short, only one or two words long. Spaces, apostrophes, and hyphens may be used. The usage of other punctuation characters is not permitted. The keywords are used to locate and extract scraps from a collection via the search mechanism explained in further detail in section 3.3.

date Each scrap can have one or more dates associated with it. These dates will typically be automatically stored by the software, and not entered directly by the user. Currently, four different dates can be associated with each scrap:

- **created** The date that the given scrap was initially created. This should not be changed over the lifetime of the scrap.
- **modified** The date the scrap was last modified on this system. Since scraps can be copied and exchanged, this date only applies to this copy of the scrap.
- **accessed** The date the scrap was last accessed on this system. As above, this date only applies to this copy of the scrap. It is updated on any access, so both reads and writes should change this date, while searches should not.
- **imported** The date the scrap was imported into this system. This date should be set if the scrap is imported into the local scrap collection from another system via RPC, or from an exported XML scrapbook.

creator The creator of the scrap. For the creator, a name and email address will be stored. This identifies, and provides a means of contacting, the person responsible for this information.

contributors We also wish to store a list of all people who have contributed to the particular scrap. This allows tracking of changes from the original as it is modified. For each contributor, we will store name, email, date of contribution, and an optional note describing the changes.

One more piece of information must be stored, and that is an *id* that allows each scrap to be uniquely identified. Since the inherent design of this system is for scraps to be exchanged, we need some form of unique identification that will not clash with identifiers created by others. The nature of the system is also such that we cannot have some central agent assigning unique identifiers. Because of this, we will use the DCE UUID (Universal Unique Identifier) scheme. UUIDs have since been standardized in ISO/IEC 11578, but as this ISO standard is not freely available, the documentation at `http://www.opengroup.org/onlinepubs/`

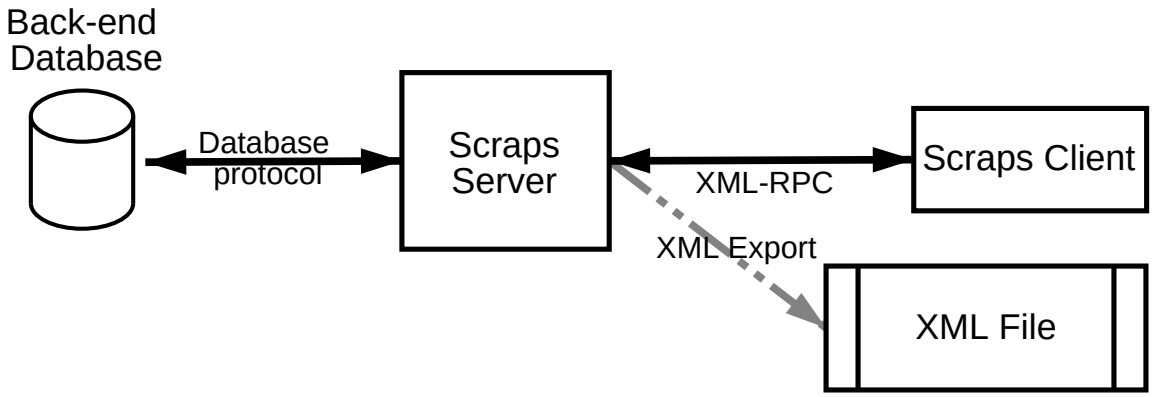

Figure 3.1: Communication within the Scraps system.

`9629399/apdxa.htm`[UUID] was used for our implementation. This scheme could be easily replaced by another method of generating a globally-unique identifier, but the UUID scheme serves our purposes adequately for the moment.

Below, in section 3.2, we describe how this data will be exported and exchanged.

3.2 How do we communicate?

At this point, we have already decided upon a network-based, client-server model for our system. This model uses two methods of communication for the purposes of data exchange. A diagram illustrating these possible communication methods can be seen in figure 3.1.

The first is data exchange via the XML-RPC protocol, intended for talking to clients and servers conforming to the design of this system. As mentioned above, XML-RPC communicates across a network connection using the HTTP protocol. Ignoring programming language dependent issues, a typical XML-RPC call proceeds in the manner described in figure 3.2. It is generated in native format on the client, encoded as XML, transmitted to the server, decoded, and processed. This then happens in reverse, to return the results to the client.

The second is the data export format, intended for data archival, off-line operation, and interaction with other applications. This method requires that the exported files be transported in some out-of-band manner before being imported through some client and into a server, either the originating one, or another server.

Generally, clients are expected to have some sort of network access. Most, if not all, modern desktop operating systems allow use of the loopback interface to access the local machine using network protocols, so this is a reasonably safe requirement. Clients that require disconnected access, where no communication with the scraps server is possible, can be implemented to cache search results in the client for later use, or to use the import/export mechanisms provided.

3.2.1 The API

The RPC API sits on top of XML-RPC (as chosen above in section 2.3.2). The goal of the API design was to support all basic functionality for manipulating the scraps in a collection as simply as possible. With that in mind, the API commands currently designed are:

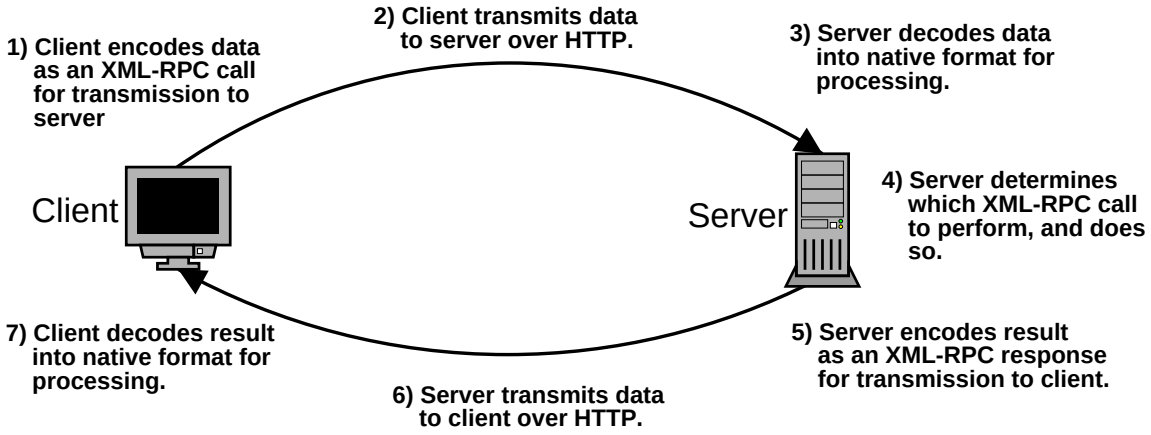

Figure 3.2: Life-cycle of an XML-RPC call.

scraps.newScrap Creates a new scrap in the collection using the given data.

scraps.fetchScrap Returns all data on a scrap with the given scrap identifier.

scraps.deleteScrap Deletes a scrap from the collection.

scraps.saveScrap Updates an existing scrap with new information, or stores a newly-imported scrap obtained from another server.

scraps.search Searches the collection for scraps matching the given criteria

scraps.exportScrap Exports the scrap with the given scrap identifier.

scraps.exportSearch Exports all scraps matching the given search criteria.

scraps.import Imports a single scrap or a scrapbook of scraps into the collection.

Each of these functions takes a username and password, both in plain ASCII text, before the function-specific arguments. The user must be authenticated successfully for the command to be processed

There are also commands for manipulating the user database. Currently these are

scraps.user.add Adds a new user to the database with the given username and password.

scraps.user.remove Removes the user with the given username.

scraps.user.changePassword Alters the password for the given user to the new password supplied.

scraps.user.verify Verifies the validity of a username/password combination.

scraps.user.list List all valid users in the database.

Each of these functions takes a username and password, both in plain ASCII text, before the function-specific arguments. The user must be authenticated successfully for the command to be processed.

This API is fully described in Appendix C.

3.2.2 The export format

The export format is to provide a serialization of all data pertaining to a scrap in the collection. As mentioned previously, this will be done using valid XML conforming to a DTD defined for this system. All of the data and metadata for the scrap, as outlined above, will be output to this export format.

The export DTD is fully described below in Appendix A. A sample of the export format in use can be seen in Appendix D.

3.3 How do we find it?

Now that we have defined what the data is, and how it can be accessed, we need to know how to make that access meaningful. Specifically, how can we fetch information from the collection?

The first way of fetching a scrap is by retrieving it by id. This is done by using the `scraps.fetchScrap()` method, which is called with an id as an argument. It will return the full data for the scrap matching that id, if it exists on the system.

The second way of fetching information is by performing a search. Searches are primarily keyword based[1], using boolean logic (AND, OR, NOT) to tie the keywords together to achieve the desired combination. On the server, the search is performed by matching the desired keywords from the search criteria against the stored keywords for each scrap.

3.3.1 Why a keyword-based system?

Once we have stored data on the system, we need a way to retrieve it later. A plain full-text search is an obvious first choice. The main drawback of this type of search is that it doesn't understand the context and semantics of the information. Also, the information itself may not be plain text, as a scrap can hold other data, such as a URL or XML-formatted text. Natural language processing could take care of some of this, but it would not provide sufficient improvement, especially in cases where the data is too short, or does not actually contain any words that would provide an adequate search key.

Instead, we will use a user-provided keyword system. This has a few obvious benefits:

- Users can supply meaningful keywords, using appropriate context.
- Keywords need not be drawn directly from the content or other metadata.
- Since keywords are user-supplied, the user is more likely to use the same key terms in searching.

Also, there is no reason why a client program could not automatically generate the keywords from the scrap data, description, or title if the client author so desired.

[1] However, support exists in the protocol for searching on other metadata, such as dates or creator names. This has not yet been implemented in the prototype code.

3.3.2 The Search Encoding

Searching must be simple, but allow more powerful searches to be used. Balancing these can be difficult as simplicity and power often conflict. In addition, we needed to define a search in such a way that:

- The same search format can be used from many different clients, possibly implemented on multiple platforms and in multiple programming languages.
- The client can make the search as simple or as powerful as needed for the users.
- The system can store searches for reuse later.
- The search system can be expanded for other types of search, such as date ranges, authorship, or data type.

To do this, it was necessary to design an intermediate format to encode searches. The search would be entered into the client program in a non-specified format, translated into our intermediate format for transmission to the server, and in the server code, translated into the appropriate format for searching the implementation-specific back-end database.

In keeping with the rest of the system, this format is structured in an XML-friendly manner. A search in our intermediate format can be in one of two encodings. First, it may be encoded as a struct in XML-RPC (as per the API specification in Appendix C) for transmission to a server, or it may be encoded in XML (using the DTD specified in Appendix B) for storage as a scrap, or for stand-alone export. The software on either end may internally represent the search in any way necessary.

The simple search format is basically a prefix-notation system, where the arguments follow the boolean operator in a list. This is simple to encode both as straight XML (where the arguments to the boolean operator are contained in a tag that defines that operator) or in an XML-RPC struct encoding (where the arguments to the boolean operator are contained as a value in that operator's struct member).

Inside this prefix-notation arrangement comes the search specifiers themselves. The boolean operators can be combined and embedded in any sort of nested arrangement, but ultimately they must contain a search specifier. Currently defined specifiers are *keywords* and *dates*.

A *keyword* specifier consists of simple character data representing one keyword. Multiple keywords are chained together appropriately by the boolean operators.

A *date* specifier is named "created","imported","accessed", or "modified". These labels serve to identify which date field of the scrap is to be matched against. Inside the date specifier is another element that contains the actual date (as plain text data) to be matched. The name of this element defines how the date is to be matched. Valid values are "on", "before", and "after", defining whether or not we wish to retrieve items that match the date exactly, occur before it, or occur after it.

Wrapping this boolean logic is a query specifier. Inside a query specifier, there are four valid elements; the three boolean operators, and an optional "server" specifier that provides a URL to the server we wish to search. See algorithm 2 for an example using a LISP-like syntax. Only one of the three boolean operators can be present inside a query. The search must be formed so that there is only one top-level boolean element.

Above this lies the top level of a search construct. This is the *multiquery* element. It can contain multiple query elements, the results of which are aggregated together and returned once the query is complete. Note that this definition does not allow a multiquery to perform operations on the retrieved results. For example, you cannot search Server A and Server B and then return the set of results that is present only on Server A but not on Server

Algorithm 2 A simple search example formatted in a LISP-like style.

```
(and
  (keyword “banana”)
  (not
    (keyword “ninja”)
  (or
    (keyword “pickle”)
    (accessed
      (after “Sat Mar 10 09:08:00 EDT 2001”)
    )
  )
)
```

This describes a search equivalent to the sentence:

> Return all scraps matching keyword “banana” and not matching keyword “ninja” and matching keyword “pickle” or accessed after March 10th at 9:08am.

B. This is an intentional omission to reduce complexity, though this an area where future enhancements to the search protocol may come into play.

The multiquery, though valid in all contexts, does not provide any benefit nor make any sense when the queries contained within it do not refer to multiple servers.

3.4 How do we store it?

As a general rule, the system should not be tied to any particular storage implementation. However, to implement a prototype, we need to choose some form of back-end storage. For ease and speed of development, it was decided that the prototype would use an SQL database.

The database used for this prototype is MySQL (`http://www.mysql.com/`).

3.5 Design Conclusions

This system will consist of two main components, a client component and a server component. These two have different responsibilities . The server is to store the data and serve it up in response to appropriate API commands. The client is to present an appropriate front-end to the user, and translate user input into the appropriate API commands to manipulate data on the server.

Communication between the components will use our defined API over the XML-RPC protocol. Indirect communication can also be achieved through exporting and importing data from each collection.

The export format will be XML, based on a DTD we have defined.

Search criteria will be encoded by the client in a set format, and transmitted to the server, where it will be decoded to perform the search on the appropriate back-end database. Search criteria can also be serialized as XML and stored in the system as a scrap, for later use.

The prototype will initially implement only the keyword-based portion of the search schema, with other portions to follow.

Chapter 4

Implementation

The prototype implementation is intended to be a demonstration of the capabilities of the Scraps system.It consists of three separate parts; a server component, a web-based client component, and a partially-functional GUI client.

For ease and speed of development, these three pieces were developed using the Perl programming language, using the excellent reusable components available through the Comprehensive Perl Archive Network (CPAN `http://www.cpan.org/`).

No performance tuning was done for this implementation, and as such, there are many open possibilities to enhance the speed and memory footprint of these programs.

This chapter documents the creation of these three components.

4.1 Server

The overall design of the server is that of a simple HTTP server that handles XML-RPC requests, decodes them from the XML-RPC encoding for handling by local Perl code, and then encodes the returned result in XML-RPC to be returned as an HTTP response.

The core of this is the use of Perl's HTTP::DAEMON class, to handle the HTTP requests, and the FRONTIER::RPC2 class, which handles the marshalling and demarshalling for the XML-RPC protocol. The database back end used is MySQL (`http://www.mysql.com/`), a freely available simple SQL database. Perl's database library, the DBI and DBD::MYSQL classes, are used to connect to the database

4.1.1 Database Schema

The database schema for this prototype was created to be relatively simple, and to follow as closely as possible the Scraps DTD in SQL. This SQL schema consists of three tables, "scrap", "contributor" and "user". An E-R diagram of this schema can be seen in figure 4.1.

The "scrap" table holds all information on a scrap. Each scrap in the database can be associated with multiple contributors, which are stored in the "contributor" table. Contributors in this table are associated with scraps by the scrap id element.

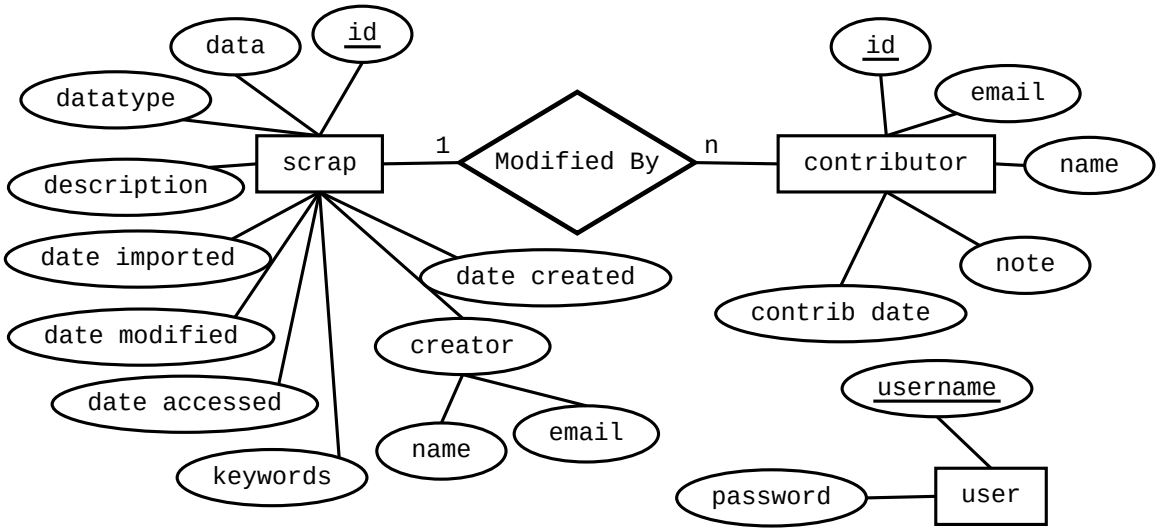

Figure 4.1: An E-R diagram of the back-end database

The “user” table holds username and password information that is currently only used to connect to the system, and is not associated with the scrap data at all.

In the first iteration of the prototype, a fourth table, “keyword” was also used. In an early revision of the prototype, the “scrap” table was further decomposed to store multiple keywords in a separate table, using the scrap id as a linking element. This proved problematic, as searching for scraps that match multiple keywords became difficult because keywords were stored in a separate table. Instead of this table, the “keywords” field was introduced into the “scrap” table. Multiple keywords are stored in this field using a delimiter to separate them. This allows search queries to be implemented in a much more straightforward manner.

4.1.2 Program flow

The prototype server is written in a simple linear fashion. At start time, a connection to the database, an HTTP server object, and an XML-RPC decoder are all created, as is a lookup table mapping API call names to Perl function calls. We then wait on incoming HTTP connections.

When an HTTP connection is received, we stop accepting incoming connections (allowing them to queue up) and handle the newly received connection. It would be desirable for a production implementation of this code to handle multiple connections simultaneously through either a threaded or forking server model, however this adds needless complexity to a proof-of-concept prototype.

For each incoming connection, we determine its validity as an XML-RPC call. If it is not one (ie: is some other HTTP request, for a document perhaps) we return an HTTP “Forbidden” response. Otherwise, we begin the process of decoding the call. This is handled by the `parse_request()` function in our server code.

`parse_request()` first attempts to decode the XML-RPC call, and then check it to ensure it is well-formed. If it is not, the appropriate error is returned using the XML-RPC fault mechanism. This mechanism will generate a catchable exception on the client.

If it is a well-formed call, we then begin to handle its data. As explained in Appendix 2.3.2, the first two

arguments[1] to any of our API calls is a username and password. We use this information to validate the user against our database (the `user` table). If the user is not valid, we return an INVALID_AUTHENTICATION fault to the client.

We then check to see that the call invoked is actually a defined one on this server. We perform this after authentication in order to not give away any information whatsoever to an unauthenticated user.

After performing this step, we then use the lookup table to determine the appropriate Perl function to handle this call

4.1.3 Handling of API calls

The API calls are handled by a corresponding Perl function. To execute the corresponding Perl code for an API call, a lookup table keyed on the full API call name is used to return a function reference. This function is then invoked, passing it the authenticated username, a reference to our database connection, and the remaining argument values given to the call. It will then either return success, in which case the results requested by the call are returned, or failure, in which case an appropriate XML-RPC fault is returned indicating the failure condition.

The Perl functions themselves are implemented in the SCRAPS and SCRAPS::USER packages. Unfortunately, due to the lookup method used, these packages are not Perl objects, but are simply functions defined in another namespace. This is sufficient for our purposes, but is not particularly elegant. A proper object-oriented dispatch method would be preferred, and will be implemented in the future, possibly using one of the creational design patterns[Gamma, 1995, pp87-107].

4.2 Web Client

The web client is designed as a simple multi-page CGI script, using several publically-available Perl modules, specifically the CGI module to deal with CGI interaction, the HTML::TEMPLATE module to encapsulate the HTML portions of our interface away from our code, the FRONTIER::CLIENT module as our XML-RPC interface, and the CRYPT::BLOWFISH and CRYPT::CBC modules, used to encrypt our authentication information (more on that later).

In addition to these publically available modules, the code uses a dispatch mechanism originally written by the author for a previous web-based project. Since the web client was implemented after the server, it does use an improved dispatch method (an implementation of the Abstract Factory pattern [Gamma, 1995, (87)]) to invoke the appropriate handler for the requested operation from the CGI request.

This script runs using the standard CGI interface, as tested using the Apache webserver, though it could easily be modified to run as a server extension using mod_perl.

4.2.1 User Authentication

Since our API requires a username and password for each call, we need some way of storing this information for reuse, as it would be extremely annoying to the user if we requested a username and password be entered for each operation. The method chosen for this was to use HTTP Cookies. A “cookie” is a small piece of information

[1] Unlike SOAP, parameters in an XML-RPC call are positional. SOAP uses named parameters.

Figure 4.2: The menu as seen in the web client

stored on the client's browser that is returned to the server with each HTTP request made. To implement this solution, we used the Perl module CRYPT::CBC, which is a simple interface for symmetric encryption of arbitrary data. It can use several underlying encryption algorithms, but we have chosen the Blowfish algorithm as implemented in CRYPT::BLOWFISH. Once a user has supplied their username and password for the first time, they are combined together and encrypted using a secret server key via the CRYPT::CBC module. This gives us an encrypted string we can then use in our cookie.

If an HTTP request comes in bearing this cookie, and the cookie decrypts to provide a username and password that can be validated via the API, the user does not need to enter their password.

4.2.2 Structure

Once a user has logged in, they are presented with a menu. This menu (seen in figure 4.2)lists the operations immediately available to them. At login, the available options are to add a scrap, search for a scrap, import scraps, or log out. Also available are user management commands, adding a user or listing users.

Viewing of a scrap is available only from the results of a search. From the view of a scrap, you can choose to edit it or export it.

Exportation of a scrapbook of scraps can be done from the search results page. First, you define your search criteria such that the data you wish to export is returned. Then, you select "export results as scrapbook".

The search results page is also where a search may be stored as a scrap. Currently, though, there is no way to execute a stored search in this prototype.

Figure 4.3: Web-client search results page.

A screenshot of the search results page highlighting these features can be seen in figure 4.3. The “store search” and “export results as scrapbook” operations can be seen near the bottom. The search may also be refined using the search box at the top.

4.2.3 Event Handling

The basic structure of this CGI script is as follows.

The CGI is invoked by the webserver. It determines the operation to perform by checking the ’op’ parameter of the incoming request. The DISPATCHER class (an Abstract Factory) then creates the appropriate handler.

Handlers are subclasses of a SCRAPS::CLIENT::HANDLER class designed for this application. This class contains common functions (for getting and setting data, generating templates, etc) and defines a common interface of over-ridable functions to be implemented by each subclass to perform the event-specific operations.

Each valid operation has its own subclass, such as SCRAPS::CLIENT::HANDLER::VIEWSCRAP. This is where the actual work is done, performing the XML-RPC calls and formatting the results for return to the browser.

4.2.4 A sample operation - editscrap

It is probably beyond the scope of this report to give a detailed explanation of each operation that can be performed, so in the interest of brevity, only the editscrap operation will be explained.

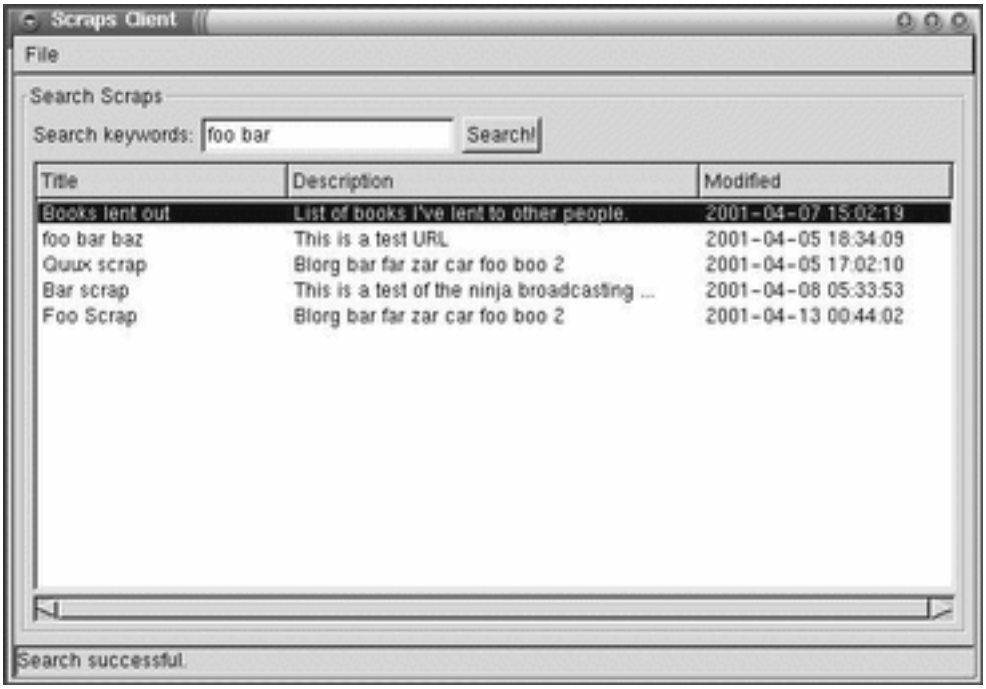

Figure 4.4: Viewing search results in the GUI client.

Once the CGI request has been received, the user authenticated, and the handler class for the "editscrap" operation created and initialized, the handler's `process()` method is called.

This method first checks to make sure any essential data is present in the request. If not, an error is returned. Then, a check is performed to see if we are submitting a completed edit, or if we are requesting the form to edit a specific scrap. If the latter, we retrieve the original data for that scrap, generate the HTML output, and return.

If we are processing a completed edit, we need to construct an API call and send it. This is relatively simple, as none of the data validation is done on the client. All we need to do is construct the appropriate Perl data structure (in this case, an anonymous hash) for the FRONTIER::CLIENT module to turn into an XML-RPC encoded struct. We then perform the appropriate RPC call, and await the results. If it is successful, we use an HTTP redirect to send the user to the viewscrap page, where they will be able to see their new changes. If the call fails, we redisplay the edit form along with an appropriate error message. Since data validation is done on the server, entering of invalid data will trigger an error, and the user should be allowed to correct the data and resubmit.

4.3 GUI client

The GUI client is intended to provide a simple method of accessing the Scraps system without the need of a web browser or CGI-enabled HTTP server. The client is written in Perl, and uses the Gtk windowing toolkit (`http://www.gtk.org/`) for its graphical interface. At the time of this writing, it is not feature-complete. The GUI client currently only supports searching and viewing of scraps. Editing, exporting, and user management are not yet supported.

The design of this client is relatively simple. Using the Gtk event framework, we bind callbacks to events that will initiate the appropriate actions. For example, the button press event for the search button triggers the search callback. This function makes an API call (using FRONTIER::CLIENT) and displays the results in the list widget (see figure 4.4 for an example).

Clicking on the titles of each of the columns in the search list will sort the list by those columns. Double-clicking on any of the search result rows will display a full view of the scrap in a separate window.

Chapter 5

Future Enhancements

Throughout the project, several ideas arose that would be excellent additions to the system, but would be too time consuming to implement within the deadline. During planning and implementation, these ideas were noted, and allowances were made for future expansion in the design and implementation stages to support them.

The author intends to continue work on this project in the future, as time allows. This section presents a few of the more feasible of these ideas.

Transaction-layer security

The current prototype implementation passes all information in the clear over the network. Future implementations should optionally be able to use TLS (Transaction Layer Security) as defined in RFC 2246 `http://www.ietf.org/rfc/rfc2246.txt` (The TLS Protocol version 1.0) and RFC 2818 `http://www.ietf.org/rfc/rfc2818.txt` (HTTP over TLS). TLS is the next iteration of the SSL (Secure Sockets Layer) protocol created by Netscape. This can be easily accomplished using the OpenSSL library available from `http://www.openssl.org/`, however the administrative overhead of generating and managing certificates may not be worth the trouble for most purposes.

Multi-threaded prototype server implementation

Currently, the prototype server only supports single connections. This was a decision made to ease debugging and implementation of the prototype. It would be reasonably simple to implement a traditional UNIX forking server to deal with multiple requests. A threaded implementation may necessitate using another language of implementation, as Perl's threading support is still unstable.

Enhanced User Management

Currently, the API supports multiple users, but does not allow the assignation of permission levels to these users. As such, all users have equal permission to add, edit, and delete scraps.

Improved user management that will allow for permissions to add, edit, delete, and search scraps added or revoked on a per-user basis is definitely necessary.

A further improvement would be to create groups, with which both users and scraps could be associated. This would allow only users in a certain group to view certain scraps.

Categorization System

The keyword-based system is reasonably good for retrieval of information via a search, but does make browsing of data rather impossible, as you need to have defined search criteria.

Adding a categorization system would allow for browsing of the system's content, and allow for easily restricting searches to a specific category. Also, a categorization system could easily be made hierarchical, similar to that of the Open Directory Project `http://dmoz.org/` or Yahoo `http://www.yahoo.com` to allow for "drilling down" into areas of interest or broaden a search across a larger area without having to deal with extra boolean operators or precedence rules when creating a search.

Clients for PDAs and portable devices

A client for PalmOS-based PDAs is a definite priority for the future. This would require the implementation of a viewer that would run on the PDA, and a desktop conduit that would connect to the server and synchronize changes using the XML-RPC API.

An interim solution may be to use existing Palm tools (such as the Memo Pad) to view exported data from the scrap collection.

Online viewing for the Palm may also be possible, given appropriate wireless Internet access.

Improved searching

The search capability could easily be improved, both in the design and in the prototype. Features such as full-text searching could be added to the design, for example. In the prototype, a full implementation of the existing search protocol is necessary.

Extend to allow for binary data

Binary data is easily handled in both XML and XML-RPC through Base-64 encoding. However, extending this system to handle arbitrary binary data may subvert its original goal of being a method for storing and exchanging *small* pieces of information. It was not designed to be a general metadata solution, and should probably not be used as one.

That said, some of the concepts explored here may be useful in designing a generic wrapper for storing metadata on any file format, as an offshoot of this project.

Pluggable "content handlers" for viewing

It would be an interesting extension to add cross-platform pluggable content types, perhaps using the eXtensible Stylesheet Language (XSL `http://www.w3.org/Style/XSL/`) to translate the structured content from the data section into properly formatted user-viewable data. For example, a data-type of xml/movie-review could be displayed and formatted appropriately using this display plugin.

Protecting content integrity

A content integrity protection system is another possible addition. This would *not* be a content protection system in the digital rights management sense of the word. It would not ensure that the content's viewing or distribution could be controlled by the original author; it would be used to ensure that any content distributed as being *from* the original author could be verified as such, and validated to ensure that it is complete, unedited, and unmodified. This could be accomplished using digital signature methods such as PGP or GPG and their existing infrastructure for key verification and key revocation. With a verification system working in this manner, it would be simple to verify that a scrap recieved by searching a network peer is as the author had intended. Digital signatures could also be chained in sequence – this would be used by contributors who have made corrections or updates to a signed original and wish them to be verifiably correct and complete.

Constraints on search results

With the current API, it is entirely possible that a search may return hundreds, or thousands of results. This could potentially cause problems on the client end, due to the sheer number of results overwhelming the client's resources. Two potential remedies for this situation are possible. First, extending the `scraps.search` call to take an argument representing the maximum number of results to return. Second, a new search call could be implemented that would store the search results on the server and allow the results to be retrieved in manageable chunks via repeated queries.

Chapter 6

Conclusion

This project has resulted in the creation of a data model for the storage of small pieces of information, a well-defined protocol (an XML-RPC API) for information storage and retrieval, and an XML-based format describing data conforming to that model.

In the design and development of the above, a prototype server implementation and two clients were created. As well, a search description language was created as part of the API.

The most important of these accomplishments is the design and development of the data model and client-server API for accessing the data. This portion of the project has produced a well-specified API and export format for the data that can be easily implemented (and extended) in the future.

Secondary to this is the implementation of the prototype. While the server and web client are functional, they are by no means polished or "production quality" pieces of software, created to verify the viability of the solution and prove the concept. Undoubtedly, better implementations of this protocol can and will be created; however the protocol itself should prove to be a solid foundation.

In addition, many ideas for future enhancements were identified, with the plan that this system will continue being developed.

Bibliography

[Bradley, 2000] BRADLEY, NEIL. (2000) *The XML Companion*. 2nd Ed. Addison-Wesley.

[Dublin Core, 1999] Dublin Core Metadata Initiative. (1999) *The Dublin Core Element Set Version 1.1*. `http://dublincore.org/documents/1999/07/02/dces/`

[Gamma, 1995] GAMMA, ERICH, HELM, R, JOHNSON, R, VLISSIDES, J. (1995). *Design Patterns: elements of reusable object oriented software.* Addison-Wesley.

[Kidd, 2001] KIDD, ERIC. (2001) *XML-RPC Howto.* `http://xmlrpc-c.sourceforge.net/xmlrpc-howto/xmlrpc-howto.html`

[RFC 2616] The Internet Society. (1999) *RFC 2616 - Hypertext Transfer Protocol -HTTP/1.1.* `http://www.w3.org/Protocols/rfc2616/rfc2616.html`

[UUID] The Open Group (1997) *DCE 1.1: Remote Procedure call - Universal Unique Identifier.* `http://www.opengroup.org/onlinepubs/9629399/apdxa.htm`

[Quin, 2000] QUIN, LIAM. (2000) *Open Source XML Database Toolkit: resources and techniques for improved development***.** John Wiley & Sons.

[Sherman, 1999] SHERMAN, CHRIS. (1999) *A Bookmark Manager Roundup*. ONLINE Magazine `http://www.onlineinc.com/onlinemag/OL1999/sherman9.html`

[SOAP 1.1] W3 Consortium. (2000) *Simple Object Access Protocol (SOAP) 1.1.* `http://www.w3.org/TR/SOAP/`

[XML 1.0] W3 Consortium. (2000) *Extensible Markup Language (XML) 1.0.* `http://www.w3.org/TR/REC-xml`

[XQuery] W3 Consortium. (2001) *XQuery: A Query Language for XML (Working draft).* `http://www.w3.org/TR/xquery/`

[Winer, 1998] WINER, DAVE. (1998) *XML-RPC for Newbies* `http://davenet.userland.com/1998/07/14/xmlRpcForNewbies`

[XML-RPC] WINER, DAVE. (1999*) XML-RPC Specification.* `http://www.xmlrpc.com/spec`

[OPML 1.0] WINER, DAVE. (2000) *OPML 1.0 Specification.* `http://www.opml.org/spec`

[Winer, 2001] WINER, DAVE. (2001) *OPML 1.0 - Sharing ideas in a safe way,* XML Magazine, Volume 1, Number 5 (Winter 2001), pp68-69.

Appendix A

Export DTD

```
<!--
  Scraps DTD
  (C) 2000 Dave O'Neill <dmo@acm.org>
  Licensed under the GPL.  See LICENSE file for details.
  -->
<!--
  A scrapbook can hold zero or more scraps.  An empty scrap-
book is allowed
  as valid.
  -->
<!ELEMENT scrapbook (scrap*) >
<!-- Basic unit is the 'scrap'.
     Scraps have mandatory and optional data, as seen below.
         title, creator, description and data are all mandatory.
     There must be at least one key-
word and one date (should be 'created'
     if there's only one)
     There can be zero or more contributors as well.
  -->
<!ELEMENT scrap
    (title, creator, contributor*, description, key-
word+, date+, data )
>
<!--
     A scrap has only one attribute, and it is required.
     Unfortunately due to a con-
straint in the XML ID type, we need to make
     our 'id' be of type CDATA instead of type ID, as XML identi-
fiers
     can't begin with a digit.
  -->
<!ATTLIST scrap id CDATA #REQUIRED>
<!--
```

```
     A creator element must contain both name and email address.
  -->
<!ELEMENT creator (name,email)>
<!--
     A contributor element must con-
tain name, email, and date of contribution.
     It may optionally contain a note explaining the contribu-
tion.
  -->
<!ELEMENT contributor (name, email, date, note?)>
<!--
     A date contains one mandatory attribute: type.
     A date's type defines what event the date refers to.
  -->
<!ELEMENT date  (#PCDATA) >
<!ATTLIST date type (created|modified|accessed|imported) "cre-
ated" >
<!--
     The <data> element is the meat-and-
potatoes of the system. Currently, it takes
     PCDATA and has one attribute, the type.
     The type attribute currently can refer to three valid types:
        text
            - this is just plain, freeform text.  It can con-
tain anything, including
              markup, though certain unpleasant charac-
ters will be escaped, and all markup
              is currently ignored and presented as-is.
        url
            - this is a URL or URI, meeting the appropri-
ate RFC standards.  Purpose of this
              is to classify URL data for use by web-
browsers that wish to use this system
              for bookmarks.
        query
            - this is a stored query; this is XML text describ-
ing a query that can
              be retrieved and run.
     Future extensions may implement one or more of the follow-
ing type systems:
        - based on MIME types.  Not entirely useful for the ma-
jority of types, though
        -
"xml/{thingy}" where {thingy} is the "flavour" of XML data (ie: DTD/schema type,
          etc.  This should be used in the future when appropri-
ate plugins can be used to
          format certain types of XML data (ie: direc-
tions, restaurant review, etc)
```

```
  -->
<!ELEMENT data (#PCDATA) >
<!ATTLIST data type (text|query|url) "text" >
<!--
    Simple elements with no attributes and no contained ele-
ments (only PCDATA)
  -->
<!ELEMENT title (#PCDATA) >
<!ELEMENT name  (#PCDATA) >
<!ELEMENT email (#PCDATA) >
<!ELEMENT keyword (#PCDATA) >
<!ELEMENT description (#PCDATA) >
<!ELEMENT note (#PCDATA) >
```

Appendix B

Search Encoding DTD

```
<!--
  Scraps DTD
  (C) 2000 Dave O'Neill <dmo@acm.org>
  Licensed under the GPL.  See LICENSE file for details
  -->
<!--
    <multiquery> can contain multiple <query> elements for
    querying multiple servers
 -->
<!ELEMENT multiquery (query*)>
<!--
    <query> contains an optional <server> tag specify-
ing other servers to
    contact, and a single boolean operator tag.
  -->
<!ELEMENT query (server?,(and|or|not))>
<!--
        <server> should con-
tain an URL, in http://hostname.tld:port/path/ format.
        If we move to XML Schemas in the future, we can spec-
ify this constraint.
  -->
<!ELEMENT server (#PCDATA)>
<!--
    The boolean operators can hold any number of sub-
tags. They function
    as prefix-
notation operators would, ie an <and> containing three tags
    represents the logical AND of all of those search conditions.
    Note that an AND or OR tag can contain multiple ele-
ments, while a
    NOT can only contain one.
  -->
```

```
<!ELEMENT and (and|or|not|keyword|created|modified|accessed|imported)*>
<!ELEMENT or (and|or|not|keyword|created|modified|accessed|imported)*>
<!ELEMENT not (and|or|not|keyword|created|modified|accessed|imported)>

<!--
    Character data representing a keyword.
  -->
<!ELEMENT keyword (#PCDATA)>
<!--
    These elements spec-
ify types of dates to be searched. They hold
    elements defining how that date is to be considered.
-->
<!ELEMENT created (on|before|after)>
<!ELEMENT accessed (on|before|after)>
<!ELEMENT imported (on|before|after)>
<!ELEMENT modified (on|before|after)>
<!--
    These elements hold a single date, and spec-
ify how that date is to
    be searched... either exact match (on), anything previ-
ous(before) or
    anything after.
    Currently, the date format is not de-
fined, but it should be limited
    to:
        - an ISO format long date
        - a character date in the format YYYYMMDDHHMMSS
    as these two formats are sufficiently different so as to be
    recognizable, and they are both human-readable.
  -->
<!ELEMENT on (#PCDATA)>
<!ELEMENT before (#PCDATA)>
<!ELEMENT after (#PCDATA)>
```

Appendix C

Scraps XML-RPC API Specification

This is version 1.0 of the Scraps API.

See `http://www.xmlrpc.com/spec` for details on XML-RPC in general.

General Information

Since XML-RPC is generally stateless, each API call must carry with it its own authentication information. This is done by passing a username and password pair with each call.

If a call results in an error, the error will be returned via the standard XML-RPC <fault> condition, using the error codes defined below and an error-specific message.

API Calls

Note that for all of these calls, INVALID_AUTHENTICATION, INVALID_PERMISSION, or INTERNAL_ERROR may be returned if the call does not complete successfully. In addition to these, each call can return other error codes as appropriate.

scraps.newScrap

```
Signature: struct scraps.newScrap(string username, string pass-
word, struct scrap_data)
```

This call creates a scrap from the given struct of scrap data (see below for description). If the creation is successful and the new scrap can be stored, a struct is returned containing the scrap, including the newly assigned Scrap ID. An INVALID_DATA error may be returned if the scrap data is missing mandatory data fields, or if these fields contain invalid data.

scraps.deleteScrap

Signature: `boolean scraps.deleteScrap(string user-name, string password, string scrap_id)`

This call deletes a scrap and all keywords and contributors associated with this scrap. If the given scrap_id does not identify a scrap in the database, ID_NOT_EXIST is returned.

scraps.fetchScrap

Signature: `struct scraps.fetchScrap(string username, string pass-word, string scrap_id)`

This call takes a Scrap ID as a string argument, and returns a struct containing the specified scrap. An ID_NOT_EXIST error may be returned if the supplied Scrap ID does not exist on this server.

scraps.exportScrap

Signature: `string scraps.exportScrap(string user-name, string password, string scrap_id)`

This call takes a Scrap ID as a string argument, and returns a string containing the scrap data, formatted in well-formed XML according to the DTD defined in Appendix A)

scraps.saveScrap

Signature: `struct scraps.saveScrap(string username, string pass-word, string scrap_id, struct scrap_data)`

This call takes a Scrap ID as a string argument and a struct containing scrap data. The scrap corresponding to the given ID is updated with the data given in the structure. The structure does not need to contain a complete scrap; only the changes need be transmitted. Complete scraps containing an ID may be added using this method as long as the ID does not already exist on this server; otherwise the scrap data may not contain an ID element. An INVALID_DATA error may be returned if the given data is not valid.

scraps.search

Signature: `array scraps.search(string username, string pass-word, struct search_criteria)`

This call takes a struct containing search criteria (see below for description) and returns an array of structs containing metadata on each of the items that match the search criteria. This array of metadata appears “on the wire” as follows”

```
<array><data>
  <value>
    <struct>
      <member>
        <name>id</name>
        <value>
          <string>054fca312df448578fbc2f656488cdf9</string>
        </value>
      </member>
      <member>
        <name>title</name>
        <value>
          <string>This is the scrap title</string>
        </value>
      </member>
      <member>
        <name>description</name>
        <value>
          <string>This scrap contains stuff...</string>
        </value>
      </member>
      <member>
        <name>date_modified</name>
        <value>
          <string>2001-03-27 09:33:53</string>
        </value>
      </member>
  </value>
</data></array>
```

Each additional search "hit" will add a new <value> section to the array, containing a struct with the appropriate metadata.

An INVALID_SEARCH error may be returned if the search criteria are not valid. An unsuccessful, but properly formed search will return an empty array.

scraps.exportSearch

Signature: `array scraps.exportSearch(string user-name, string password, struct search_criteria)`

This call takes a struct containing search criteria (see below for description) and returns a string containing the full scrap data for all search results, formatted in well-formed XML according to the DTD defined in Appendix A.

An INVALID_SEARCH error may be returned if the search criteria are not valid. An unsuccessful, but properly formed search will simply return an empty <scrapbook> tag.

scraps.import

```
Signature: array scraps.import(string username, string pass-
word, string import_xml)
```

This call takes a string containing valid XML conforming to the Scraps DTD and adds the scraps described therein. It returns an array containing structs with two string elements:

- The scrap ID
- The scrap title
- The status of that scrap. Status may be “added”,”exists”, “invalid”

Scrap ID’s of all the scraps successfully imported. An INVALID_IMPORT error may be returned if the input XML is not properly formed. Note that individual scraps in the import XML may not be imported for one reason or another. This fact will be noted for that scrap in the array returned.

Scrap Struct

The scrap struct as returned by the above calls closely resembles the DTD structure. When submitted through newScrap, only the mandatory fields must be included. When submitted through saveScrap, only fields that should be changed on the server need be included. When being returned to the client, all fields for which data exists on the server are included. Mandatory fields for the API are the same fields marked as mandatory elements in the DTD structure in Appendix A.

This is the structure of a scrap, as seen “on the wire” in XML-RPC transport encoding. The appearance of the structure to your client application will vary based on the XML-RPC implementation (for example, using Frontier::RPC in Perl, the structures are represented as Perl hashes). In the interest of brevity (not that it helps much here), the sample below does not include the date entries “created”, “imported” or “accessed”, nor does it include multiple contributors.

```
<struct>
  <member>
    <name>data</name>
    <value>
      <struct>
        <member>
          <name>type</name>
          <value>
            <string>text</string>
          </value>
        </member>
        <member>
          <name>data</name>
          <value>
            <string>This is the data</string>
          </value>
```

```
      </member>
    </struct>
  </value>
</member>
<member>
  <name>description</name>
  <value>
    <string>This is a description</string>
  </value>
</member>
<member>
  <name>creator</name>
  <value>
    <struct>
      <member>
        <name>email</name>
        <value>
          <string>creator@hostname.tld</string>
        </value>
      </member>
      <member>
        <name>name</name>
        <value>
          <string>The Creator</string>
        </value>
      </member>
    </struct>
  </value>
</member>
<member>
  <name>title</name>
  <value>
    <string>Foo Scrap</string>
  </value>
</member>
<member>
  <name>date</name>
  <value><struct>
    <member>
      <name>modified</name>
      <value>
        <string>2001-03-22 18:23:22 EST</string>
      </value>
    </member>
  </struct></value>
</member>
<member>
  <name>contributor</name>
```

```
    <value>
      <array><data>
        <value>
          <struct>
            <member>
              <name>name</name>
              <value>
                <string>A. Contributor</string>
              </value>
            </member>
            <member>
              <name>date</name>
              <value>
                <string>2001-03-22 18:23:22 EST</string>
              </value>
            </member>
            <member>
              <name>email</name>
              <value>
                <string>contrib@MyHost.tld</string>
              </value>
            </member>
            <member>
              <name>note</name>
              <value>
                <string>Change made to this scrap</string>
              </value>
            </member>
          </struct>
        </value>
      </data></array>
    </value>
  </member>
  <member>
    <name>keywords</name>
    <value>
      <array><data>
        <value><string>word1</string></value>
        <value><string>word2</string></value>
        <value>
          <string>compound keyword</string>
        </value>
      </data></array>
    </value>
  </member>
</struct>
```

Search Struct

The search struct required by the scraps.search and scraps.exportSearch calls is a representation of the extensible query structure described above in section 3.3. This struct follows roughly the same structure as the DTD described in Appendix B, but it is encoded as an XML-RPC struct, rather than in its straight XML format, as this makes it easier to work with as an argument to an XML-RPC call.

Error Codes

These are the error codes used for <fault> returns from an API call. The numbers were chosen so as to not conflict with valid HTTP status codes (as per [RFC 2616]), as these may also be returned in an XML-RPC <fault> response.

700 SUCCESS

The command completed successfully. This is typically not returned to the client, but is used internally, as a SUCCESS response is never used in the context of an XML-RPC <fault>

701 INVALID_AUTHENTICATION

The supplied authentication credentials could not be validated.

702 INVALID_PERMISSION

The authenticated user does not have permission to perform that call. Note that this is distinct from the “403 Forbidden” HTTP protocol error message.

703 INVALID_DATA

One or more of the data parameters supplied for that call are not valid. The string accompanying this message should further explain why the data is invalid.

704 INVALID_SEARCH

The search data supplied is not valid. The string accompanying this message should further explain why the search is invalid.

705 ID_NOT_EXIST

The requested Scrap ID does not exist on this server.

706 COMMAND_NOT_IMPLEMENTED

The requested call is not implemented on this server.

707 INTERNAL_ERROR

An internal error has occurred. The string accompanying this message should further explain the error.

708 INVALID_IMPORT

The data given for import was not valid. The string accompanying this message should further explain the error.

709 ID_EXISTS

An attempt to add or import data has been made that conflicts with an existing scrap ID.

Appendix D

Example XML Export Document

The sample document below provides an example of several types of scrap, as they would appear when exported from the system.

```
<?xml version="1.0"?>
<scrapbook>
<scrap id="81779c77299843a98f7dfcacb7a4a0c2">
   <title>Directions to Dave’s house</title>
   <creator>
      <name>Dave O’Neill</name>
      <email>dmo@acm.org</email>
   </creator>
   <contributor>
      <name>Dave O’Neill</name>
      <email>dmo@acm.org</email>
      <date>2001-03-05 01:44:40</date>
      <note>Initial entry</note>
   </contributor>
   <contributor>
      <name>Dave O’Neill</name>
      <email>dmo@acm.org</email>
      <date>2001-03-05 01:48:03</date>
      <note>Spelling corrections</note>
   </contributor>
   <contributor>
      <name>Dave</name>
      <email>dmo@acm.org</email>
      <date>2001-04-15 17:22:04</date>
      <note>new keywords</note>
   </contributor>
   <description>Directions to Dave O’Neill’s house in Ot-
tawa</description>
   <keyword>dave o’neill</keyword>
```

```
   <keyword>house</keyword>
   <keyword>directions</keyword>
   <date type="modified">2001-04-15 17:22:04</date>
   <date type="accessed">2001-04-16 02:57:51</date>
   <date type="created">2001-02-28 00:00:00</date>
   <data type="text">34 Northview Road, Apt 5.
Going west on Baseline, turn right onto Merivale.
Take a left at the next set of lights, then turn right at the end
of the road.
Continue straight for about 500m, and then turn left.
Continue straight for about 250m, then turn left onto Northview.
   </data>
</scrap>
<scrap id="32440f8bc8f8438f8322700a8e858b21">
   <title>Slashdot.org</title>
   <creator>
      <name>Dave O'Neill</name>
     <email>dmo@acm.org</email>
   </creator>
   <description>A very geeky "news" and discus-
sion site.</description>
   <keyword>slashdot</keyword>
   <keyword>url</keyword>
   <keyword>news</keyword>
   <keyword>tech</keyword>
   <date type="modified">2001-04-05 18:34:09</date>
   <date type="accessed">2001-04-16 02:57:51</date>
   <data type="url">http://www.slashdot.org/</data>
</scrap>
</scrapbook>
```