RMS: A Robust Mail Store and Retrieval System

Peter Thiemann

November 18, 2002

Abstract

RMS is a new mail store and retrieval system which is designed for use in a nomadic setting. It supports a number of unique features:

- unlimited replication of the mail store's contents,
- disconnected operation on individual replicas,
- fully automatic merging of replicas using standard file synchronization tools,
- per-message access capabilities for untrusted external readers,
- access to messages based on sets of message attributes — no hierarchical structure imposed.

We have designed and implemented a single-user prototype of RMS, however, the approach scales to a collaborative setting where many individuals share a common mail store via a server.

1 Introduction

A nomadic computing setting is characterized by mobile computing devices that connect to the network from unpredictable locations and at unpredicatble times. In addition, their equally nomadic users may employ multiple (mobile, desktop, and server) computers to pursue their work. In the absence of permanent, high-speed network connections, nomadic users replicate all or part of their working environment on all these machines. Consequently, file synchronization is an important problem for the user (sufferer?) of such a nomadic setting. After extended periods of disconnected operation, there may be diverging working copies of files on these systems and sometimes considerable effort is required to arrive at a consistent status, again. The growing number of file synchronization tools provides ample evidence for this practice [2], just check

`http://www.google.com/search?q=file+synchronization`

One particularly hairy synchronization story concerns email. Many nomadic users carry around a substantial amount of email messages that are part of their day-to-day work. This collection of email messages is what we call the user's

personal mail store. The problem with the mail store is that users may incorporate mail from different sources (*e.g.*, different mail drops) while connected and then may want to perform extensive reorganizations while disconnected. Unfortunately, this pattern of operations on a mail store may lead to wildly diverging contents that defy the capabilities of available file synchronization tools.

Let's illustrate these problems for two common implementations of a mail store:

- A collection of files in UNIX mbox format [3] with many messages per file: incorporating new messages amounts to appending them to the end of a file `mbox`, say, selection amounts to searching the file, and reorganizing amounts to selecting messages and moving them to another file (in another directory).

- An MH-style [11, 10] mail store is a directory hierarchy with (potentially) message files at each level of the hierarchy. There is one message per file and the files in each directory are numbered consecutively. Incorporating a new message amounts to generating an unused filename (the smallest number greater than all file names presently in use) and storing the message in a designated directory `inbox`, say. The selection operation amounts to a search operation over a directory hierarchy. Reorganization amounts to selecting messages and moving them to another directory (and renaming them to the next number in sequence for the target directory). Reorganization may include packing, which renumbers message files so that gaps in the numbering are removed.

Now imagine a disconnected setting with two replicas, A and B, of the same mail store containing folders `inbox`, `work`, and `inbox.october` and let Joe F. User perform operations on both replicas. Selection operations are harmless because the do not modify the mail store, so we consider a sequence of incorporation and reorganization operations.

1. Reorganize A: move messages 5, 13, 19 from `inbox` to `work`.
2. Reorganize B: move messages 1-10 from `inbox` to `inbox.october`.
3. Incorporate A: Add message X to `inbox`.
4. Incorporate B: Add message Y to `inbox`.

If we now try to synchronize the replicas A and B of the mail store with a file synchronization tool, then we are in trouble.

- In the mbox setting, the files `inbox`, `work`, and `inbox.october` are different in both replicas. The files `work` and `inbox.october` have only changed in one replica, so the synchronizer will replace the obsolete versions. However, for `inbox` a diff-based file merger is required and it will probably suggest to collect all messages except message 5 in the file.

  Things get worse if the mailbox format is not a plain text file, but we leave that to the reader's imagination.

- In the MH setting, let's start out with just the two reorganizations. The file synchronizer will happily detect that 5, 13, 19 are deleted in one copy and 1-10 are deleted in the other copy and will delete 13 and 19 in B and 1-4, 6-10 in A. Again, the additional message in `work` and `inbox.october` do not cause problems. However, the warm feeling fades away if we also consider message incorporation: To incorporate a new message, each copy has determined the highest message number m in the `inbox` directory and stored the new message as file $m+1$. Hence, A and B both have a message named $m+1$ with different content and the file merger cannot operate on them in a useful manner.

  Actually, we have simplified the reorganization bit. With MH, reorganization may involve renumbering (packing) the message files in a directory. In that case the message numbers become messed up and the file merger cannot be usefully applied to the message directory.

Our conclusion from the above scenario is that current implementations of mail stores are not adequate in a nomadic setting with replication. In fact, not much attention is paid to implementing mail store functionality as an isolated mechanism. It is usually added as a second thought to mail user agents. The exception here is the IMAP approach which provides mail store functionality. Unfortunately, IMAP only works as long as there is a network connection to the IMAP server and it depends on the particular implementation whether IMAP folders can be replicated.

Hence, we have designed and implemented RMS, the robust mail store, starting from the following requirements:[1]

1. Functional requirements (operations on mail stores)
   - incorporate new messages;
   - select and retrieve messages;
   - reorganize the mail store;
   - extract personal mail store (*e.g.*, a replica for use on the laptop or for archiving);
   - integrate a personal mail store.
2. Non-functional requirements
   - strong support for replication:
     - all operations work in disconnected mode on personal mail stores;
     - personal mail stores automatically kept in synch with standard file synchronization tools;
   - access to messages based on attributes, not on accidental hierarchical structure;

[1] Interestingly, archiving is virtually unsupported with existing implementations of mail stores.

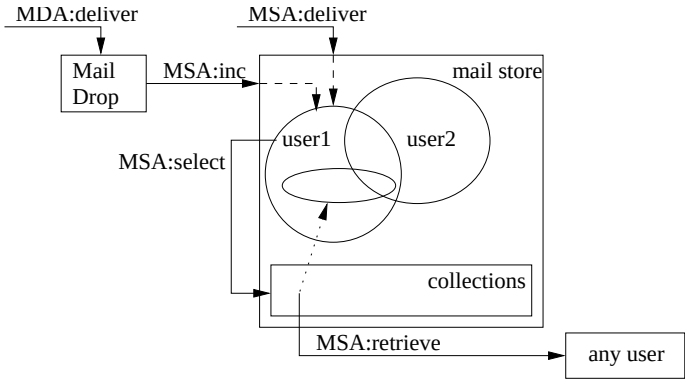

Figure 1: Structure of RMS

- multi-user collaborative operation.

To meet these goals we have employed the following principles:

- strict separation of message data and meta data;
- all data pertaining to a message is stored under a globally unique name constructed from a cryptographic hash of the message's contents;
- message data is considered immutable, all organizational attributes are kept in (mutable) meta data.

Our design resulted in a further useful feature. Any registered user can create a message collection from the messages visible to him in RMS. For each message collection, RMS produces a capability that enables external users to access (only) the messages in the collection without further authenication.

2 Concepts

In this section, we explain the basic concepts underlying our design and justify the design decisions taken. The mail store keeps track of three kinds of data: message data, meta data, and message collections. Of these, only the meta data is mutable, whereas message data and message collections are immutable. Data relating to a single message (the message and its meta data) is content-addressed via its globally unique message digest (GUMD). Message collections are addressed via randomly generated names.

Figure 1 gives an overview of the structure of RMS. All arcs labeled with MSA:... reflect operations of RMS. The ellipses within the store indicate the subset of messages visible to user1 and user2, respectively. A message M is visible to a user if the user incorporated M or if the user holds a message collection that contains M. The flat ellipse indicates a message collection formed by user1 using an select operation.

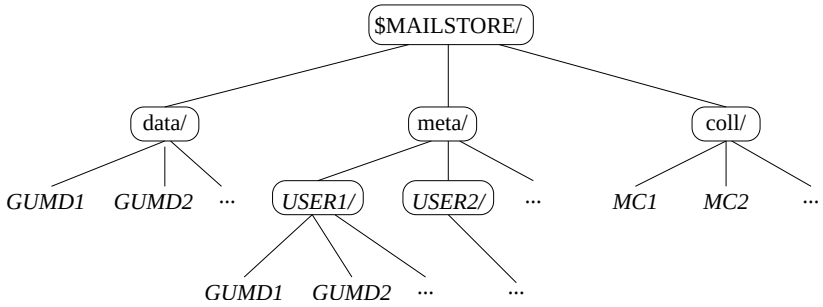

Figure 2: Directory Structure of the Implementation

2.1 Globally Unique Message Digest

A globally unique message digest (GUMD) contains two parts:

1. the contents of the `Date:` header field of the message as a 64 bit unsigned integer expressing the number of seconds from the epoch (Jan 1, 1970, 00:00:00, UTC) to that date and
2. a cryptographic hash of the message.

The GUMD serves a number of purposes.

1. It serves as a unique key for accessing the message data and its meta data. Uniqueness is due to the combination of the time stamp and the properties of a cryptographic hash function, which is conjectured to have a very low probability of collision.
2. If a mail store is shared by a group of users, then email messages are automatically shared without the individuals noticing it. A mail store-aware mail delivery agent may take advantage of that from the start.
3. The time stamp in the GUMD provides an efficient means of selecting and sorting messages by date (*e.g.*, for displaying or for archiving them) without accessing the message's data itself.

The interaction between items 1 and 2 raises a subtle privacy concern discussed in Sec. 5.2.

2.2 Data and Meta Data

All information about a message is stored in one data file and one meta data file for each user who holds on to this message. All data files are stored in one directory, whereas meta data files are stored in one directory for each user. All files pertaining to one message have the same name (the GUMD).[2] The

[2] An alternative implementation might store all meta data files in a database, but that complicates the task of the file synchronizer.

data file contains the raw message as the mail delivery agent placed it in the mail drop. Its contents never change. Each meta data file contains a set of user-specific message attributes that may change over time. See Fig. 2 for a directory structure suitable for implementing the mail store.

In our implementation, the content attributes are listed in the meta data files. For example, a file containing

```
read closed RMAIL
```

is stored in `$MAILSTORE/meta/thiemann/` with the GUMD of the corresponding message as its file name. The attributes `read` and `closed` are specific to my mail user agent (the message has been read and the subject is closed), whereas `RMAIL` indicates that the message was incorporated from an `RMAIL` file.

The separation of the data from the meta data has a number of interesting consequences that we examine in subsequent subsections.

2.2.1 Efficient Selection and Reorganization

All selection operations that do not examine the actual contents of the messages need only access the meta data. Selection by date does not even require opening the meta data file.

Reorganization operations *never* require moving or changing the data file. Only the content attributes in the meta data are ever changed.

2.2.2 Straightforward Archiving

The archival state of a message is simply indicated by the presence or absence of the data file and meta data file.

- If both are present, then the message is live.
- If only the meta data is present, then the message is archived and may be present in another mail store or on a backup media.
- If only the data file is present, then the message has been removed from the system and the data file can be garbage collected.

2.2.3 Interaction with File Synchronizers

File synchronization of a personal mail store using the directory structure from Fig. 2 can be fully automated. We assume a file synchronizer that propagates the newer file if only one replica has changed and invokes a merge program in the uncommon case where both replicas have changed [2].

Nothing special needs to be done for data files. If a file is only present in one replica it is propagated according to the file synchronization rules (it is either copied or removed). If a particular data file is present in both replicas of the mail store, then the files are identical by construction of the mail store so *no merge operation is ever attempted on a data file.*

For meta data the situation is slightly different since meta data is mutable. Again, the only problematic case occurs when the meta data must be merged due to changes in both replicas. However, merging meta data is not a problem for the following reasons.

- Meta data is textual and small and consists of user defined attributes.
- Meta data can be merged automatically by taking the union of the attributes. This is always safe, in the sense that a message only looses an attribute if it has been removed consistently in both replicas.

2.2.4 Splitting off Personal Mail Stores

In a collaborative setting, the directory structure of the mail store will be hidden on a server so that file synchronization is not possible. For that reason, the server provides operations that allow a user to

- replicate part of the mail store that contains (some or all of) the user's messages and the user's meta data;
- remove all messages concerning the user from the system (by removing the user's meta data and then garbage collecting); and
- integrate a private replica back into the mail store while preserving modifications to meta data that have been performed on either replica.

These facilities alleviate moving from one organizational structure to another (*e.g.* changing jobs) as well as reorganizing mailboxes while disconnected (*e.g.* on a plane, in a meeting, or in front of the fire place).

2.3 Message Collections

Each selection operation creates a message collection in the mail store. Intuitively, a message collection is a list of GUMDs of the selected messages. It is identified solely by its name.

It is not possible to retreive individual messages directly (see Sec. 5.1 for an explanation of this decision), all retrivals are indirect via a message collection and its name serves as a capability for accessing its messages. Using the name in that way requires it to be unguessable. Hence, the name is generated using a strong random number generator.

For illustration, consider the following message collection:

$$\texttt{MC-name} \mapsto (\texttt{GUMD}_1, \ldots \texttt{GUMD}_n)$$

Just asking the access interface for `MC-name` returns the number of messages in the collection n. Then, individual messages can be accessed using

$$\texttt{MC-name}/i$$

(where i ranges from 1 through n) without exposing the $\texttt{GUMD}_i$.

3 Operations

The operations on a mail store split naturally in operations that only affect meta data, operations on messages, and operations on personal mail stores.

3.1 Operations on Meta Data

Changing the organization of the mail store is done solely by rewriting the attributes (for one particular user). Since the meta data is considered as a set of attributes, the generic operation on it is subset replacement:

`replace` (x_1 ... x_n) (y_1 ... y_m) MC

The meaning of this operation is the following: For each message in MC, replace its attribute set A by $(A \setminus \{x_1, \ldots, x_n\}) \cup \{y_1, \ldots, y_m\}$.

There is an operation to remove the messages in collection MC. It only remove the present user's meta data. The actual message data is unaffected and remains in the store.

`remove` MC

RMS does not have to manage access rights since the shared data is read only (the message itself). To grant another user access to a message collection it is sufficient to pass that collection to the user.

3.2 Operations on Messages

Messages can be incorporated into the mail store or retrieved from it. The command

`incorporate` (y_1 ... y_m) *`messages`*

incorporates the *`messages`* into the mail store and attaches the attributes $y_1, \ldots, y_m$ to them. If a message already exists, then only its attributes are updated to contain $y_1, \ldots, y_m$.

The command

`retrieve` MC/i

retrieves the raw contents of the ith message in collection MC.

3.3 Operations on Personal Mail Stores

The operations on mail stores are replication and integration. The replication command

`replicate` MC

constructs a personal mail store[3] from a message collection MC. The new personal mail store contains all messages from MC and all of the user's meta data pertaining to these messages. The new mail store *does not* contain any message collections.

Finally, the command

`integrate` PMS

incorporates all messages from personal mail store PMS and merges their attributes as described in Sec. 2.2.3. Furthermore, all message collections are copied from PMS. This command cannot be simulated using `incorporate` because it treats attributes differently and because it also transfers message collections.

The asymmetric treatment of message collections is again driven by privacy concerns: Since the personal mail store is created as a directory structure, its owner can freely examine its contents. If the `replicate` command were to copy message collections, too, then the privacy of users of the originating mail store could be compromised because message collections may be present that were not meant for the owner to see. Hence, the replica should at most contain those message collections that the owner knows about, anyway. However, RMS does not control access to message collections, so it has no data to make this decision. Consequently, it does not replicate message collections.

The integration of message collections from a personal mail store is not problematic because they are all created by the owner from messages in that very store. To prevent malicious owners from fabricating their own message collections, RMS filters incoming message collections by removing GUMDs that reference messages outside the incoming personal store.

3.4 Selection

The selection operation receives a query and returns a message collection of messages satisfying the query. In a typical mail user agent as well as in IMAP, the main selection criterion (sometimes implicitly) is the specification of a mail folder. The structure of the mail folders is directly reflected in the directory hierarchy in which the messages are stored. In contrast,

RMS does not impose a hierarchical organization structure

although a mail user agent may choose to present it in this way.

The following selection criteria are supported by RMS (ordered from cheap to expensive).

- member of a collection (`--collection` *`MC-name`*); this may be narrowed to a subset of the members by specifying an ascending list of indexes in square brackets (`--collection` *`MC-name`* `[1,5,7]`)
- date (`--earlier-than` *`datespec`*, `--later-than` *`datespec`*)

[3] A personal mail store only admits a single user.

- presence of an attribute (*attribute*)
- regular expressions on header fields (*e.g.*, `--header-to:` *regexp*)
- regular expressions on message body (`--contains` *regexp*)

These criteria can be combined in the style of the Unix `find` command using the boolean operators `!` (negation), juxtaposition (logical and), `-o` (logical or), and bracketing with `(` and `)`.

The selection criteria are sufficiently general to provide the usual functionality of mail user agents for dealing with newly arrived mail. For example, the program that moves messages from the mail drop to the mail store may insert designated attributes `unread` and `open`, meaning that the message has not been read and that it belongs to an open transaction.

4 Implementation

We have implemented a prototype of the RMS system. The incorporation of a message in the mail store is implemented in C using Bernstein's mess822 library [5] and Peter Deutsch's implementation of MD5. This program is used by a number of shell scripts that convert various mailbox formats into RMS.

All operations of the mail store are implemented as command-line programs in Haskell [9]. Additionally, there is a WWW-based mail access utility that allows external access to a message collection. It is implemented with the WASH/CGI library [16, 18].

Our implementation of GUMDs represents the date by a 20 digit decimal number followed by a hyphen and the cryptographic hash is the MD5 message digest of the message body represented as a 32 digit hex number. For example:

```
00000000000685613056-3b50c7b997bc57cf90b7bb328cfc35fb
```

Regarding the probability of collision for MD5 *it is conjectured that the difficulty of coming up with two messages having the same message digest is on the order of* 2^{64} *operations, and that the difficulty of coming up with any message having a given message digest is on the order of* 2^{128} *operations* [13]. Given the time stamp, it follows that messages need to be generated at a rate of 2^{64} *per second* to create colliding GUMDs (but see Sec. 5.2 for further discussion).

The `Message-ID` header of a message was deemed unsuitable for uniquely identifying messages although RFC2822 demands that the message identifier is globally unique [12]. However, it requires that we trust the host generating it to comply with RFC2822.

The name of a message collection consists of the the size of the collection as a 7-digit decimal number, followed by a hyphen, an then universally unique identifier generated by a DCE compliant library [17].

5 Discussion

Our design has yielded a number of benefits beyond the original design goals.

- We can grant any external user read access to messages via message collections by communicating its MC-name and providing (Web-based) access to the mail store via the MC-name.
- Forwarding of messages can be reduced to passing an MC-name.
- The GUMD reliably detects duplicate messages.

5.1 External Access to Messages

Originally, the idea was to allow external users access to individual messages in a mail store directly via the GUMD of a message. However, this was rejected for privacy concerns: Any person that receives a message can check if this particular message is in the mail store, simply by querying the GUMD. If the mail store is attributable to a single person or a small group, then information about recipients of the message may be disclosed (*e.g.*, in case of `BCC`ed messages, `Undisclosed Recipients:` addressing groups).

With the present design, messages can only be accessed through a message collection. Given just a GUMD, there is no way to detect the presence or absence of a message in the mail store. Only an explicit action of a user authorized to access a message can create a collection containing it. Then, in another explicit action, that user may decide to pass the collection's name to somebody else, thus granting access permission.

For housekeeping purposes, it would have been convenient for the name of the message collection to also contain the time when it was created. However, this has been abandoned for privacy concerns because the time when a collection was prepared might also convey information to a malevolent external user.

For similar reasons, the `replicate` operation does not copy collections and `integrate` censors them to be self-contained. Collections are considered server-specific and ephemeral. They may be removed from the server at any time.

5.2 GUMDs and uniqueness

Here is a scenario that creates messages with identical GUMDs. It is based on the observation that messages may have identical bodies, but different message headers, except for the `Date:` header. For that to happen, a message sender needs to send the same message body during the same second to different users sharing one mail store. This can happen easily, if those users subscribe to the same mailing list or if they receive the same spam mailing.

The current design shares these messages. The visible effect is that a number of users receive a message addressed to some user A, who happens to be part of the same bulk mailing. This might be regarded as a breach of privacy since it is possible to discover that another user subscribes to the same mailing list.

There are a number of possible solutions to this problem. One is to remove the header fields that may identify the addressee of the message from the data file and store them in the individual meta-data files. Another is to include those header fields in the computation of the GUMD. At present, neither of these options is implemented.

It should be noted that neither the choice of MD5 nor of any particular way of computing the GUMD from the message is an eternal commitment. In fact, it is possible to switch to other cryptographic hashes (*e.g.*, SHA1 [8] or Hash127 [4]) or to change the way that the hash is applied to the message even while the system is running (also some loss of sharing of messages might be incurred unless all messages are translated to the new hashing scheme). RMS relies only on the properties of the GUMD, not on its particular format.

6 Related Work

The maildir format [6] used by qmail implements a mail drop as a directory where each message is stored in a uniquely named file. A time stamping method is used to guarantee machine-wide uniqueness. Thus, the maildir format achieves reliability (no concurrent write operations on a single maildrop file: no locks requried) and speed of delivery (no costly append operations and no reading of the entire directory to figure out the next file name). In contrast, we are using *globally* unique message names and our objectives for using them are to support merging of replicated mail stores via standard file synchronization tools in the presence of arbitrary reorganization.

IMAP is a network protocol for accessing a mail store [7]. Existing IMAP servers include an MH-style implementation of the mail store. In contrast to our approach, IMAP relies on a single centralized mail store which also serves as a mail drop. Disconnected operation, in particular reorganization of the store, is not directly supported. However, there is a memo that *deals with the issues of what might be called the "driver" portion of the synchronization tool: the portion of the code responsible for issuing the correct set of IMAP4 commands to synchronize the disconnected client in the way that is most likely to make the human who uses the disconnected client happy* [1]. In contrast, RMS guarantees automatic successful synchronization. IMAP employs a concept called "Unique Identifier Message Attribute", but these identifiers are message sequence numbers indicating the order in which they were added to the IMAP mailbox and their uniqueness is only guaranteed with respect to one particular mailbox. In contrast, our GUMDs are globally unique and may be transferred between mail stores. In addition, the IMAP protocol assumes a hierarchically structured store (see 6.3.3 of [7]), although an implementation might be free to work differently.

OpenCM [14, 15] is a configuration management system which is based on cryptographic hashes. It uses those hashes to identify immutable objects stored in a repository. The hash value also serves as an external access capability, so it is also possible to externally verify if a certain object is stored in an OpenCM repository. It is not clear if there are similar implication to those discussed

above (Sec. 5.1). For mutable objects, like configurations, OpenCM uses random numbers for name generation. In contrast, we are using random numbers for generating names for collections (which are immutable). Our motivation for doing so is to generate unguessable access capabilities, as in OpenCM.

At the time of writing, we also learned of Compaq SRC's Pachyderm mail system (`http://research.compaq.com/SRC/pachyderm/`) but we were unable to obtain further information about it. Apparently, it also provides non-hierarchical database-style access to its stored messages base. We were not able to determine if Pachyderm requires connection to the server or if it supports disconnected operation.

7 Conclusions

It came to us as a surprise that seemingly conflicting design goals (like support for replication and non-hierarchical organization) worked out so well together. Our implemented prototype fulfills all requirements and proved to be sufficiently efficient for dealing with about 3500 messages taken from the author's email archives (although the mail store operations were implemented without regard to efficiency).

As ongoing future work, we are planning to modify a mail user agent to work directly with RMS. Further, we plan to explore if an IMAP server might use RMS as its underlying storage mechanism. We are currently setting up a Webpage

`http://www.informatik.uni-freiburg.de/~thiemann/MailStore`

that provides access to the implementation and to the Web-based frontend.

References

[1] Rob Austein. Synchronization operations for disconnected imap4 clients. `http://asg.web.cmu.edu/cyrus/rfc/draft-ietf-imap-disc-01.html`, November 1994. Expired Draft.

[2] S. Balasubramaniam and Benjamin C. Pierce. What is a file synchronizer ? In *Proceedings of the 4th Annual ACM/IEEE International Conference on Mobile Computing and Networking (MOBICOM-98)*, pages 98–108, New York, October 25–30 1998. ACM Press.

[3] Daniel J. Bernstein. Mbox — file containing mail messages. `http://www.qmail.org/man/man5/mbox.html`.

[4] Daniel J. Bernstein. Floating-point arithmetic and message authentication. `http://cr.yp.to/papers/hash127.ps`, March 2000.

[5] Daniel J. Bernstein. The mess822 library. `http://cr.yp.to/mess822.html`, February 2000.

[6] Daniel J. Bernstein. Using maildir format. http://cr.yp.to/proto/maildir.html, July 2001.

[7] Mark R. Crispin. Internet message access protocol - version 4rev1. http://www.faqs.org/rfcs/rfc2060.html, December 1996.

[8] Donald E. Eastlake and Paul E. Jones. Us secure hash algorithm 1 (sha1). http://www.faqs.org/rfcs/rfc3174.html, September 2001.

[9] Haskell 98, a non-strict, purely functional language. http://www.haskell.org/definition, December 1998.

[10] The rand mh message handling system uci version 6.8.3. http://www.ics.uci.edu/~mh/, November 2000.

[11] Jerry Peek. *MH & xmh: Email for Users & Programmers.* O'Reilly & Associates, Inc, 1995. Available at http://www.ics.uci.edu/~mh/book/.

[12] Peter W. Resnick. Internet message format. http://www.faqs.org/rfcs/rfc2822.html, April 1992.

[13] Ronald L. Rivest. The MD5 message-digest algorithm. http://www.faqs.org/rfcs/rfc1321.html, April 1992.

[14] Jonathan S. Shapiro. CPCMS: A configuration management system based on cryptographic names. In USENIX, editor, *Proceedings of the FREENIX Track: 2002 USENIX Annual Technical Conference, June 10–15, 2002, Monterey, California, USA*, pages ??–??, Berkeley, CA, USA, 2002. USENIX.

[15] Jonathan S. Shapiro and John Vanderburgh. Access and integrity control in a public-access, high-assurance configuration management system. In USENIX, editor, *Proceedings of the 11th USENIX Security Symposium 2002, August 5–9, 2002, San Francisco, CA*, pages 109–120, Berkeley, CA, USA, 2002. USENIX.

[16] Peter Thiemann. Wash/CGI: Server-side Web scripting with sessions and typed, compositional forms. In *Practical Aspects of Declarative Languages, Proceedings of the Fourth International Workshop, PADL'02*, number 2257 in Lecture Notes in Computer Science, pages 192–208, Portland, OR, USA, January 2002. Springer-Verlag.

[17] Theodore Ts'o. Ext2 file system utilities. http://e2fsprogs.sourceforge.net, nov 2002.

[18] Web authoring system in Haskell (WASH). http://www.informatik.uni-freiburg.de/~thiemann/haskell/WASH, March 2001.