Ubiquitous Mail

Erick Gallesio
Université de Nice - Sophia Antipolis
930 route des Colles, BP 145, F-06903 Sophia Antipolis, Cedex, France
Erick.Gallesio@essi.fr

Manuel Serrano
Inria Sophia Antipolis
2004 route des Lucioles - BP 93 F-06902 Sophia Antipolis, Cedex, France
http://www.inria.fr/mimosa/Manuel.Serrano

ABSTRACT

Bimap is a tool for synchronizing IMAP servers. It enables two or more IMAP mirrored servers to be modified independently and later on, synchronized. Bimap is versatile so, in addition to synchronizing emails, it can be used for filtering and classifying emails. For the sake of the example, the paper shows automatic emails classification and white-listing programmed with Bimap.

Bimap is implemented in Scheme. The most important parts of its implementation are presented in this paper with the intended goal to demonstrate that Scheme is suited for programming tasks that are usually devoted to scripting languages such as Perl or Python. With additional libraries, Scheme enables compact and efficient implementation of this distributed networked application because the main computations that require efficiency are executed in compiled code and only the user configurations are executed in interpreted code.

1. Introduction

Low cost computers, ADSL, and wireless connections have made ubiquitous computing a reality. Because the Internet is now available nearly everywhere on the planet, most of us are nearly permanently connected. Many of us use various computers (maybe, one at home, one at work, and a roaming laptop). All these computers ideally use the same synchronized data. Enforcing this synchronization is not always so easy. Hopefully, some dedicated tools such as Unison [4] allow two replicas of a collection of files and directories to be stored on different hosts, modified separately, and then brought up to date by propagating the changes in each replica to the other. However, as convenient as these tools are for file and directory synchronization, they are of little help when considering email synchronization. In this paper, we address the specific problem of synchronizing email.

Bimap is a tool for synchronizing email. It enables emails to be manipulated from different computers and localizations. A user can read, answer, and delete emails from various computers amongst which some can be momentarily disconnected. Bimap automatically propagates the changes to all these computers. As demonstrated in this paper, synchronizing email is a simple problem of synchronizing lists. Functional languages are therefore candidates of choice for implementing such algorithms. Bimap is implemented in one of them, namely Scheme, our favorite programming language.

The rest of this paper is organized as follows. Section 2 presents the *Internet Message Access Protocol* which constitutes the foundation of Bimap. Section 3 shows the limitations of IMAP and it presents the general system architecture used by Bimap. Section 4 presents the synchronization algorithm and its implementation in Scheme. Section 5 presents *white-listing*, a filtering application implemented with Bimap.

Sixth Workshop on Scheme and Functional Programming. September 24, 2005, Tallinn, Estonia.

2. IMAP

The *Internet Message Access Protocol* (IMAP [1]) allows a client to access and manipulate electronic email messages on a server. It permits manipulation of mailboxes (remote message folders) as it also provides the capability for an offline client to resynchronize with the server.

In contrast with other protocols such as the *Post Office Protocol* (POP3 [3]), IMAP includes operations for creating, deleting, and renaming mailboxes, checking for new messages, permanently removing messages, creating new ones, etc. Hence, IMAP is a perfect match for our email synchronizer architecture. We have implemented a simple library that binds IMAP facilities in Scheme. It is briefly presented in this section.

Messages in IMAP are accessed by the use of numbers. These numbers are either message sequence numbers or unique identifiers. ¿From a programming point of view, the latter are much more convenient than the former. Message sequence numbers evolve as emails are created of deleted. On the other hand, each email is associated with a unique identifying number (UID henceforth) that remains valid during an IMAP session. Note that while UIDs unambiguously refer to messages in a given session, no provision is made by IMAP to make UIDs pervasive, i.e., the UID associated with an email may change from an IMAP session to another. Hence, it is difficult to implement a synchronization mechanism that relies on IMAP's UIDs. As seen in Section 4 our synchronization tool prefers to use the stamps allocated by the senders than IMAP's UIDs. These stamps are extremely likely to be unique and in practice we have never found a collision.

Instead of tediously presenting all the functions composing the library, we present in Figure 1 a typical example that illustrates the most important functions. The example shows an interactive session where a user logs in and browses some folders and emails.

Note that contrary to some other protocols, IMAP is stateful. The state of an IMAP connection describes the current directory. The command `imap-folder-select` sets it to a new value.

In addition to the functions used in the example, the library offers various functions for accessing the header elements, the attributes, and the body of the messages.

```
;; establish an ssl connection with the IMAP server
(define sock (make-ssl-client-socket "imap.nohwere.org" 993))
;; get logged in with user name and password
(imap-login sock "john doe" "eodjohn")
;; get the folders list
(imap-folders sock)
   -> ("INBOX" "INBOX.-Unknown" "INBOX.foo" ...)
;; go into a folder
(imap-folder-select sock "INBOX.foo")
;; get the list of messages in "INBOX.foo"
(imap-folder-uids sock)
   -> (33612 32977 29895 29132 29018 28958 26938 26937 26129)
;; get the message information about one email
(imap-message-info sock 33612)
   -> ((message-id . <429F0564.9040400@foo.com>)
       (date . 02-Jun-2005 15:16:09 +0200)
       (size . 6025) (flags \Seen) (uid . 33612))
;; remove one of these messages
(imap-message-delete! sock 33612)
;; create a new folder
(imap-folder-create! sock "INBOX.bar")
;; create a new message in the folder "INBOX.bar"
(imap-message-create! sock "INBOX.bar" "subject: foo\n\nFoo is not bar")
   -> 32739
;; get the whole header of message 32739
(imap-message-header sock 32739)
   -> '((subject . "foo"))
;; get the message 32739 body
(imap-message-body sock 32739)
   -> "Foo is not bar"
;; copy a mail accross server (usually the servers differ)
(imap-message-copy! sock 29132 sock "INBOX.bar")
;; move it into folder "INBOX.foo"
(imap-message-move! sock 32739 "INBOX.foo")
;; log out
(imap-logout sock)
```

Figure 1. An IMAP session in Scheme

3. The general architecture

IMAP is a distributed platform that allows clients to access email servers. IMAP is a big step in the direction of ubiquitous email because IMAP servers can be accessed by basically any computer connected to the Internet. In the first place, we have thought that one IMAP server would be sufficient to satisfy our needs. We have discovered that it is not. The difficulties are two fold. First we occasionally happen to manage our email using disconnected computers such as in-flight laptops. Even if some IMAP-capable email readers maintain a local copy of the emails and are able to work offline, we are seeking a neutral architecture that does not impose a specific mail reader. Second, we also happen to access our distant incoming IMAP server using Web email clients. These clients directly run on the computer hosting the server, modifying its state. This introduces a de-synchronization between the server and the local state of a disconnected laptop.

Generally, IMAP providers enforce quotas and maximal sizes for attachments. These IMAP servers do not provide enough resources for storing all emails. In consequence some emails have to be moved in local folders (e.g., user file system). The location and the format used for these locally stored emails highly depend on email readers. So, they are difficult to synchronize with general synchronization tools.

We came up with a solution that uses several IMAP servers: a main server (or *incoming server*) where the emails first arrive and one local server per computer. All the email clients only access the IMAP server that runs locally. Any modification to an email (deletion, access, ...) is thus local. On a regular basis all the servers are synchronized which maintains a coherent global state. Note that the emails on the *incoming server*, which is generally on a machine which is not under our control, can be a subset of the emails that are stored on our personal computers.

The Figure 2 illustrates a possible use of Bimap synchronization. For the sake of the example let's assume two servers receiving email (*provider1.edu* and *provider2.com*). Both servers only offer IMAP accounts. Let's also assume three machines used to read email (*Classic*, *Desktop* and *Laptop*). *Classic* is not synchronized. It remotely accesses *provider1.edu* and *provider2.com* via a direct IMAP connection. *Desktop* and *Laptop* have local folders that are synchronized with the servers. These two machines expose a set of folders which is the union of the emails located on the two servers and the local folders. In addition to synchronizing *Desktop* and *Laptop* with *provider1.edu* and *provider2.com*, Bimap is also in charge of synchronizing *Desktop* and *Laptop* together in order to ensure a correct synchronization of the email stored locally (and to back them up on another machine too).

4. The synchronization

In this section we present the synchronization algorithm. First, the principles are exposed. Then the specific parts of the synchronization are presented, as well as a sketchy presentation of their implementation.

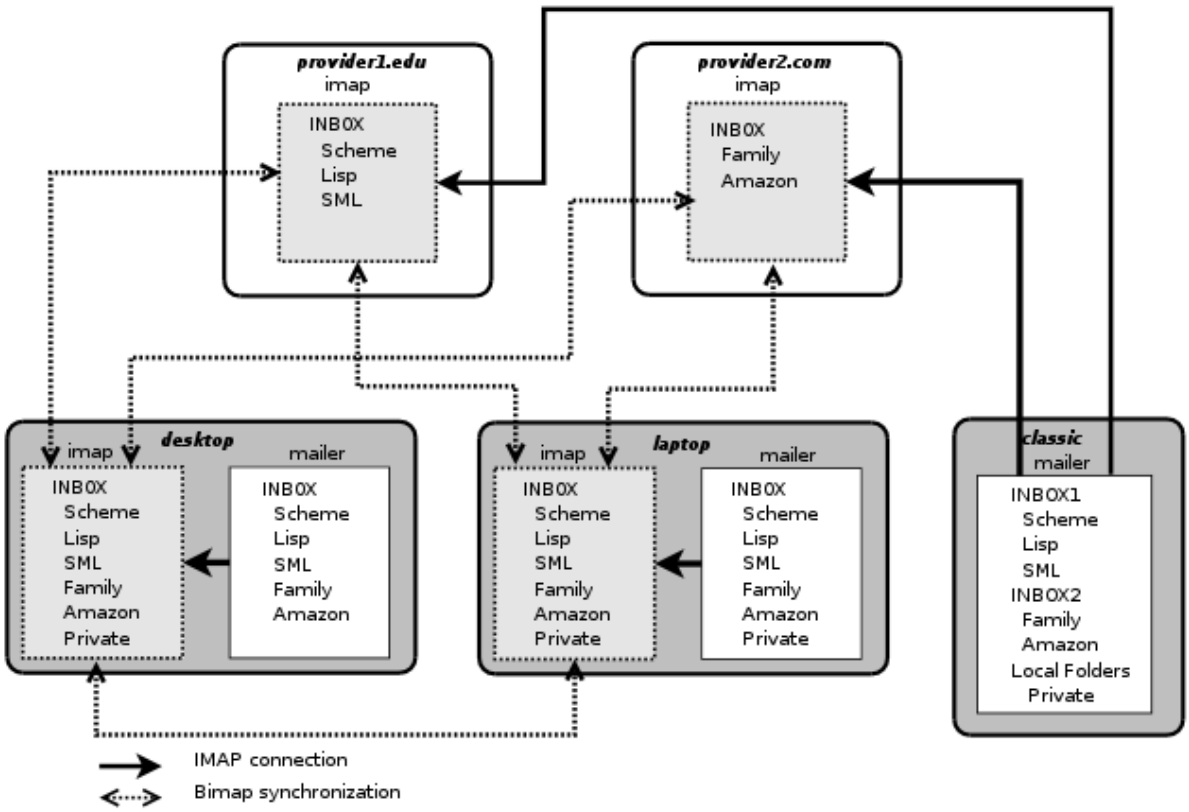

Figure 2. General architecture

4.1 The synchronization principles

The email synchronization relies on the `MESSAGE-ID` header fields [2]. Each of these fields contains a unique identifier which refers to *one* version of *one* email. The uniqueness of the email identifier is guaranteed by the host which generates it. The email identifier pertains to exactly one instantiation of a particular email. In the synchronization process presented here, two emails are considered equal (i.e., the same) if and only if their `MESSAGE-ID` is the same.

`MESSAGE-ID` are not guaranteed to be globally unique hence one could think to replace them with another identification mechanism. As exposed in Section 2, we have decided not to use IMAP's `UID` because they are not pervasive. Alternatively we could have used a checksum mechanism applied to the entire email. This would have had the nasty drawback of enlarged network traffic because the whole email body would need to be transfered for identification. Using the `MESSAGE-ID`, only one field is transfered.

The process of synchronizing email consists in switching from one synchronization state (*sync-state* in short) to another. A sync-state is an abstract notion. In Bimap, it is represented by a sync-table which is made of a list containing the `MESSAGE-ID` of the emails that have been synchronized. When a new synchronization takes place between two servers `S1` and `S2` with a sync-table `SYNCT`, first the lists `L1` and `L2` of emails present in `S1` and `S2` are computed. Then the emails of `L1` are compared to `SYNCT`. An email present in `L1` and absent in `SYNCT` is a new email that has to be copied in `L2` (and vice-versa). An email present in `SYNCT` but absent in either `L1` or `L2` is a deleted email. The implementation of this synchronization is presented in figures 5 and 6.

For the sake of the example, here is an example of a sync-table:

```
(("INBOX"
 ("<OFBF04EB.82F939-O125705F@us.ibm.com>" ((\Seen)))
 ("<1000.4779.9924@gardenia.artisan.com>" ((\Seen)))
 ("<169.62.5348.1708@gardenia.artisan.com>" ((\Seen)))
 ("<11280737.6750.7.camel@localhost.localdomain>" ())
 ("<2508011658.j71GTF002229@sea.inria.fr>" ((\Seen)))))
```

The sync-table is an association list whose `car` is a folder name (`"INBOX"` in this example) and the `cdr` is the list of synchronized messages. For each synchronized message, we retain its `MESSAGE-ID` as well as the list of its IMAP flags. Usage of sync-tables is explained in Section 4.5.

The synchronization only deletes emails that are present in the sync-table. So, if for any reason, such as network failures, the sync-tables are lost, the synchronization algorithm will resurrect emails deleted on only one server but it will never erroneously delete emails. When a message is copied from a server `S1` to a server `S2`, it preserves its `MESSAGE-ID`. Hence, if sync-tables are missing, the synchronization will not erroneously duplicate emails. Consequently, Bimap synchronization is conservative.

4.2 The synchronization state

Instead of being implemented as files in the user file space, sync-tables are stored in the IMAP servers they describe. Hence, users cannot easily modify sync-tables which are private to Bimap. This ensures the coherence of the sync-tables with respect to the servers they describe. In addition, this technique also avoids cluttering user

home space with various private files. Since IMAP servers only contains emails, a sync table is implemented as an email. Since only folders can be named in IMAP, the sync-table is stored as the single message of a well-known folder.

Here is the complete implementation of the sync-table shown in Section 4.1:

```
Subject: bimap Tue Aug 23 06:27:37 2005

(("INBOX"
 ("<0FBF04EB.82F939-0125705F@us.ibm.com>" ((\Seen)))
 ("<1000.4779.9924@gardenia.artisan.com>" ((\Seen)))
 ("<169.62.5348.1708@gardenia.artisan.com>" ((\Seen)))
 ("<11280737.6750.7.camel@localhost.localdomain>" ())
 ("<2508011658.j71GTF002229@sea.inria.fr>" ((\Seen)))))
```

When synchronizing servers `S1` and `S2` which are both referenced by a socket, the name of this folder on `S1` is the name of `S2` concatenated with the user login name on `S1`. The function `sync-folder-name` implements this naming:

```
(define (sync-folder-name s1 s2)
   (let ((n (string-replace
             (socket-hostname s1)
             (string-ref (imap-separator s2) 0)
             #\-)))
      (string-append (bimap-folder-name)
                     (imap-separator s2)
                     n "+" (socket-login s1))))
```

Some special attention has been paid to produce a legal name for the folder name. Since IMAP servers reserve one character (`"."` or `"/"`) as a folder separator, this character cannot be used in folder names. The function `imap-separator` returns the character used on the server. The function `server-sync-table-folder-select` goes into the folder of `S1` containing the sync-table of `S2`.

```
(define (server-sync-table-folder-select s1 s2)
   (let ((f (sync-folder-name s2 s1)))
      (and (folder-exists? s1 f)
           (and (= (folder-length s1 f) 1)
                (imap-folder-select s1 f)))))
```

The function `server-sync-table-get` reads either `S1`'s sync-table for `S2` or `S2`'s sync-table for `S1` if the former is absent.

```
(define (server-sync-table-get s1 s2)
   (cond
      ((server-sync-table-folder-select s1 s2)
       (with-input-from-string
             (imap-message-body
              s1 (car (imap-folder-uids s1)))
          read))
      ((server-sync-table-folder-select s2 s1)
       (with-input-from-string
             (imap-message-body
              s2 (car (imap-folder-uids s2)))
          read))
      (else
       '())))
```

The default case of `server-sync-table-get` is to return an empty sync-table. Remember from Section 4.1 that the algorithm is conservative. That is, if it happens that the sync-table is lost on both servers, the synchronization algorithm will resurrect emails deleted on only one server but in no case will it erroneously delete emails.

4.3 Synchronizing servers

The synchronization of two IMAP servers `S1` and `S2` is parameterized by `folders`, a list of folders to be synchronized. The function `synchronize-servers!`, presented in Figure 3, scans all the folders in the list. For each folder it checks if the folder is new, deleted, or to be synchronized in each server. This function updates the new sync-table in order to reflect the synchronizations that took place.

When all folders are synchronized, the new sync-table is stored on both servers. The function `server-sync-table-store!` accepts three parameters: the socket `S1` and `S2` accessing the servers and the new sync-table `nsync`. It computes the name of the folder where the sync-table on both servers (see `sync-folder-name` in Section 4.2) is to be stored.

When a folder `F` is missing on one server, it has to be determined first if `F` is a freshly created folder or an older one which has been deleted. In order to simplify the understanding of the source code, contrary to the actual implementation, we have duplicated the cases where a folder is either absent on `S1` or `S2`. The code, here duplicated, can easily be merged into a single function. The function `new-folder?` answers this question. A folder is new if at least one of the following conditions is met:

- It is not present in the sync-table (see Figure 4, line *3*).
- It contains sub-folders. Since, synchronization first checks sub-folders, if `F` is old, all its sub-folders would have been previously deleted (line *4*).
- In order to enforce conservativeness, a folder containing new emails (i.e., at least one non-synchronized email) is also considered new (line *5*).

4.4 Synchronizing folders

When a folder is present in both servers (Figure 3, line *25*), each email in this folder is inspected by `synchronize-folders!` defined Figure 5.

The function `sync-table-folders-find`, whose code is not given here, retrieves the information available in the sync-table about the folder `F`. That is, it searches the association list presented in Section 4.1. A hash table is built (line *8*) for improving the performance of the algorithm. It enables fast access to the sync-table.

4.5 Synchronizing messages

The last step of the algorithm is the synchronization of an email. The function `synchronize-message!` synchronizes a message `M1` localized in the folder `F` on server `S1` according to the sync-table `synct`. In addition to copying and deleting emails, this function also propagates the *flags* that are associated with emails. As specified by IMAP these flags denote meta-informations about emails such as *an email is read*, *an email is answered*, ... While not absolutely required, synchronizing flags is important. It enables coherent views of the email on all servers. In order to synchronize flags, Bimap stores the flags of synchronized emails in the sync-tables. That is, for each synchronized email, the sync-table denotes its flags on the servers at the moment of the last synchronization. The function `hashtable-message-flags` returns the flags stored in the sync-table for a given email. It returns either the list of the email flags or `#f` if it is out of synchronization. In the seldom situation where two servers have separately modified the flags of one email, Bimap randomly selects one server for synchronizing flags, loosing the modifications applied on the other server.

5. Filtering and classifying emails

Email has escaped the professional IT sphere. One now emails to colleagues as he does to relatives. Electronic merchandising also generates emails. Electronic billing and confirmation numbers are frequently sent by email. Many administrative procedures can also be completed with the Web and email. All in all this represents a huge number of emails that are sent (and also received) every day.

```
 1: (define (synchronize-servers! s1 s2 folders)
 2:    (let ((sync (server-sync-table-get s1 s2))
 3:          (folders1 (imap-folders s1))
 4:          (folders2 (imap-folders s2)))
 5:       (let loop ((folders folders)
 6:                  (nsync '()))
 7:          (cond
 8:             ((null? folders)
 9:              (server-sync-table-store! s1 s2 nsync))
10:             ((not (memq (car folders) folders1))
11:              ;; folder absent in s1
12:              (let ((f (car folders)))
13:                 (if (new-folder? sync s2 f)
14:                     (begin
15:                        (imap-folder-create! s1 f)
16:                        (let ((s (synchronize-folders! sync s1 s2 f)))
17:                           (loop (cdr folders) (cons s nsync))))
18:                     (begin
19:                        (imap-folder-delete! s2 f)
20:                        (loop (cdr folders) nsync)))))
21:             ((not (memq (car folders) folders2))
22:              ;; folder absent in s2, symmetric to absent in s1
23:              ...)
24:             (else
25:              ;; the folder is present in both servers
26:              (let* ((f (car folders))
27:                     (s (synchronize-folders! sync s1 s2 f)))
28:                 (loop (cdr folders) (cons s nsync))))))))
```

Figure 3. Server synchronization implementation

```
1: (define (new-folder? sync s f)
2:    (let ((dsync (sync-table-folders-find sync f)))
3:       (or (not dsync)
4:           (pair? (imap-subfolders s f))
5:           (any? (lambda (i) (not (assoc (message-id i) dsync)))
6:                 (begin
7:                    (imap-folder-select s f)
8:                    (map imap-message-infos (imap-folder-uids s)))))))
```

Figure 4. Is a folder new?

```
 1: (define (synchronize-folders! sync s1 s2 f)
 2:    ;; go into the synchronized folders
 3:    (imap-folder-select s1 f)
 4:    (imap-folder-select s2 f)
 5:    (let* ((l1 (map imap-message-infos (imap-folder-uids s1)))
 6:           (l2 (map imap-message-infos (imap-folder-uids s2)))
 7:           (fsync (or (sync-table-folders-find sync f) '()))
 8:           (synct (sync->hashtable fsync)))
 9:       ;; synchronize mails
10:       (for-each (lambda (m1)
11:                    (let ((m2 (find-mid (message-id m1) l2)))
12:                       (synchronize-message! synct f m1 s1 m2 s2)))
13:                 l1)
14:       (for-each (lambda (m2)
15:                    (let ((m1 (find-mid (message-id m2) l1)))
16:                       (synchronize-message! synct f m2 s2 m1 s1)))
17:                 l2)
18:       ;; returns a new synchronization state
19:       (let ((fsyncn (hashtable-map synct list)))
20:          (cons f fsyncn))))
```

Figure 5. Folder synchronization implementation

```
 1: (define (synchronize-message! synct f m1 s1 m2 s2)
 2:    (let* ((mid (message-id m1))
 3:           (uid1 (message-uid m1))
 4:           (flags1 (message-flags m1))
 5:           (flags (hashtable-message-flags synct mid)))
 6:       ;; if flags is false (the message is not in the sync table)
 7:       ;; then the message is un-synchronized
 8:       (cond
 9:          ((and (not flags) m2)
10:           ;; an un-synchronized mail, presents in s1 and s2
11:           ;; (e.g., sync-table lost)
12:           (let ((flags2 (message-flags m2))
13:                 (uid2 (message-uid m2)))
14:              (imap-message-flags-change! s2 uid2 flags1)
15:              (hashtable-put! synct mid (list flags1))))
16:          ((not flags)
17:           ;; an un-synchronized mail which does not exists in s2
18:           (imap-message-copy! s1 uid1 s2 f)
19:           (hashtable-put! synct mid (list flags1)))
20:          ((not m2)
21:           ;; a synchronized mail, removed from s2
22:           (imap-message-delete! s1 uid1)
23:           (hashtable-remove! synct mid))
24:          (else
25:           ;; a synchronized mail, present in s1 and s2
26:           ;; when flags differs they have to be synchronized
27:           (synchronize-message-flags! sync m1 m2 s1 s2)))))
```

Figure 6. Message synchronization implementation

Those of us that are used to communicate via the Internet are so overwhelmed by emails that tools are needed for reading, filtering, and classifying emails. In addition to synchronizing email, Bimap can easily be adapted to these tasks, in the spirit of tools such as `procmail`.

Variables declared via the macro `define-parameter` are called *Bimap parameters*. The puspose of `define-parameter` is threefold: it declares a variable, a function named after the parameter that returns the value of the variable, and a function that stores a new value in the variable. Here is an example of a parameter declaration and use:

```
(define-parameter bimap-verbose 0)

(for-each (lambda (o)
             (if (string=? o "-v")
                 (bimap-verbose-set!
                  (+ 1 (bimap-verbose)))))
          (command-line-arguments))
```

When started, Bimap loads a user configuration file that specifies which IMAP servers and folders have to be synchronized. This file can also contain various definitions that are used for email classification and email filtering. Instead of inventing a new little language for implementing these customizations, Scheme augmented with the IMAP binding library is used. In consequence, a Bimap execution uses compiled Scheme code for running the synchronization algorithm and interpreted Scheme code for running the user configuration code. This blending of compilation and interpretation enables high expressiveness without jeopardizing performance.

5.1 Classifying emails

Bimap can be adapted to enable automatic folder selection. Slightly modifying Bimap enables user customizations that automatically deliver incoming emails into dedicated folders. For instance, one may choose to archive emails emitted for a mailing list into dedicated folders or another user may choose to split professional email from personal email. This customization is specified in the new Bimap parameter, `bimap-folder-rewrite`. The value of this parameter is a procedure that accept four parameters: the connection to the IMAP server, the folder in which the email is currently stored, the message info, and its header fields.

Email classification takes place when a new email is detected on only one of two servers. Instead of copying the new email in the folder of synchronization (line *18*, Figure 6) it is copied into a folder whose name is computed by `bimap-folder-rewrite`. That is, line *18* is replaced with:

```
18: (let* ((hd1 (mail-header->list
19:              (imap-message-header s1 uid1)))
20:        (fdest ((bimap-folder-rewrite) s2 f2 m1 hd1)))
21:    (imap-message-copy! s1 uid2 s2 fdest))
```

The new email is copied in the `FDEST` directory (which defaults to `F2`). No other treatment is needed. Since this email is marked as synchronized as any other email, the next time the two servers are synchronized, the message will be moved in `S1` from folder `F1` to `Fdest`.

The following user configuration example illustrates how the parameter `bimap-folder-rewrite` is used to store the emails sent to a mailing list into a dedicated folder.

```
(let ((old-rewrite (bimap-folder-rewrite)))
   (bimap-folder-rewrite-set!
    (lambda (sock folder msg header)
       (let ((to (message-header-field header "to")))
          (if (equal? to "bigloo@sophia.inria.fr")
             "Bigloo"
             (old-rewrite sock folder msg header))))))
```

The email classification requires no extra synchronization treatment. That is, no provision is taken to ensure the synchronization of an email *e* that is stored into a re-written folder *F*. The next time a synchronization takes place, if the folder *F* on the list of synchronized folders, the message *e* will be automatically synchronized too. This framework require no implementation effort but it introduces a delay in synchronization. It takes two server synchro-

nizations to correctly classify such an email and propagate the classification to the servers.

5.2 Surviving Spam

Spam email is a plague. They clutter our mailboxes, threatening email usefulness. Spams are more and more numerous. The Google Gmail accounts of the authors of this paper are cluttered with approximatively 3000 to 6000 spam emails per month. That is, between 100 and 200 spam emails are received each day! The 20 to 30 legitimate emails that are received are literally lost in this ocean of ineptitude. Spam emails are terribly annoying because they are cumbersome, distracting, and polluting. So, it is a popular challenge to stop spams. Many research labs have started projects on this topic. Anti-SPAM software is widely available. The best of these systems do an impressively good job at stopping spams. They use more and more clever methods to decide, according to its content, if an email is spam or not. Bayesian filtering is one of them. Unfortunately, as good as these systems are, as with anti-viruses software, they are bound by their very nature to be late on spam: anti-spam filters cannot anticipate new spamming techniques. Even more pessimistically, we think that content-based filtering is a partial solution that could only produce middling results. What can be reasonably expected from such filters when applied to emails like:

```
 .oooo.o  .ooooo.  oooo    ooo oo.ooooo.
d88(  "8 d88' `88b  `88b..8P'   888' `88b
`"Y88b.  888ooo888    Y888'     888   888
o.  )88b 888    .o  .o8"'88b    888   888
8""888P' `Y8bod8P' o88'   888o  888bod8P'
                                888
                               o888o
```

Spammers can use other techniques for obfuscating emails. A lot of them attempt to fool Bayesian filters by introducing meaningless texts. This ranges from c*h*a*n*g*i*n*g the space character to replacing letters with numb8rs and n0nsense 4ccents. Presumably the most intriguing fooling technique swaps the letters composing the words. Aoccdrnig to rscheearch at Cmabrigde uinervtisy, it deosn't mttaer waht oredr the ltteers in a wrod are, the olny iprmoetnt tihng is taht the frist and lsat ltteres are at the rghit pclae. The rset can be a tatol mses and you can sitll raed it wouthit a porbelm. Tihs is bcuseae we do not raed ervey lteter by itslef but the wrod as a whole.

Content-based filtering is not good enough, it lets too much spams entering our mailboxes. To work around this problem, we have coupled content-based filtering with a more drastic approach: white-listing. This well known technique consists of accepting incoming emails only from authenticated users. We are using a very straightforward technique. We save all the email address of our correspondents into in a big database which compose our white-list. When a new email goes through the content-based filter, the address of the email sender is checked against the white list. If the sender is unknown, the email is moved into a special folder. Otherwise, it is directly delivered to the regular mail box folder. This technique is extremely effective. In our personal setting, white-listing succeeds in detecting 99.9% of spam and only a few legitimate emails go into the spam-dedicated folder. A vast majority of legitimate emails are correctly handled and are no longer lost in a forest of spam. The dedicated folder of unknown senders can be checked once in a while when time permits.

Implementing white-listing exercises email pre-filtering as described above. White-listing is trivial to implement because it only requires a hash table. In the following we assume that the email addresses are stored in the local file `~/.bbdb` and are organized according to the Emacs' Big Brother Data Base format [5]. Since this code takes place in the user configuration file, it can be easily adapted to satisfy everyone's needs.

```
(define *white* (load-bbdb "~/.bbdb"))

(define (unknown-mail? header)
   (not (hashtable-get
         *white* (header-field 'from header))))

(let ((old-filter (bimap-filter)))
  (bimap-filter-set!
   (lambda (sock folder msg header)
     (if (unknown-mail? header)
         (imap-message-move! sock msg "INBOX.-Unknown")
         (old-filter sock folder msg header)))))
```

6. Summary and Conclusion

We have presented Bimap, a tool for synchronizing IMAP servers. We have shown that with very few modifications to the synchronization algorithm, Bimap is also able to filter and classify email. As such, Bimap could be a potential replacement for `procmail`. This is highly convenient because it enables email filtering with simple small Scheme scripts. Two such scripts have been presented: one for classifying emails that belong to mailing lists and a second one for implementing white-listing. Each of these scripts is no more that a few lines of Scheme code.

In order to ease the reading of the present paper, a simplified version of the synchronization algorithm has been presented. Contrary to the code presented here, the actual implementation supports folder re-writing. That is, it enables synchronizing folder `F1` of server `S1` with folder `F2` of server `S2` without imposing name equality between `F1` and `F2`. This is convenient for managing different IMAP accounts intended for different purposes. This incurs a small additional implementation complexity, such as an indexing with two folders name in the sync-table (so the functions `new-folder?`, `synchronize-folders!`, `synchronize-message!`, and `sync-table-folders-find` no longer take only one folder name as parameter but two), but the main principles of the implementation stay the sames.

We are now permanently using Bimap for our own email. We have found that email classification and white listing coupled with Bayesian filtering (that only runs on our incoming email server) is highly effective to filter out nearly all illegitimate emails. Without pretending to have rediscovered the pleasure and excitement of answering our first 80's emails, we claim that Bimap significantly reduces the modern burden of coping with email. We are no longer disturbed by irrelevant emails arriving continuously in our mailboxes. This makes our professional life significantly nicer!

Bimap is not yet the perfect tool. It still needs improvement. In particular, IMAP does not support locking. This is quite unfortunate, because lacking locks makes it impossible to prevent situations where two servers simultaneously attempt to synchronize against a shared third server. In such a situation, inconsistencies might occur in the IMAP responses that cause Bimap to fail. As stated in Section 4.1 this is not critical because the only consequence of corrupted sync-tables is emails resurrection. In no case could it lead to erroneous email deletion.

Bimap is a realistic, yet simple, application written in Scheme. It benefits from the expressiveness of this language and, more importantly, it uses a feature that is frequently available in Scheme implementations: the blending of compiled and interpreted programs. For efficiency, the tree comparisons are compiled. For usability and convenience the user scripts are interpreted. Few other languages present these capabilities.

[1] Crispin, M. – **Internet Message Access Protocol** – RFC 3501, The Internet Society, 2003.
[2] Crocker, D. – **Standard for the format of ARPA Internet text messages** – RFC 822, Dept. of Electrical Engineering, University of Delaware, 1982.
[3] Myers, J. and Rose, M. – **Post Office Protocol - Version 3** – RFC 1939, Carnegie Mellon and Dover Beach Consulting, Inc., 1996.
[4] Pierce, B. and Vouillon, J. – **What's in Unison? A Formal Specification and Reference Implementation of a File Synchronizer** – MS-CIS-03-36, Dept. of Computer and Information Science, University of Pennsylvania, 2004.
[5] Zawinski, J. – **The Insidious Big Brother Database** – 20th century.