Handling Email with a Tcl assist

Clif Flynt
Noumena Corporation,
8888 Black Pine Ln,
Whitmore Lake, MI 48189,
`http://www.noucorp.com`
`clif at noucorp dot com`

September 30, 2010

Abstract

Handling email has become a greater and greater burden on the average computer user. The user must be able to read mail safely, including examining the non-obvious fake mail. There is a need to sort valid from invalid, urgent and non-critical, or just sorting into mail from mailing lists, friends, relatives, etc.

This problem is worse for the systems administrator who also has to deal with the quantities of mail clogging the networks, filling the disks and filling the administrator mailbox with failure notices.

There are many tools to mitigate these problems, ranging from using a webmail service like Gmail, letting Microsoft solve your problems by using Outlook, or using *ix tools like procmail, etc.

All of these tools are written in compiled languages. The speed is good, but the ability to reconfigure and adapt is low.

Tools written in Tcl are as fast as the average user, and are more easily adapted to the ever-changing email environment.

1 Introduction

I've been dealing with email since the early 1980s, when the address `clif!clif` was a unique address, and `ihnp4!clif!clif` would guarantee delivery.

When a big email day was 5 messages, `mail` and `mailx` were great. As time passed, and email volume approached 20 messages a day, `elm` and `pine` were even better.

The graph in **Figure 1** shows the email server activity logged over a period of a few weeks. The black bar is the number of connections made to the server. The dark gray bar is the number that were rejected by hard-coded rules (invalid sender, invalid recipient, etc). The light gray bar is the number of valid messages sent to users.

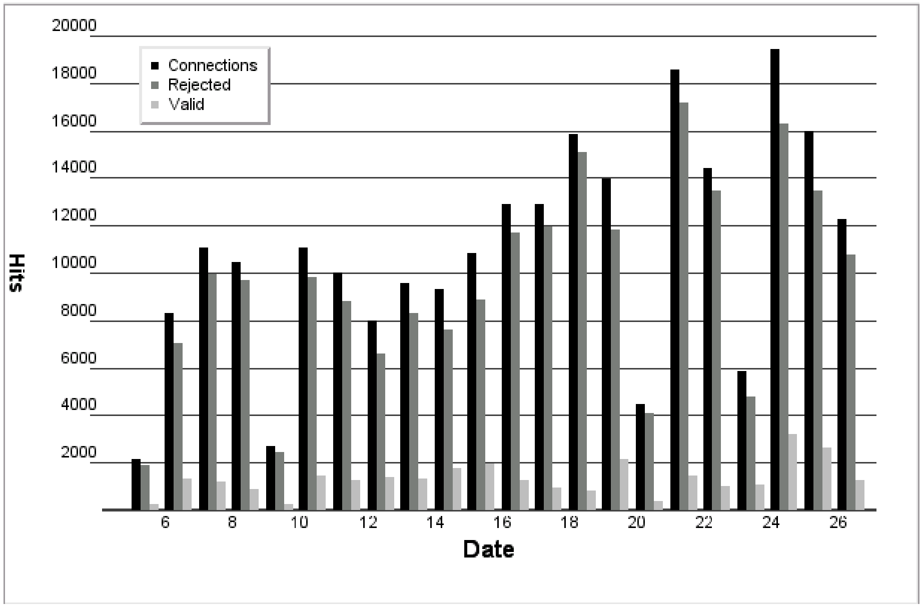


Figure 1: Connections, Rejections and Valid emails

The number of valid messages is much smaller than the number of invalid attempts, but still tops 1000 emails / day for each user on the system.

Obviously, the modern connected user needs tools allow them easy access to potentially hundreds of emails per day and to protect themselves while reading them.

My preference is to host my mail server and read mail locally on a *ix system.

This paper will discuss tools that I use to handle the volumes of mail I receive, avoid viruses, avoid spam and control the system.

2 Reading Mail

I happen to like the `mutt` mailreader for general use. It provides a simple user interface, handles MIME attachments and doesn't put too much of its own philosophy on how I read mail.

However, it does become awkward when receiving over 200 emails per day, spread across several mailing lists.

One feature of `mutt` and other text based mail readers is that they will read any file, as long as it is in the standard Unix mail format. This hook allows sorting mail into multiple mailboxes and then reading the email via mutt with a command like `mutt -f tcl_core_list`.

2.1 Trivial solutions aren't

My presorting application started life as an application for a client who needed a simple email application to accept email, parse it for simple commands like ″send info″ and generate canned responses. The client wanted something cheap and fast, so it was necessary to use tools I knew would work.

It seems simple to open the Unix mail file, read in the data, process it and clear the file.

This isn't quite the case.

The system may be putting new mail into the mail file while your application is emptying it for instance. Your application needs to be careful about understanding how the file locking is implemented. The existing email readers already have locking code that works.

Having been warned that the `procmail` developers went through several iterations before they got the locking right, I side-stepped the issue by using the traditional `mail` text mail reader and `expect` to read and process the mail.

The `mail` mail reader is a first generation mail reader designed for use on a printing terminal. It has single letter commands to find out how many messages exist, read a message and delete a message.

An expect script to read email looks a bit like this:

```
spawn mail
# First message resembles:
# "/var/spool/mail/clif": 10 messages 6 unread
 # Look for the : and a set of numbers.
 expect {
    -re {: *([0-9]+) } {
          set messageCount $expect_out(1,string)
        }
 }
 # Step through messages
 for {set i 1} {$i <= $messageCount} {incr i} {
   exp_send "$i\n"
   # Read message
   # Process message
 }
```

In the real world it gets a bit more complex.

- Messages get large

  The `mail` application it pure text. If a friend sends you a video, `expect` will need to read several megabytes of text. This will overflow the `expect` input buffer unless you use the `match_max` command to reset the buffer size. I'm currently using 4 million as the buffer size.

- Large messages take a long time to be read

Using mail as a front end means that text is flowing through stdin and stdout. Pushing a few megabytes through standard I/O can take time. The expect command set timeout will increase the length of the timeout interval. This is currently set for 500 seconds.

- Trivial parsing won't work.

  There are dozens of email systems out there. I doubt that any of them conform to the complete RFC standard. For example, an email address may be reported in several ways including:

  ```
  From: Your Friend <friend@spamsite.com>
  From: Facebook <notification+o64s0y26@facebookmail.com>
  From: User Name [mailto:username@roadrunner.com]
  From: Local User
  From: remoteuser@att.net
  ```

  The parsing code needs to understand all of them, and perhaps a few more. (Note that the + format means disregard the part of the email address between + and @ when delivering email.)

  Tcl's regexp command returns a 1 if it matches a pattern in the string, and a 0 if the string does not match a pattern. Assuming that an email address will not include the illegal characters you can use a regexp to identify the address portion from one of those strings.

  ```
  if {[regexp {([^!^;^:^ ^\[^<]+@[^:^ ^>]*)} $string m1 addr] == 0} {
    # Must be local address
    set addr $string
  }
  ```

2.2 Out of sorts

The main functionality needed from a mail sorting program is to sort the mail into different known bins. The mail reading program starts by parsing the headers into a (relatively) consistent format and storing the data in a Tcl array where the index is a header field.

```
% parray Parts
Parts(from)    = client@customer.com
Parts(subject) = New Project
Parts(to)      = clif@noucorp.com
```

This data structure makes it easy to write a set of rules that looks like this:

```
set Actions {
 { reply-to {
   {{[firstNsave ltsp-discuss@lists.sourceforge.net %s LTSP]} {}}
```

```
  {{[firstNsave googlegroups %s GSoC]} {}}
  {{[firstNsave -nextgen@noucorp.c %s tk-nextgen]} {}}
}
}
{ sender {
  {{[firstNsave ev-bounces@lists.sjsu.edu %s ev]} {}}
  {{[firstNsave tcl-webmaste %s TclWeb]} {}}
  {{[firstNsave snort-users %s snort]} {}}
  {{[firstNsave starkit %s starkit]} {}}
}
}
```

The rule engine steps through rules in the order they appear until one succeeds. The `firstNsave` procedure examines the provided string and if a pattern is matched, saves the message and returns a true.

This started life as a trivial application. At this point, it's become about 700 lines of Tcl code sorting on about 80 criteria. It runs fast enough to keep up with the mail stream, taking under a second to identify most messages.

If a message is not matched by any rule, it's checked to see if it's spam and if it passes that test, it's put into an `unknown` folder.

2.3 Spam, Spam, Spam, Spam

Hard rules are great for figuring out if an email came from a particular mailing list or a friend with just a few email addresses.

Hard rules don't always work for detecting spam.

One rule that does work (frequently) is to check whether or not the `To:` field of the header matches the expected user. This doesn't work for many mailing lists where the `To:` field is modified. (The Google Summer of Code `To:` field is sometimes `To: Google Summer of Code Mentors List .` Mail from mailing lists can usually be matched by checking the `Sender:` and `Reply-To:` fields. Once these have been weeded out, the test for `To:` works.

2.4 Open the pod baysean door, Hal

Given the dynamic and changing spam contents, you need a dynamic spam detection technique, not hard-coded algorithms or even sets of keywords.

The baysean statistical tests work by checking word counts in a suspect piece of mail against good mail and known spam mail. Put simply, the test mail gets a point for every word that only appears in good mail, and loses a point for every word that only appears in spam mail.

In the implementation, each word is assigned a positive value between 0 and 1 based on how often it appears in spam or good email, then the average value for the words in an email is calculated. If the value exceeds 0.7, it's very likely to be spam.

A sample of words from my dictionary looks like this:

```
purported 0.1
brownie 0.1
bytes 0.1
TANSTAAFL 0.1
listening 0.200642136694
routine 0.200838638462
automated 0.601255325782
warranties 0.601255325782
Funds 0.846210738272
funds 0.855801144836
FUNDS .9
Sexually .9
phonesex .9
```

As you can see, words that aren't commonly used by spammers get low scores. The words that are in fairly common use get medium scores, and words that appear in requests for assistance in moving large funds, or email suggesting illicit encounters are most likely to be spam.

This proves that I'm a geek, not a playboy.

Before checking an email, it needs to be cleaned a bit. Removing punctuation marks allows the code to use simple Tcl list commands and not treat a word followed by period as a different word than one not followed by a period.

This cleanup can be done with regular expressions, or the Tcl `string replace` command. Each of these techniques worked, but both were unreasonably slow.

To improve the speed, I wrote a small "C" extension to convert punctuation to spaces. This extension was compiled with `critcl`. It was a few minutes worth of work, and sped up the code greatly.

The addition of the `string map` command in more recent versions of Tcl removed the need for compiled code to clean a message before processing it.

The word definition file contains about 80,000 words. It takes a significant amount of time to read this much data. This becomes an issue when many emails arrive in a clump (for instance, when all the spam-bots owners turn on their PC's in the morning).

The solution to this issue was to make the baysean comparison application a server application which the mail sorting program calls.

The simplicity of using Tcl sockets made this a obvious solution.

The guts of the server simply opens a socket in server mode, waits for a connection and reads lines into a buffer. When an end-of-message marker is received, it processes the message and sends the score back to the client socket.

```
proc readLine {channel} {
   global Server

   set len [gets $channel line]
   # if we read 0 chars, check for EOF.  Close the
   #   channel if we’ve hit the end of the road.

     if {($len <= 0) & [eof $channel]} {
         close $channel
         return
     }
     # Watch for an unlikely end-of-message string
     # If it appears, process the message and clear the buffer
     # else append this line to the message being processed
     if {[string match $line "EndOfMailEndOfMail---liaMfOdnEliamFodnE"]} {
          set return [processMessage $Server(input.$channel)]
          puts $channel $return
          flush $channel
          set Server(input.$channel) ""
          return
        } else {
          append Server(input.$channel) " " $line
}
}

proc serverOpen {channel addr port} {
   global Server

   # Set up fileevent to be called when input is available

   fileevent $channel readable "readLine $channel"
   fconfigure $channel -buffering line
}


#
# Open the server for business
#
set server [socket -server serverOpen $Server(port)]
```

This reduced the lag time for sorting mail significantly.

2.5 Double, Double, Toil and Trouble

Occaissionally, my system will end up on a spam-bot’s list. The first time this happened, I received over a thousand pieces of identical email each hour.

This event made it obvious that having an easily modified sorting system is a good. It was the work of a few minutes to add a new test to see if a the body of incoming email matched the md5 of previously received email and discard duplicates. The `tcllib` md5 support just worked.

Making this the first test in the sorting application reduced the system load enough that other work could be done while the system was discarding email.

2.5.1 Subtracting with logs

One trick that reduces spam-bot email is greylisting. This technique refuses email from a new sender with a "temporary mail failure" message. The greylist milter waits a given length of time (usually about 1/2 hour) before the email from a new site will be accepted.

The justification for this technique is that legitimate email processors will accept a temporary failure for an email server and will retry. The spam-bots will either give up after one attempt or the user may turn the machine off before the retry interval elapses.

When I first installed the greylist milter on my mail server it reduced the spam count to about 1/10'th the amount of spam I had been receiving.

The commonly used greylist milter provides an easy to read ASCII log that includes the name of systems that have attempted to deliver email, the time of the attempt and the recipient address (which may be `bogusName@mysite.com`).

It's simple with Tcl to read that file and look for sites that have tried multiple times to send email to many different addresses (legit and otherwise). These sites (particularly ones trying to send to invalid addresses) are probably spam bots.

It's nice to not receive spam email, but it's even nicer to not even have to process it. Most Linux and Unix systems have firewall support that allows the system to reject any connection from a host.

The Tcl `exec` command provides a simple way to add a firewall rule that will drop any future connection from a given IP address. This reduces the load on the email server (it doesn't need to process data) and potentially crashes the spam-bot.

After a couple hours the IPTables rules are removed. If a site is actually legit, but being misused (for instance, an ISP that has had a number of customer's machines become infested), email will be received from that site again (until the site goes rogue again).

3 Mr. MUA Check and see - is there some email, some email for me?

These techniques work for reducing the amount of email that gets into the system, but eventually a user would like to read the email that's been sorted, folded, spindled and mutilated.

The mail read/sort application puts the email into individual files using the in Unix Mail format based on the sort criteria. All of the files go into the same folder.

Since the only files in this folder are mailboxes, a data-driven GUI can be built to provide a simple MUA to access the mail. The GUI learns what to put on the buttons using the glob command, and thus does not need to be modified when a new rule (and new mailbox) is added to the read/sort application.

The main portion of a button based MUA is built from code like this.

```
proc readMail {name}  {
  global Internal

  exec rxvt -e mutt -f $Internal(holderDir)/$name

  fillGUI
}

proc fillGUI {} {
  global Internal
  eval destroy [winfo children .files]
  foreach f [lsort [glob -types f -nocomplain $Internal(holderDir)/*]] {
    set fileName [file tail $f]

    set date [clock format [file mtime $f] -format "%b %d %H:%M"]
    set displ [format "%-12s %10s" $fileName $date]

    set w [button .files.f_$fileName -text $displ -pady 2 \
      -command "readMail $fileName"]
    pack $w -side top -anchor w -expand y -fill x
  }
}
pack [frame .files]
```

3.1 A MIME is a terrible thing to waste

The mutt mail reader, like pine, elm and other modern text-based mail readers works for many things and supports helper applications to read various MIME formats.

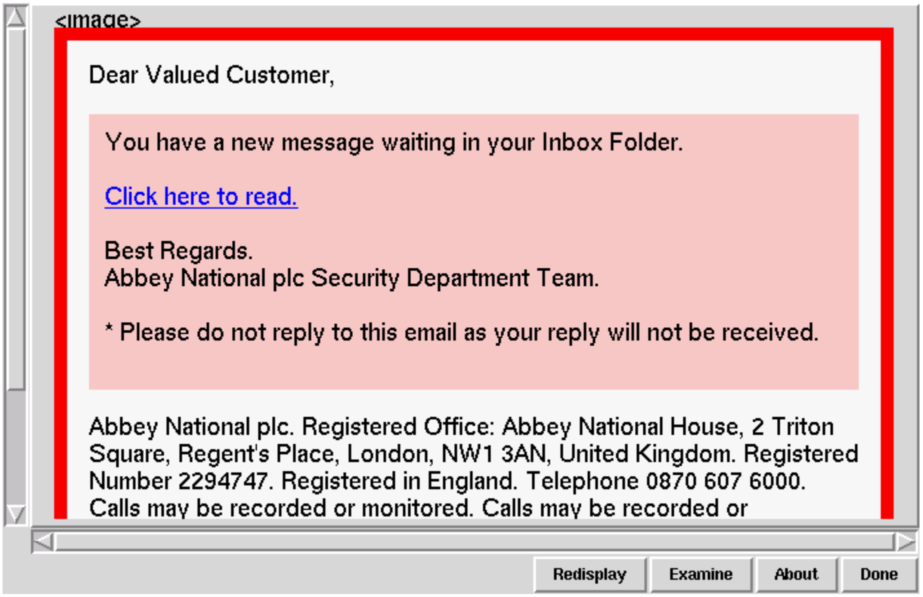
A glitch is reading HTML mail. The default choice for many folks is to invoke firefox, konqueror, opera or some other web browser to read HTML mail.

The problem with this technique is that the spammers who send mail with HTML attachments get tricky and just checking to see if this mail is *really* from Paypal can be enough to mark your email address as a valid addresss and ensure you years of overflowing mailboxes.

This problem led me to write my own HTML viewer. I did this initially using Steve Uhler's HTML widget, and later reworked it to use D. Richard Hipp's htmlwidget.

The advantage of this widget from a security/paranois aspect is that it's absolutely stupid. It does not have any HTTP support. It can't load an image, follow a link, run Javascript or do anything that might alert the outside world that you've read the email.

What it will do is provide a highlighted version of the HTML code to make it easy to find the HREF tags and see if the `Click Here` actually goes to your bank, or to some site in Nigeria.



<image>

Dear Valued Customer,

You have a new message waiting in your Inbox Folder.

Click here to read.

Best Regards.
Abbey National plc Security Department Team.

* Please do not reply to this email as your reply will not be received.

Abbey National plc. Registered Office: Abbey National House, 2 Triton Square, Regent's Place, London, NW1 3AN, United Kingdom. Registered Number 2294747. Registered in England. Telephone 0870 607 6000. Calls may be recorded or monitored. Calls may be recorded or

Figure 2: A phish message as viewed in an HTML reader

Figure 2 shows the `htmlview` application displaying a phish attack for a casual read. **Figure 3** shows the same message in `Examine` mode, highlighting the HREF and showing that the actual URL is `abbeynational.co.uk.0e3f0bc994.com`, not `abbeynational.co.uk`.

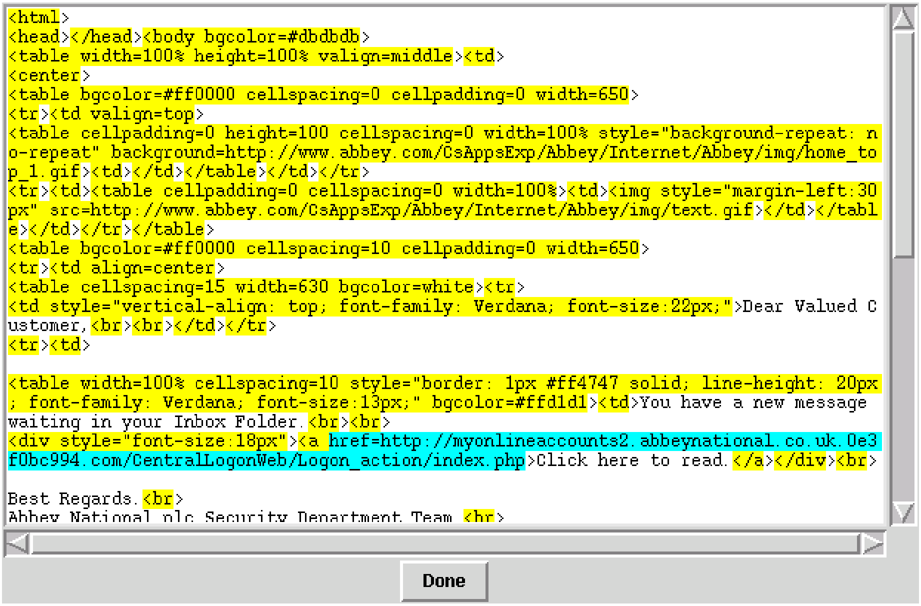
This application is available at `http://www.cwflynt.com/htmlview/`.

4 Getting to the root of the problem

All of these tools were fairly easy to write and made handling large amounts of email and larger amounts of spam possible.

To truly control the levels of spam, you need to catch it before it gets into your Mail User Agent. It's best if you can even drop something before it's fully sent to the mail server.

`Sendmail` and `postfix` support user written Mail Filters (milters). The `libmilter` library has hooks so that external filtering applications can connect to the Mail Transfer Agent (MTA), receive information about the email as



```
<html>
<head></head><body bgcolor=#dbdbdb>
<table width=100% height=100% valign=middle><td>
<center>
<table bgcolor=#ff0000 cellspacing=0 cellpadding=0 width=650>
<tr><td valign=top>
<table cellpadding=0 height=100 cellspacing=0 width=100% style="background-repeat: n
o-repeat" background=http://www.abbey.com/CsAppsExp/Abbey/Internet/Abbey/img/home_to
p_1.gif><td></td></table></td></tr>
<tr><td><table cellpadding=0 cellspacing=0 width=100%><td><img style="margin-left:30
px" src=http://www.abbey.com/CsAppsExp/Abbey/Internet/Abbey/img/text.gif></td></tabl
e></td></tr></table>
<table bgcolor=#ff0000 cellspacing=10 cellpadding=0 width=650>
<tr><td align=center>
<table cellspacing=15 width=630 bgcolor=white><tr>
<td style="vertical-align: top; font-family: Verdana; font-size:22px;">Dear Valued C
ustomer,<br><br></td></tr>
<tr><td>

<table width=100% cellspacing=10 style="border: 1px #ff4747 solid; line-height: 20px
; font-family: Verdana; font-size:13px;" bgcolor=#ffd1d1><td>You have a new message
waiting in your Inbox Folder.<br><br>
<div style="font-size:18px"><a href=http://myonlineaccounts2.abbeynational.co.uk.0e3
f0bc994.com/CentralLogonWeb/Logon_action/index.php>Click here to read.</a></div><br>

Best Regards.<br>
Abbey National plc Security Department Team <br>
```

Figure 3: The same image with the actual URL highlighted

it arrives and send Go/No-Go messages back to the MTA as the connection evolves.

Architecturally, a `sendmail` milter is familiar to a Tcl programmer: a milter registers a callback to be invoked when an event occurs. In this case events include receiving a connection from a remote mail server, receiving envelope information, reading a field in an email header, reading the body of an email and closing a session.

Because a `sendmail` application may be receiving many messages at the same time it uses threads to control the conversations.

At this point, it looks like Tcl and Milters are made for each other.

4.1 It ain't necessarily so

Several years ago, Michael Kirkham <`mikek@muonics.com`> wrote a Tcl Milter extension.

Unfortunately, the `libmilter` has been a moving target, and his need for this extension evaporated, so it was not maintained. His extension is a good start for someone doing more work with `libmilter`, and is available at `http://www.muonics.com/FreeStuff/TclMilter/`.

I downloaded this extension, and discovered that there were some simple issues with thread locking that could be fixed by changing only a few lines of code.

Then the fun started.

4.1.1 Hanging by a Thread

One feature of the libmilter architecture is that when an application calls smfi_main to enter the libmilter loop the smfi_main function will not return until the application exits. After this call, all activity is done via the callbacks to new threads.

Each time a new email message arrives, a temporary thread is created and the filter callback is evaluated in that thread. The overhead in creating a new Tcl interpreter is fairly low, but initializing an interpreter - loading extensions and command files, etc can be slow.

It makes more sense for a Tcl oriented mail filter to maintain a single persistent fully initialized interpreter and only use the per-message threads to redirect control into the primary interpreter.

Each message that the libmilter is processing gets a unique identification value that's passed to the "C" layer, and can be passed to the Tcl layer.

For a Tcl programmer, this makes the multiple threads rather redundant. We're used to dealing with this sort of construct with an upvar command:

```
proc doStuff {identifier value1 value2} {
  upvar #0 State_$identifier State
  global UniversalState
  # Do what needs to be done
}
```

The thread::send command will let one thread send a script to another thread to be evaluated in the target thread. The value returned by evaluating that script in the target thread will be return value of the thread::send command.

This feature could allow a procedure running in a per-message thread to send information to be retained to the primary, persistent thread. It also allows a per-message thread pass control to a filtering procedure in the primary thread, and only return the final result to the libmilter code.

From a Tcl script writer's perspective, this makes more sense than writing a filter in a thread that has to initialize an interpreter, load extensions, load command files, etc for each email that arrives.

The architecture I ended up with looks like **Figure 4**.

4.1.2 If I had a hammer

This pattern works well. It has been easy to write simple milters to do white, gray and black listing.

One common spam pattern is mail that comes from outside your domain with an Envelope From field from inside the domain. This could be valid mail, if you are the size of IBM with mail servers spread around the planet. Since I

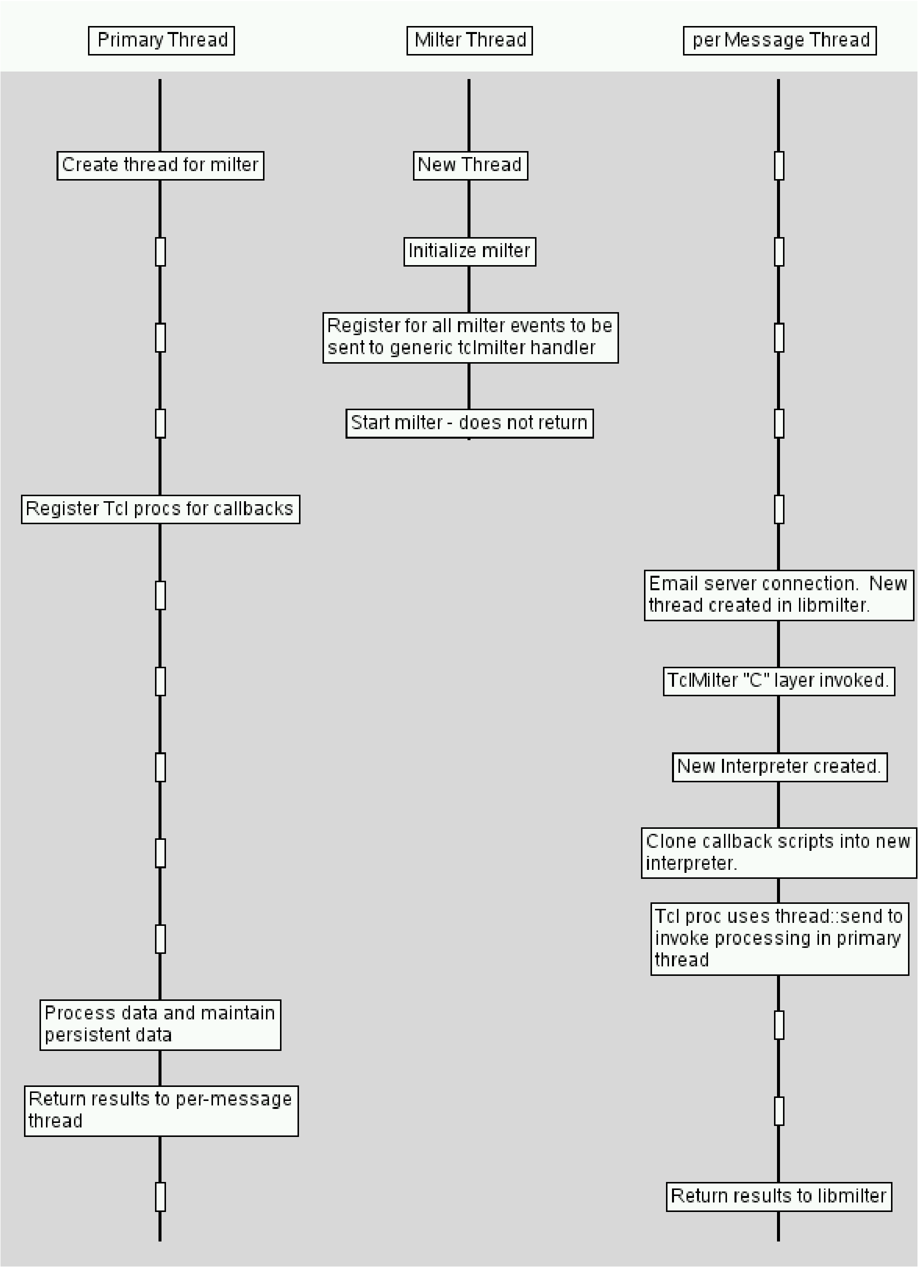

Primary Thread
Milter Thread
per Message Thread
Create thread for milter
New Thread
Initialize milter
Register for all milter events to be sent to generic tclmilter handler
Start milter - does not return
Register Tcl procs for callbacks
Email server connection. New thread created in libmilter.
TclMilter "C" layer invoked.
New Interpreter created.
Clone callback scripts into new interpreter.
Tcl proc uses thread::send to invoke processing in primary thread
Process data and maintain persistent data
Return results to per-message thread
Return results to libmilter

Figure 4: Multi-Thread Flow for TclMilter

only have one mail server, outside mail *never* comes from any of the domains I host.

The code to reject a forged `From:` address is fairly simple:

```
proc rejectForgedEnvFrom {context addressList } {
  upvar #0 mailState$context mailState
  global localNames
  global localIP

  # Return fail if we got here without the source being
  # identified.
  if {![info exists mailState(sourceIP)]} {
     return "SMFIS_TEMPFAIL"
  }

  # Assume it's good mail
  set rtn SMFIS_CONTINUE

  # Look at the addresses this email claims to be from.
  # This list *SHOULD* only contain one element.
  foreach address $mailState(envfrom) {
    # Split out the user and site portions of the From field
    lassign [getNameSite $address] nm site

    # If it didn't come from inside my network, see if it
    # claims to come inside my network.
    if {[lsearch $localIP $mailState(sourceIP)] < 0} {
      foreach id $localNames {
        if {[string first $id $site] >= 0} {
  # It claims to be from my site, but it isn't
  # Fail it and log it.
          set rtn SMFIS_TEMPFAIL
          milter::log "$context: Set TEMPFAIL because \
      from=$mailState(sourceIP) and ..$address.. \
      is not valid sender"
        }
      }
    }
  }

  # If it fails,
  if {$rtn eq "SMFIS_TEMPFAIL"} {
     setreply 451 4.9.8 "Invalid Sender"
  }
  return $rtn
}
```

Another annoying spammer trick is sending email to every possible user on the system - not valid users, but lists like alpha@noucorp.com, baker@noucorp.com, gamma@noucorp.com, etc.

These are never delivered, but the bounce messages clog the mail admin mailbox (making it impossible to actually use the mailbox) and chew up bandwidth.

A procedure similar to the rejectForgedEnvFrom proc removed all of that garbage from my system.

Greylisting is not so simple a task. However, the procedure uses under 150 lines of code.

4.1.3 Making Memories

The architecture seems simple and obvious and in small tests the filters worked just fine.

When I put the code into production, it wasn't so much fun.

Within an hour, the milter was using up almost 500 megabytes of RAM, and it was still growing.

Libmilter doesn't simply use threads to control email. It creates N threads at startup, and reuses them as necessary, rather than add the overhead of creating new threads. To further complicate matters, since sendmail may need to handle N+M threads at a time (to service N+M connections), it has facilities to handle multiple conversations per thread. Each connection is given a thread and a *context* within that thread to identify the message being processed.

The *context* construct that libmilter uses provides a clean just-like-new thread for each conversation, whether the thread has been used before (or is in use by another message currently) or not.

For something as simple as a pure "C" language milter, this isn't a problem.

For an interpreter like Tcl, it's a large issue. Tcl expects a thread to be created, used and to die, or to hang around and have some persistent data available.

The pthread library includes two functions to set and retrieve persistent data.

- pthread_setspecific

  Associates a key with a value within a given thread. Each thread can register a key value and retrieve that key value.

- pthread_getspecific

  Retrieves the value associated with a key for the calling thread.

The Tcl thread model requires a separate interpreter for each thread, and allows multiple interpreters per thread.

The pthread_setspecific and pthread_getspecific are used internally by the Tcl thread extension (version 2.6.5) to see if an interpreter has initialized the Thread Specific Data structure yet.

As part of libmilter providing a clean context for each message, it destroys data associated with the pthread_* calls, causing the Tcl interpreter to believe that it needs to initialize the ThreadSpecificData structure multiple times for a thread. This involves allocating new memory.

When a conversation is complete, the associated interpreter is destroyed, and the thread extension cleans up one ThreadSpecificData structure, though it may have created several.

This seems to be the mechanism behind the memory leaking.

4.1.4 The fat lady ain't singin' yet

Making libmilter and Tcl work together is a work in progress. Like Thomas Edison, I've started by trying dozens of ideas that didn't work. Unlike Thomas Edison, I haven't found the one idea that does work.

The most success to date has been to add a command to the thread extension to explicitly clean the ThreadSpecificData structure, and invoke that from the Tcl code that is processing an email conversation.

This has reduced the memory leakage from hundreds of megabytes / hour to merely thousands of bytes per hour. By restarting the TclMilter every 24 hours, it doesn't swamp the email system.

The new code in the thread extension looks like this:

```
static int
ThreadInit(interp)
    Tcl_Interp *interp; /* The current Tcl interpreter */
{
...
    TCL_CMD(interp, THNS"deinit",    ThreadDeInitObjCmd);
...
}

static void
DeInit(interp)
    Tcl_Interp *interp;         /* Current interpreter. */
{
    ThreadSpecificData *tsdPtr = TCL_TSD_INIT(&dataKey);
    ListRemove(tsdPtr);

}

static int
ThreadDeInitObjCmd(dummy, interp, objc, objv)
    ClientData  dummy;           /* Not used. */
    Tcl_Interp *interp;         /* Current interpreter. */
    int         objc;           /* Number of arguments. */
    Tcl_Obj    *CONST objv[];   /* Argument objects. */
```

```
{
    DeInit(interp);
    return TCL_OK;
}
```

5 The Future isn't quite Now

I've used several mail filters in the past. Despite the problems with memory leakage, I'm having more success at controlling email with my homegrown Tcl filters than with mailScanner, greyList and others.

It's fast and easy to add a short Tcl script to perform some new test when I need one. The email environment changes fast enough that yesterday's solution won't protect you from today's problem.

However, writing email filters is not my paying job. Work on the `libmilter` project is a spare time project, and once the code reached a level of useable (if not releasable) I had to move on to other projects.

It is my intention to return to this project. The ease of developing custom milters in Tcl is too useful a technique to ignore.