DAVID N. BLANK-EDELMAN

practical Perl tools: why I live at the P.O.

David N. Blank-Edelman is the Director of Technology at the Northeastern University College of Computer and Information Science and the author of the O'Reilly book *Perl for System Administration*. He has spent the last 22+ years as a system/network administrator in large multi-platform environments, including Brandeis University, Cambridge Technology Group, and the MIT Media Laboratory. He was the program chair of the LISA '05 conference and one of the LISA '06 Invited Talks co-chairs.

dnb@ccs.neu.dnb@ccs.neu.edu

WITH APOLOGIES TO THE GREAT BUT deceased Southern writer and a nod to her fan Steve Dorner, I'd like to take some time this issue to explore how Perl can help you with the business of manipulating data that resides on mail servers. Sending mail from Perl is a fairly well-known process (heck, it is even in the Perl FAQ; see perldoc perlfaq9 for more details), but the process of pulling data down from a server or moving it around on that server could use a little more explanation.

(Quick aside: I intentionally used the formulation "data on mail servers" instead of mailboxes or mail messages in that last paragraph. The AUP-breaking hacks that let you treat outsourced mail systems like Gmail as remote data stores have forever changed my view of just what is or can be stored on those servers. I'll return to the usual conventions now.)

Before we dive into the *how* of this process, I think it is reasonable for you to demand a good answer to the *why*. There are a whole host of reasons why you might want to automate mail operations like this via Perl. Some of these reasons are immediately apparent if you run the mail server in question. For example, to truly test that your mail system is working it is important to be able to check that your users can actually read their mail. I've publicly advocated that people write round-trip tests for mail systems that involve sending automatic mail to a test account that is then retrieved in a similarly automated fashion. This is far better than just a simple banner scrape to show your MTA is still listening on a socket.

If you don't run any mail servers you are still likely to encounter situations where automated mail-manipulation knowledge could be useful. For example, if your ISP does not perform spam filtering to your satisfaction, you could pull down all of the mail in your inbox, run it through whatever rigorous tests suit your fancy (no doubt involving some goat entrails) and then act on the messages before they can sully your mail reader. If that ISP can't do server-side filtering, your program could take over that job as well. The list goes on.

POP3 Goes the Weasel

Let's start someplace simple. The POP3 protocol, documented in RFC1939, offers a relatively un-

complicated way for a client to interact with a mail store. In most cases a POP3 client will:

1. Connect to a POP3 server and authenticate as a known user (known as a mailbox).
2. See if there is new mail.
3. Request the contents of the first new message and squirrel it away on the local machine.
4. Request that the server delete that message.
5. Repeat #3 and #4 for every remaining new message.
6. Signal that it is done with the connection and exit (with the server performing the actual deletion of data for the messages marked as deleted in step #4).

This set of six steps shows virtually all of the operations available in the protocol. The only two things of interest we did not mention are how "new messages" are handled and the TOP command. Let's quickly hit those two subjects in that order.

If a client always deletes all of the messages once it has downloaded them, it is trivial to determine when a message is new and requires downloading: Anything found in the mail store is by definition "new." But it isn't always advantageous to delete upon reading. The most common case where this isn't desirable is one where a user wants to have two separate POP3 clients looking at the same mailstore (e.g., your home machine and your work machine). One of them simply downloads the mail; the other will both download and delete it.

The client that doesn't delete the mail needs a way of remembering which messages it has seen before so it doesn't download them a second time. This is typically done using the POP3 UIDL command. UIDL asks the server to display a "unique-id listing" for a message or for each message on the server. This gives the client a piece of information that uniquely identifies each message on the server, which it can cache for future reference when deciding which messages to download. UIDL is officially "optional" in the RFC, but I have yet to see a modern POP3 server that didn't implement it.

I know that you are about to suffer from the DTs because you haven't seen any Perl code yet in this column, but hang on for a couple of more sentences because I want to mention one more POP3 feature I think will come in handy for you. POP3 also has an optional TOP command that allows the client to request the headers of a message followed by the first N lines of a message. This allows a client to get a peek at the contents of a message without having to download the whole thing.

Phew. With that verbiage out of the way, let's get to some code:

```
use Mail::POP3Client;

my $pop3 = new Mail::POP3Client(
   USER     => 'user',
   PASSWORD => 'secretsquirrel',
   HOST     => 'pop3.example.edu',
   USESSL   => 'true',
);

die 'Connection failed: ' . $pop3->Message() . "\n"
  if $pop3->Count() == -1;

print 'Number of messages in this mailbox: ' . $pop3->Count() . "\n\n";
print "The first message looks like this: \n" . $pop3->Retrieve(1) . "\n";

$pop3->Close();
```

This code uses the Mail::POP3Client module to connect to the POP3 server (over SSL, natch), retrieve the number of messages present, and then display the first message. There are a few POP3-oriented modules available on CPAN (the other popular one being Net::POP3), but I tend to like this one because the methods it provides mostly map directly to the commands in the protocol. If I had one quibble it would probably be that "mostly" part because I'd prefer it to offer methods I can infer from the RFC (even as an option) rather than making up its own. For example, it provides Retrieve(), also known as HeadAndBody(), whereas RFC calls that RETR.

Still, you can probably guess how you could extend this code to do more sophisticated things. Delete(message #) can be called to mark a message for deletion. Uidl() is available to you, returning an array whose contents contain the "unique-id listing" for each message or messages sought. TOP is even present in the form of a Head() method, and so on. Both the Head() and HeadAndBody() methods will return either a scalar or an array based on their calling context, so it is easy to get a mail header or message in the form desired by packages such as Mail::SpamAssassin.

IMAP and You Never Go Back

I don't want to dwell on POP3 any longer, because we have some more interesting fish to fry. The other protocol people use for interacting with their mail data is IMAP4. IMAP4 is a significantly more powerful (read: complex) protocol. Its basic model is different from that of POP3. With POP3 it is assumed that the POP3 client polls the POP3 server and downloads mail periodically. With IMAP4 a client connects to a server for the duration of the mail reading session. (Warning: There is a little hand-waving here, because of something known as disconnected mode, which we'll talk about in a sec.) With POP3 the client is expected to do all of the heavy lifting in the process. With IMAP4 the discussion between the server and the client is much richer and so the protocol has to be considerably smarter. Smarter how?

1. IMAP4 can deal with multiple mail folders and their contents (including other folders). RFC3501 says, "IMAP4rev1 includes operations for creating, deleting, and renaming mailboxes, checking for new messages, permanently removing messages, setting and clearing flags, RFC 2822 and RFC 2045 parsing, searching, and selective fetching of message attributes, texts, and portions thereof."
2. IMAP4 has support for disconnected clients. In disconnected mode a client can operate on a local cache of a mailbox even when not connected to its server. Later the client will play the changes back to the server to bring the local cache and the server's copy into sync. This is what allows you to sit on a plane without network access, deleting and filing mail, later to have those changes be propagated to the server when you get back on the Net.
3. IMAP4 has a much more granular understanding of an individual mail message. POP3 lets us grab a mail message's headers or headers plus N of the first lines of the message body. IMAP4 lets us say, "Give me the part of the message body that includes the message text but don't send me the data for the embedded attachments." It does this by grokking MIME natively.
4. Since this isn't a user-visible thing, you never hear about this last feature in POP3 vs. IMAP4 comparisons. If you watched the discussion between a POP3 client and server (actually, most client-server discussions), it would look like this: command from client, reply from server, command, reply, command, reply . . .

With IMAP4, the client can send a slew of commands at one time and have the server send responses to any of those commands anytime in the session. The two don't have to communicate in lockstep with each other. Each command is prefixed with a unique tag that the server will repeat back at the beginning of the response for that command. This lets both sides keep track of what has been asked and what is being answered.

Note that your code doesn't have to be written in a highly asynchronous manner using this capability (and in fact the examples in this column won't be), but it is good to know it exists if you do need to write high-performance IMAP4 code.

I just want to mention up front that given the complexity of the protocol, working with IMAP4 isn't always as intuitive as you'd like. Unfortunately, we don't have enough room in this column to look at all of the little squirrelly bits, so I'm going to constrain myself to very simple examples. If you start to write your own programs you *must* read the relevant RFCs (RFC2060 at a minimum, RFC2683 suggested) plus the documentation for whatever Perl module you choose to use.

For the sample code we're about to see, I'll be using my current preferred IMAP module, Mail::IMAPClient. This is the same module that forms the basis of the superb imapsync program (http://www.linux-france.org/prj/imapsync/dist/), a great tool for migrating data from one IMAP4 server to another. In addition to the vote of confidence because of imapsync, I like this module because it is mostly complete when it comes to features while still offering the ability to send raw IMAP4 commands should it become necessary. The other module that I would consider looking at is Mail::IMAPTalk by the primary developer behind Fastmail.fm. Even though it hasn't been updated in a few years, the author assures me that the current release still works well and is in active use there.

So let's dig into some IMAP4 code. As an example we'll use some code that connects to a user's mailbox, finds everything that was previously labeled as spam, and moves those messages to a SPAM folder. We'll start with connecting to the IMAP server:

```
use IO::Socket::SSL;
use Mail::IMAPClient;
my $s = IO::Socket::SSL->new(PeerAddr =>'imap.example.com',
                             PeerPort => '993',
                             Proto    => 'tcp');
die $@ unless defined $s;
my $m = Mail::IMAPClient->new(User     => 'user', Socket=>$s,
                              Password => 'topsecret');
```

This code is a little more verbose than I'd like, but I thought it was important to demonstrate how one uses an SSL connection to connect to the server. Mail::IMAPClient doesn't have SSL built in in the same way Mail::POP3Client does, so we had to construct an SSL-protected socket by hand and pass it to Mail::IMAPClient.

Once connected, the first thing one typically does is tell the server which folder to operate on. In this case we'll select the user's INBOX:

```
$m->select('INBOX');
```

Now that we have a folder selected, it's time to get to work. Let's find all of the messages in our INBOX that have the X-Spam-Flag header set to YES:

```
my @spammsgs = $m->search(qw(HEADER X-Spam-Flag YES));
die $@ if $@;
```

Now that I have a list of messages in @spammsgs, I can move each one over to the folder named SPAM:

```
foreach my $msg (@spammsgs){
  die $m->LastError unless defined $m->move('SPAM',$msg);
}
```

Once we've moved all messages we can close the mailbox and log out of the server:

```
$m->close();      # expunges currently selected folder
$m->logout;
```

There's a hidden detail in the first of these two lines of code that I feel compelled to mention. You might remember from the POP3 discussion that we talked about messages being "marked as deleted." The same tombstoning process takes place here as well. Deletes are always a two-step process in IMAP4 (flag as \Deleted and expunge messages marked with that flag). When we requested that a message be moved, the server copied the message to the new folder and marked the message in the source folder as being deleted. Ordinarily you would need to expunge() the source folder to actually remove the message, but RFC2060 says that a CLOSE operation on a folder explicitly expunges that folder, so we get away without having to do it ourselves.

I'd like to show only one more small IMAP4 example because there's still one last major topic left to cover in this column after IMAP4. I mentioned that IMAP can take apart messages (specifically, into their component MIME parts). Here's some code that demonstrates it. In the interests of saving space, I'll leave out the code from the last example that performed the initial SSL socket/connect to server and INBOX select:

```
my @digests = $m->search(qw(SUBJECT digest));

foreach my $msg (@digests) {

  my $struct = $m->get_bodystructure($msg);
  next unless defined $struct;

  # messages in a mailbox get assigned both a sequence number and
  # a unique identifier. By default Mail::IMAPClient works with UIDs
  print "Message with UID $msg (Content-type: ",$struct->bodytype,"/",
        $struct->bodysubtype,
        ") has this structure:\n\t",
        join("\n\t",$struct->parts) ,"\n\n";
}

$m->logout;
```

This code searches for all of the messages whose subject has the word "digest" in it. For each message it attempts to parse the structure of the message and print out a list of parts it finds. Here's a small snippet of output you might expect from the code:

```
Message with UID 2457 (Content-type: TEXT/PLAIN) has this structure:
    HEAD
    1

    Message with UID 29691 (Content-type: MULTIPART/MIXED) has this structure:
        1
        2
```

```
3
3.1
3.1.HEAD
3.1.1
3.1.2
3.2
3.2.HEAD
3.2.1
3.2.2
3.3
3.3.HEAD
3.3.1
3.3.2
4
```

If we needed to access just one of the parts of the message we can call body-part_string with the message number and part number. For example:

```
print $m->bodypart_string(29691,'4');
```

prints out the footer of the message with UID 29691:

```
_____________________________________________
Perl-Win32-Database mailing list
Perl-Win32-Database@listserv.ActiveState.com
To unsubscribe: http://listserv.ActiveState.com/mailman/mysubs
```

Mail::IMAPClient uses the Parse::RecDescent module to take apart MIME messages. I find that it works most of the time but has some issues with certain messages. If you are doing a lot of MIME groveling you may find that you'll either want to call a dedicated MIME parser or look at the module Mail::IMAPTalk mentioned earlier, which has the ability to parse messages into easy Perl structures. If we used Mail::IMAPTalk to fetch the body structure of that message and turned on its spiffy parse mode, here's an excerpt of what we would see stored for the footer part of the message:

```
3  HASH(0x85bf38)
   'Content-Description' => 'Digest Footer'
   'Content-Disposition' => HASH(0x85d9cc)
      empty hash
   'Content-ID' => undef
   'Content-Language' => undef
   'Content-MD5' => undef
   'Content-Transfer-Encoding' => '7BIT'
   'Content-Type' => HASH(0x85d0f8)
     'charset' => 'us-ascii'
   'IMAP-Partnum' => 4
   'Lines' => 4
   'MIME-Subtype' => 'plain'
   'MIME-TxtType' => 'text/plain'
   'MIME-Type' => 'text'
'Remainder' => ARRAY(0x856528)
  0  undef
'Size' => 193
```

Make All the Hairy and Scary Code Go Away

There remains one last thing to mention: If all of this gnarly POP3 and IMAP4 code has you worried, there are a few modules out there that attempt to abstract out the details necessary for dealing with mail on a POP3 or

IMAP4 server. For example, the Email::Folder family (part of the Perl Email Project, at emailproject.perl.org/wiki/Email::Folder) lets you write code like this (from the doc):

```
use Email::Folder;
use Email::FolderType::Net;

my $folder = Email::Folder->new('imaps://example.com'); # read INBOX

print $_->header('Subject') for $folder->messages;
```

The other package worth considering is the all-singing-all-dancing MailBox. Here's what the author says: "The MailBox package is a suite of classes for accessing and managing email folders in a folder-independent manner. This package is an alternative to the Mail::Folder and MIME::* packages. It abstracts the details of messages, message storage, and message threads, while providing better performance than older mail packages. It is meant to provide an object-oriented toolset for all kinds of e-mail applications, under which Mail User Agents (MUA) and mail filtering programs [*sic*]."

MailBox is a highly engineered package with tons of functionality (which may be a good or a bad thing in your eyes). It ships with enough module tests to choke a horse (which is likely to be a good thing from your management's perspective). MailBox actually uses Mail::IMAPClient under the hood to do its IMAP4 work, but you'll never know it because it abstracts all of the IMAP4 details away for you.

With that pointer, it is time to exit. Have fun manipulating your mail data from Perl. Take care, and I'll see you next time.