



alemba\_arts, Fotolia

Perl script tunnels mail traffic on demand

# Tunnel Vision

Instead of running a local mailserver, a Perl daemon listens to outgoing SMTP requests and drills a temporary SSH tunnel to a remote SMTP server on demand. *By Michael Schilli*

**M**y Internet service provider normally handles the job of shoveling data packets around fairly well. But if something fails, I often get a script-reading ignoramus on the hotline who totally ignores elementary, logical principles. Instead, they attempt to put the blame on the user instead of telling the trained system administrators who work with them that the problem is obviously on their side. Once, when I called to complain about a slow DNS server, somebody actually asked if my DSL modem was on the floor or in the book case.

## The Age of Spam

Just recently, I had a problem with their SMTP server and wanted to avoid the frustration of calling my provider. I don't send much in the line of email from my home desktop, but when I do, I expect it to reach its destination. For example, if there's a power failure, my UPS cuts in,

a fact that is noticed by Nagios, which in turn quickly sends me an email.

Of course, I could turn to my hosting provider instead, a private company who doesn't operate as a government-protected quasi-monopoly. Their SMTP server is very reliable, but in the age of spam, they won't accept mail from unknown IPs. Because the provider offers SSH access, I could drill a tunnel like

```
ssh -L 1025:localhost:25 Z
mschilli@host.provider.com
```

from my local port 1025 to the SMTP port (25) on the hosted computer. From the point of view of the computer in my hosting provider's farm, it would look like the request came from the leased shared host Web server.

## Dynamic Drilling

Budget hosting providers will probably not want scrooges like myself keeping

SSH tunnels open day and night without typing something into their leased websites; but, if I only drill the tunnel when I want to send out email and then tear it down afterward, they'll probably be okay with it. To implement this, the `minimail` daemon, written in Perl, listens for requests from local mail clients on the SMTP port (25). The clients are blissfully unaware of the complexity behind this, they'll be under the impression that they're talking to a local mailserver.

The daemon accepts the request, opens a tunnel on local port 1025 to port 25 of the hosting provider, then waits for the connection to come up. For the local mail client, this just looks like a fairly slow mail server. The daemon then shoves the request lines from the client (local port 25) to local port 1025. Packets are entering the tunnel and pushed through to port 25 on the provider's side (Figure 1). Return packets, arriving back through the tunnel are forwarded by the

daemon to the local client, which completes the impression that it is indeed talking to the local SMTP server.

If multiple requests to send mail occur in quick succession, it doesn't make sense to break down and build up the tunnel again; to handle this case, the daemon leaves the tunnel up for 10 seconds after the last client has bailed out. To keep this looking human in the host's logs, the script adds a random number between 0 and 25 seconds to the wait.

## To Root or Not to Root?

To allow the daemon in Listing 1 to `bind()` the SMTP port (25), it must run as the root user; to mitigate the security risk this implies, the daemon drops these privileges later on. A program launched with `sudo` has the `SUDO_USER` set to the environment variable of the account that ran the `sudo` command. The script drops its privileges and changes the effective user ID to this non-privileged account. The `sudo_me()` command in line 15 from the CPAN `Sysadm::Install` module checks if `root` ran the script and, if not, uses `sudo` to change things.

The CPAN `App::Daemon` module exports the `daemonize()` function which lets the script act as a daemon and process the `minimail start|stop` commands. It will put itself into the background after running through the start sequence – only the logfile reveals what

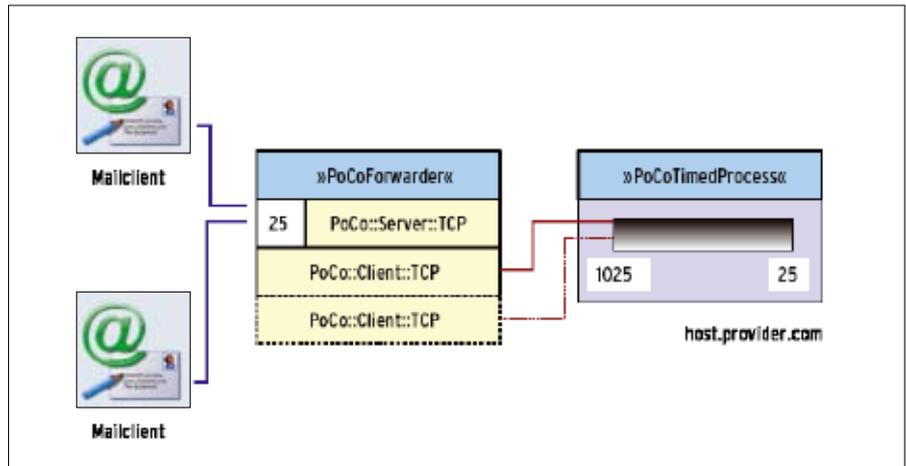


Figure 1: The mail client talking to port 25 on the forwarder, whose TCP client session talks to the tunnel.

the daemon is currently doing. The `Log4perl` logfile is set by the `-l` option or, programmatically, via the `App::Daemon::logfile` variable, as shown in line 18. If the daemon is launched in the foreground with the `-X` option, the log output is sent to `Stderr` instead.

The `BEGIN` block in lines 14-23 makes sure that the POE module in line 25 is not loaded until the process has been daemonized (line 22). This is important, so a helpful soul from the POE mailing list told me; otherwise, POE won't clean up the child processes it creates later on.

Because `App::Daemon` also offers a feature for dropping root privileges, line 16 of the module assigns a value of `root`

to the `$as_user` variable and thus leaves the security switch to the script, which handles it after binding the daemon to port 25 in the forwarder code, starting at line 48.

## POE to the Rescue

Writing your own network daemon normally costs plenty of blood, sweat, and tears, but, thankfully, CPAN offers a number of POE components you just need to glue together. For example, `minimail` creates the `PoCoForwarder` port forwarder from the `POE::Component::Client::TCP` and `POE::Component::Server::TCP` components. It binds with the local `$port_from` port and forwards anything

## LISTING 1: minimail

```
01 #!/usr/local/bin/perl -w
02 #####
03 # minimail - SMTP daemon
04 # auto-opening tunnels
05 # Mike Schilli, 2010
06 # (m@perlmeister.com)
07 #####
08 use strict;
09 use Sysadm::Install qw(:all);
10 use App::Daemon
11 qw(daemonize);
12 use Log::Log4perl qw(:easy);
13
14 BEGIN {
15 sudo_me();
16 $App::Daemon::as_user =
17 "root";
18 $App::Daemon::logfile =
19 "/var/log/minimail.log";
20 $App::Daemon::loglevel =
21 $INFO;
22 daemonize();
23 }
24
25 use POE;
26 use PoCoForwarder;
27 use PoCoTimedProcess;
28
29 my $port_from = 25;
30 my $port_to = 25;
31 my $tunnel_port = 1025;
32 my $real_smtp_host =
33 'host.provider.com';
34
35 my $process =
36 PoCoTimedProcess->new(
37 heartbeat => 10,
38 timeout => int(rand(25)) +
39 10,
40 command => [
41 "ssh", '-N', '-L',
42 "$tunnel_port:" .
43 "localhost:$port_to",
44 $real_smtp_host
45 ],
46 );
47
48 my $forwarder =
49 PoCoForwarder->new(
50 port_from => $port_from,
51 port_to => $tunnel_port,
52 port_bound => sub {
53 INFO "Dropping privileges";
54 $< = $> = getpwnam(
55 $ENV{SUDO_USER});
56 },
57 client_connect => sub {
58 $process->launch();
59 },
60 );
61
62 $process->spawn();
63 $poe_kernel->run();
```

that arrives there to the `$tunnel_port` – and vice versa. This is no trivial matter because multiple mail clients can use the local port at the same time and would need to be served in parallel.

The second component, that is, `PoCoTimedProcess`, uses the `launch()` method to start a process like the tunnel for a certain amount of time or extends its lifetime if it is already running. Every time the forwarder discovers a newly docked client, it calls the `launch()` method in the `client_connect()` callback (line 58). The method calls the `ssh` command in lines 41-44,

```
ssh -N -L 2
1025:localhost:25 host.provider.com
```

thus connects to the host at `host.provider.com` via the encrypted SSH protocol, logs in when it gets there, and, thanks to the `-N` option, doesn't start an interactive shell but just hangs around forwarding datastreams back and forth.

Port 1025 is the desktop-side end of the tunnel; however, `localhost` in the `ssh` command above refers to `host.provider.com`, because the SSH session is logged in there at this point. The `25` following the colon is the SMTP port on the

hosted machine. If the username name on the hosted machine is not the same as on the desktop, the call needs to add a valid account name like `mschilli@host.provider.com` to tell SSH which to use.

## Component Glue

What happens behind the scenes in the two POE components? Figure 1 shows the diagram with the server and client components and the port numbers they use. The port forwarder TCP server listening on port 25 winds up a TCP client session for each client to connect them to the tunnel independently.

### LISTING 2: PoCoForwarder.pm

```
001 #####
002 # POE Port Forwarder
003 # Mike Schilli, 2010
004 # (m@perlmeister.com)
005 #####
006 package PoCoForwarder;
007 use strict;
008 use Log::Log4perl qw(:easy);
009 use
010 POE::Component::Server::TCP;
011 use
012 POE::Component::Client::TCP;
013 use POE;
014
015 #####
016 sub new {
017 #####
018 my ($class, %options) = @_;
019
020 my $self = {%options};
021
022 my $server_session =
023 POE::Component::Server::TCP
024 ->new(
025 ClientArgs => [$self],
026 Port => $self->{port_from},
027 ClientConnected =>
028 \&client_connect,
029 ClientInput =>
030 \&client_request,
031 Started => sub {
032 $self->{port_bound}->(@_)
033 if defined
034 $self->{port_bound};
035 },
036 );
037
038 return bless $self, $class;
039 }
040
041 #####
042 sub client_connect {
043 #####
044 my (
045 $kernel, $heap,
046 $session, $self
047 )
048 = @_[
049 KERNEL, HEAP,
050 SESSION, ARG0
051 ];
052
053 $self->{client_connect}
054 ->(@_)
055 if defined
056 $self->{client_connect};
057
058 my $client_session =
059 POE::Component::Client::TCP
060 ->new(
061 RemoteAddress =>
062 "localhost",
063 RemotePort =>
064 $self->{port_to},
065 ServerInput => sub {
066 my $input = $_[ARG0];
067
068 # $heap is the
069 # tcpserver's (!) heap
070 $heap->{client}
071 ->put($_[ARG0]);
072 },
073 Connected => sub {
074 $_[HEAP]->{connected} = 1;
075 },
076 Disconnected => sub {
077 $kernel->post($session,
078 "shutdown");
079 },
080 ConnectError => sub {
081 $_[HEAP]->{connected} = 0;
082 $kernel->delay(
083 'reconnect', 1);
084 },
085 ServerError => sub {
086 ERROR $_[ARG0]
087 if $_[ARG1];
088 $kernel->post($session,
089 "shutdown");
090 },
091 );
092
093 $heap->{client_heap} =
094 $kernel->ID_id_to_session(
095 $client_session)
096 ->get_heap();
097 }
098
099 #####
100 sub client_request {
101 #####
102 my ($kernel, $heap,
103 $request) =
104 @_[ KERNEL, HEAP, ARG0 ];
105
106 return if
107 # tunnel not up
108 # yet, discard
109 !$heap->{client_heap}
110 ->{connected};
111
112 $heap->{client_heap}
113 ->{server}->put($request);
114 }
115
116 1;
```

The class expects the `port_from` port (the one on which the server is listening to client requests), the `port_to` port (the desktop end of the tunnel), and two callback routines as parameters. The component jumps to the subroutine reference stored in `port_bound` once the server has bound to port 25 and can thus drop its root privileges.

When dropping root privileges, it is important to do it in the right order for effective and real user IDs; otherwise, the daemon could reestablish its root privileges later [2]. With multiple parallel threads, `PoCoTimedProcess` internally would have to prevent a race condition launching the tunnel twice. In the one-process, one-thread environment that POE provides, a simple variable check without locking is fine – robust, easy to code, and easy to understand when you come back to the program years later!

The second forwarder callback, `client_connect`, is accessed whenever a mail client docks on port 25. The `PoCo-`

`TimedProcess` component's `launch()` method, which is executed in the callback, then sets up the tunnel if it doesn't exist. Internally, `PoCoForwarder` provides a `PoCo::Client::TCP` type POE component for each client connection, and each connects to the desktop tunnel port. In other words, although `PoCo::Server::TCP` can manage any number of clients, you need to deploy a separate `PoCo::Client::TCP` component for each.

## Closures: Confusingly Elegant

Line 32 in Listing 2 shows how the component implements the `port_bound` callback. The POE TCP server created in line 24 enters the `Started` state after launching successfully. `PoCoForwarder` retrieves the subroutine reference defined by `minimail` from the `$self` object hash and calls it. The callback code defined in `Minimail` handles everything else.

Note that `$self` is not in the scope of the handler assigned to the `Started`

state. Instead, it comes courtesy of the `PoCoForwarder` class's `new()` constructor; however, the subroutine mutates to a closure that includes the lexical `$self` variable and thus remains valid after leaving the scope of the constructor (but only within the callback).

On the other hand, the `Client-Args` parameter in line 25 makes sure the server component provides the `$self` object hash as an argument, `ARGO`, if it enters the `client_connect()` callback function. In line 54, the component runs the `client_connect` callback set by the main script earlier, which launches the tunnel process. Note the timing problem that occurs here because it is difficult to predict how long the tunnel will take to come up. This means that our newly fired up TCP client might try to bind to a port later when no one is listening in.

In this case, it isn't an issue. The TCP client enters the `ConnectError` state (line 80), which schedules a reconnect event for one second later in POE's todo list

### LISTING 3: PoCoTimedProcess.pm

```

001 #####
002 # POE Timed Process
003 # Launcher Component
004 # Mike Schilli, 2010
005 # (m@perlmeister.com)
006 #####
007 package PoCoTimedProcess;
008 use strict;
009 use warnings;
010 use POE;
011 use POE::Wheel::Run;
012 use Log::Log4perl qw(:easy);
013
014 #####
015 sub new {
016 #####
017 my ($class, %options) = @_;
018
019 my $self = %options;
020 bless $self, $class;
021 }
022
023 #####
024 sub launch {
025 #####
026 my ($self) = @_;
027
028 $poe_kernel->post(
029 $self->{session}, 'up');
030 }
031
032 #####
033 sub spawn {
034 #####
035 my ($self) = @_;
036
037 $self->{session} =
038 POE::Session->create(
039 inline_states => {
040 _start => sub {
041 my ($h, $kernel) =
042 @_[ HEAP, KERNEL ];
043
044 $h->{is_up} = 0;
045 $h->{command} =
046 $self->{command};
047 $h->{timeout} =
048 $self->{timeout};
049 $h->{heartbeat} =
050 $self->{heartbeat};
051 $kernel->yield(
052 'keep_alive');
053 $kernel->yield(
054 'heartbeat');
055 },
056 sig_child => sub {
057 delete $_[HEAP]->{wheel};
058 },
059 heartbeat => \&heartbeat,
060 up => \&up,
061 down => \&down,
062 keep_alive => sub {
063 $_[HEAP]->{countdown} =
064 $_[HEAP]->{timeout};
065 },
066 closing => sub {
067 $_[HEAP]->{is_up} = 0;
068 },
069 }
070 )->ID();
071 }
072
073 #####
074 sub heartbeat {
075 #####
076 my ($kernel, $heap) =
077 @_[ KERNEL, HEAP ];
078
079 $kernel->delay("heartbeat",
080 $heap->{heartbeat});
081
082 if ($heap->{is_up}) {
083 INFO
084 "Process is up for another ",
085 $heap->{countdown},
086 " seconds";
087

```



**Figure 2:** When a message needs to be sent, Minimail needs to open the tunnel for the first request ...



**Figure 3:** ... then the SMTP server at the other end of the tunnel will respond within about one or two seconds ...



**Figure 4:** ... and the client can then exchange SMTP commands as if connected directly. The server thinks it is talking to a local client.

with the `delay()` POE kernel function. This game can go on for a couple of rounds, but the tunnel will come up eventually. The TCP client then binds the port, which is now working, and can enter the `Connected` state as of line 73.

### The Tunnel is Ready

If Minimail sends a command, the TCP server branches to the `client_request` state and thus to the handler (lines 100-114), which checks that the tunnel is already up and ignores the client command if the connection is down. The SMTP protocol stipulates the server has to start the communication with a greeting. A well-behaved client will not start to talk until the server says hello, which

only happens if the tunnel is up. With other protocols (e.g., HTTP), it is different; in this case, the forwarder would have to buffer the client commands until the connection was up then forward them in lieu of the client in a bundle.

If the tunnel is ready, the heap variable `connected` is 1 in the `Connected` state handler. To forward the message to the tunnel, line 112 retrieves the saved TCP client heap and pulls out its `server` entry, whose `put` method is then used to forward the request to the tunnel entry the client docked onto earlier. Note that `client_request()` is a server session callback that knows nothing about the client, which is running in another session, or the client's heap. The `client_`

heap heap variable, set in line 93 in the server session, solves this problem.

When messages come back out of the tunnel, the TCP client switches to the `ServerInput` state in line 65, which then uses the `put()` method to return the text with the client reference stored on the heap, and hence to Minimail. If Minimail disconnects from the TCP server, the server enters the `Disconnected` state, and the handler sends a shutdown event to the running session (line 77), finally interrupting the client server connection.

### Processes with Countdown

Handles in the `PoCoTimedProcess.pm` component (Listing 3) set up and break

#### LISTING 3: PoCoTimedProcess.pm (part2)

```

088  $heap->{countdown} -=          113  }                               138  $_[KERNEL]
089  $heap->{heartbeat};           114  }                               139  ->yield('keep_alive');
090                               115  my ($prog, @args) =           140  $heap->{is_up} = 1;
091  if (                           116  @ { $heap->{command} };       141  }
092  $heap->{countdown} <= 0)       117  }                               142
093  {                               118  $heap->{wheel} =             143  #####
094  INFO                           119  POE::Wheel::Run->new(         144  sub down {
095  "Time's up. Shutting down";    120  Program => $prog,            145  #####
096  $kernel->yield("down");         121  ProgramArgs => [@args],      146  my ($heap, $kernel) =
097  return;                        122  CloseEvent => "closing",     147  @_[ HEAP, KERNEL ];
098  }                               123  ErrorEvent => "closing",    148
099  }                               124  StderrEvent => "ignore",    149  if (!$heap->{is_up}) {
100  }                               125  );                           150  INFO
101                               126  }                               151  "Process already down";
102  #####                          127  my $pid =                    152  return 1;
103  sub up {                        128  $heap->{wheel}->PID();        153  }
104  #####                          129  INFO "Started process $pid"; 154
105  my ($heap, $kernel) =          130                               155  INFO "Killing pid ",
106  @_[ HEAP, KERNEL ];           131  $kernel->sig_child($pid,      156  $heap->{wheel}->PID;
107                               132  "sig_child");              157  $heap->{wheel}->kill();
108  if ($heap->{is_up}) {           133  $kernel->sig(                 158  $heap->{is_up} = 0;
109  INFO "Is already up";         134  "INT" => "down");           159  $kernel->sig_handled();
110  $_[KERNEL]                     135  $kernel->sig(                 160  }
111  ->yield('keep_alive');         136  "TERM" => "down");          161
112  return 1;                      137                               162  ;
    
```

down the tunnel. When `minimail` uses `spawn` (line 62) to launch the process timer's POE session, its first course of action is running the `_start` handler defined in `PoCoTimedProcess.pm` (line 40). The handler in turn uses a closure to extract all the critical parameters, such as `heartbeat` (check frequency for a timeout), `timeout` (number of seconds until tunnel breakdown), and `command` (the SSH command for setting up the tunnel) from the `self` object hash and stores them on the session's own heap. It then sets two events for processing by the POE kernel at a later stage: `keep_alive` and `heartbeat`. The former resets the heap `countdown` variable to the maximum value in seconds to keep a tunnel open, which is defined in `timeout`. Additionally, POE calls the `heartbeat` event at regular intervals, thanks to the `delay` method in line 79, every time the number of seconds defined in the heap `heartbeat` variable has elapsed.

The tunnel is closed at first, but as soon as the `launch()` method triggers the `up` event and POE activates the matching `up` handler (line 103), a `POE::Wheel::Run` object (line 119) fires up the SSH tunnel process. The handlers for the Unix `INT` and `TERM` signals defined in lines 134 and 136 ensure that the `minimail` process will tear down an open tunnel if the main script is killed unexpectedly.

Once the tunnel has reached its maximum lifetime, line 96 triggers the `down` event and the matching handler (line 144) sends a kill signal to the `ssh` process. To let other handlers know that the tunnel no longer exists, `down()` sets the `is_up` variable to 0. This completes the processing of the triggering signal; the call to `sig_handled()` in line 159 prevents the POE kernel from acting on it as well, which would be undesirable because the kernel's default action on these signals is to terminate the daemon.

To prevent the killed process mutating into a zombie, joining a growing army of other zombies, and finally bringing the computer to its knees, line 131 defines a `sig_child` handler, which reaps the dying process and then enters the `sig_child` state of the POE session, defined in line 56. This helps POE give the dying tunnel its last rites (internally, via `waitpid()`) and prevents it from going to zombie hell. The handler finally deletes the last remaining reference to `POE::`

```
mybox> tail -F /var/log/minimail.log
2010/04/18 23:40:56 Process ID is 17415
2010/04/18 23:40:56 Written to /tmp/mailer.pid
2010/04/18 23:40:56 Dropping privileges
2010/04/18 23:41:05 Starting program
2010/04/18 23:41:06 Process is up for another 23 seconds
2010/04/18 23:41:06 Is already up
2010/04/18 23:41:56 Process is up for another 23 seconds
2010/04/18 23:42:06 Process is up for another 13 seconds
2010/04/18 23:42:16 Process is up for another 3 seconds
2010/04/18 23:42:16 Time's up. Shutting down
2010/04/18 23:42:16 Shutting down process
mybox>
```

Figure 5: The daemon logs critical events.

Wheel. POE figures out it has nothing left to do and neatly folds up the kernel.

## Keys Instead of Passwords

Because a daemon can't use an interactive password dialog to identify itself, the `ssh` tunnel command requires the user to create a keypair:

```
ssh-keygen -t rsa
```

The keys will typically be stored in the `id_rsa` (Private Key) and `id_rsa.pub` (Public Key) files in the `.ssh` directory below the user's home directory.

To make sure the hosting service provider lets the daemon connect to it, the user has to push the public key created with the `no_passphrase` option to the server. This involves appending the local content of the `id_rsa.pub` file to the `.ssh/authorized_keys` file on the hosting server. If you then enter the `ssh` tunnel command in Minimail manually (without the `-N` option), you should be logged in to the hosting server without being asked for your password.

## Trial Run with Telnet

The Telnet command in Figure 2 with `localhost` and port 25 discovers whether the mail server that was launched by `sudo minimail start` really works. If the daemon tunnel is down, Minimail will delay the response by one or two seconds until the mail server provider-side responds and then patch through to the SMTP server on the other end (Figure 3).

If you speak some SMTP, you can try out a couple of tricks (for test purposes only, of course – Figure 4). The daemon will busily take note of this in the `/var/log/minimail.log` logfile (Figure 5). It will not store the mail headers or text for data protection reasons.

While running tests with the `telnet` command, you can get out of a hung

session caused by a server not releasing the client by pressing the keyboard shortcut `Ctrl + ]`, which takes Telnet down into a shell that you can terminate by pressing `q`.

## Waiting for a Power Failure

To launch the Minimail server automatically every time you boot your machine, you need to add

```
SUDO_USER=mschilli /path/to/minimail
```

on Ubuntu to the `/etc/init.d/minimail` file, which you might need to create, then make the file executable with `chmod +x` and finally call

```
sudo update-rc.d minimail defaults 80
```

to add the script to the boot process. When the power returns, the new mail server boots automatically and makes sure it is ready to take messages once Nagios reports that power has been restored and disaster averted. ■■■

## INFO

- [1] Listings for this article: <http://www.linuxpromagazine.com/Resources/Article-Code>
- [2] Dropping privileges, but properly: [http://perlmonks.com/?node\\_id=833950](http://perlmonks.com/?node_id=833950)

## AUTHOR

Michael Schilli works as a software engineer with Yahoo! in Sunnyvale, California. He is the author of *Goto Perl 5* (German) and *Perl Power* (English), both published by Addison-Wesley, and he can be contacted at [mschilli@perlmeister.com](mailto:mschilli@perlmeister.com). Michael's homepage can be found at <http://perlmeister.com>.

