Reprinted from the

Proceedings of the Ottawa Linux Symposium

June 26th–29th, 2002
Ottawa, Ontario
Canada

Conference Organizers

Andrew J. Hutton, *Steamballoon, Inc.*
Stephanie Donovan, *Linux Symposium*
C. Craig Ross, *Linux Symposium*

Proceedings Formatting Team

John W. Lockhart, *Wild Open Source, Inc.*

Metanet: Message-Passing Network Daemons

Erik Walthinsen
omega@temple-baptist.com, http://www.omegacs.net/~omega/

Abstract

MetaNet is a message-passing architecture designed for the construction of network services daemons. Services are implemented as a series of mailboxes with attached functions to provide the functionality. The mailbox namespace is global across all services, allowing new daemons to hook into existing daemons to modify their behavior. A simple example is a stock DHCP daemon hooked by an external application that does an LDAP lookup for a machine's IP before falling back to the normal DHCP allocation scheme. This paper covers the architecture of the MetaNet system, and uses the example of a Captive Portal as used for public wireless network control to show how multiple services can be quite easily tied together to provide more complex services. Other possible projects using MetaNet will also be explored.

1 Introduction

Network services on Unix machines are provided today by separate daemons designed for each protocol and service. Each has its own internal structure, configuration files, behaviors, history, and so on. This works well in most environments because each service is designed to be completely standalone, as this is one of the driving principles behind Unix: "Do one thing, and do it well." For the most part, each of these daemons does indeed do its job reasonably well.

Unfortunately, there are situations where this myriad of separate daemons can make it difficult to accomplish the task at hand, or more likely they are simply too big to fit on the target device. The platform example used in this paper is that of an intelligent wireless access point (AP) running Linux. The M1 from Musenki [Musenki] is just such an access point, consisting of a Motorola MPC8241 embedded PPC processor, 32MB of RAM, 8MB of flash, a 10/100 NIC, and a MiniPCI slot for a wireless card. It runs Linux natively and must be able to provide all the normal network services, as well as be useful to the PersonalTelco Project [PTP] in its city-wide free networking projects. This drives some of the more interesting requirements, pushing well beyond the boundaries of current software.

MetaNet is an attempt at creating an infrastructure that enables these services to be not only small, but capable of extremely complex interactions. MetaNet itself is only an architectural design, not a specific implementation. The reference implementation is currently (as of this writing) written in Perl for convenience, but could be and will be implemented in other

languages such as C and Python. More advanced features ensure that these languages can be mixed freely in larger systems, allowing high-speed services to be written in C and other services or glue code to be written in a scripted language.

2 Unusual Requirements

The PersonalTelco Project (PTP) has several (4 as of this writing, more on the way) open, public access points covering several very public locations in Portland, OR. The most visible of these is the node at Pioneer Courthouse Square, aka "Portland's Living Room", in the center of downtown. A wireless AP covers the Square and is connected to the Internet by redundant T-1's that are otherwise heavily underutilized by the company donating the space and bandwidth. Because the signal is unencrypted and usable by anyone with a laptop and a wireless card, some steps are necessary in order to keep usage under control, and limit or avoid liability.

The solution used is called a "Captive Portal." The idea behind such a portal is that new clients on the wireless network are by default completely firewalled off from the Internet. In order for the user to gain access to the network they must log in after agreeing to a usage policy, etc. Custom client login software is unworkable because it would have to be distributed to clients and installed, which most savvy users wouldn't even consider. Instead, the portal uses software the user already has: a browser. Linux Netfilter [Netfilter] transparent redirects are used to shuttle all HTTP connections to a web server contained in the portal software itself. When the user successfully logs in, the firewall is modified to allow that client access to the Internet.

Once the client is associated and logged into the portal, there are several other aspects that have to be dealt with. The first is the fact that a spammer could trivially send mass quantities of email through such a wide-open connection, and could do so untraceably. Another is making sure that no single client abuses the upstream connection and effectively locks other clients off the network. Statistics logging and "Extrusion Detection" are also critical components that have to be built into such a system.

3 An Example

Rather than explaining how one might implement a complete Captive Portal with fully discreet daemons and why it would be nearly impossible, a simpler example will suffice to explain the basic MetaNet concepts.

On many large networks, an LDAP database is used to hold information about each machine and user. If this database includes the MAC address of a machine and the intended IP address, this information must be made available to the DHCP server.

With ISC dhcpd [ISC] as shipped with almost all Linux distributions, this would have to be accomplished by extracting the relevant pieces from the LDAP database and constructing a dhcpd.conf. This would have to be done every time the LDAP database changes, must be pushed to each DHCP server, and dhcpd has to be restarted. On a large network this could become quite a hassle to manage.

4 The MetaNet Approach

MetaNet is designed to split the implementation of network services into discreet components which communicate by passing messages. The degree to which the software is split into pieces determines the degree to which it can be integrated with other services.

The fundamental entity is a “message”, which is simply a list of tag/type/value tuples sent to a specific “mailbox”. All control and data transfer takes place via these messages. Objects that encompass various services such as sockets or web servers are simply a set of mailboxes with which other objects interact.

The mailbox names are strings that use a filesystem-style path syntax, allowing for an effectively unlimited namespace if used properly. Common messages include `"/system/socket/new"` to create a new socket, or `"/httpd/request"` when the web server gets a remote request. Most such mailbox names are indeed derived from the name of the object that created them, such as in the previous example where the object is simply named `"/httpd"`.

In order to capture messages sent to a mailbox, a “listener” is attached. This is a function pointer (or reference, in Perl), coupled with another list of tuples to allow the specific instance of that function pointer to be uniquely identified. (Mailboxes themselves in fact have tuples associated with them for the same reason.) Each mailbox maintains an ordered list of these listeners, so that when a message is sent to the mailbox, these listeners are called in order. As a special case, a listener can return an error code that indicates that no more listeners should be called.

There are two conceptually different kinds of mailboxes, with no actual technical difference between them. The difference lies in who is listening and who is sending the messages. In the case of a socket, the `"read"` message is sent by the socket code itself, and the nominal owner of the socket supplies a listener to catch the data received on the socket. The `"write"` request on the other hand works the other way around, with the owner sending the message and the socket’s listener responsible for write()ing the data to the socket. In some cases both the object and external entities can provide listeners, for instance as a means of keeping track of what an object is doing or being told to do.

5 The Main Loop

At the core of any MetaNet application is the main loop, which is responsible for dealing with all the external events the application may listen for. This is basically a glorified select() loop capable of sending messages whenever a socket is available for reading or writing, as well as handling timeout routines. In some cases messages may actually be queued up for delivery directly from the main loop as well. The specific implementation details of the main loop depend entirely on the language and general program style used to construct the MetaNet library. The current Perl prototype does not yet use messages to indicate readable file descriptors or timeouts, though there is no technical reason this cannot be changed to a more consistent style.

6 DHCP and LDAP

In our example scenario, the DHCP server would be written in such a way as to expose many mailboxes through which its internal control passes. Upon creation, it would instantiate a socket to listen for requests. When a request packet arrives, the socket will send a `"read"` message for the server to pick up and translate from the packed DHCP format to a more readable set of tuples, which is then sent as a DHCP-specific message. The DHCP server then must maintain the whole of the DHCP state machine internally, likely using the machine's MAC address ("chaddr" in dhcp) as key.

The state machine will at some point need to find an IP address for the host. Normally this is done by sending a message to a mailbox that might be named `"lookup"` or similar, which another part of the stock DHCP server would be listening to. This code would perform the usual lookup of a free IP address in the pool, or find an address recently associated with that MAC address. It then sends this information back to the state machine by sending a `"lookup-response"` message, for instance. This re-engages the state machine and eventually results in the client being successfully configured.

In order to integrate this DHCP server with and LDAP database, only a few lines of code must be written. The first function is attached as a listener to the `"lookup"` mailbox, and is responsible for triggering the LDAP query. It simply sends a message to a previously instantiated LDAP client object requesting a specific lookup based on the MAC address of the client. It then returns an error code that indicates that it should be the last listener called for this particular message. Since the listener would have been prepended to the mailbox, this precludes the DHCP server's normal lookup routine from being called. The second function listens to the LDAP response mailbox and constructs the necessary message to be sent back into the DHCP state machine.

The only thing missing from this design is the ability for a failed LDAP lookup to fall back to a lookup from the pool. As a quick hack, this could be accomplished by triggering the `"lookup"` message a second time with a tuple in place to indicate to the LDAP glue code that it should not make another attempt to look in the database, but rather fall through and defer to the original DHCP server's listener.

7 Synchronous Operations

The MetaNet architecture as described is obviously highly asynchronous, since the only way to return data from a listener is by sending another message. In many cases, such as the above LDAP attempt, it is very advantageous to be able to execute certain operations synchronously, waiting for some kind of response. DNS lookups are an obvious candidate, since a large system may have to deal with both names and IP addresses may be doing DNS requests in many places. A completely asynchronous system requires that every function that does a DNS request be split into the part before and the part after the request. This could very quickly become a coding nightmare, especially when error cases are introduced.

The solution is to implement a method for blocking on the receipt of a given message.

This is done by attaching a special listener to the mailbox, then sleeping. The listener is responsible for waking up the suspended code and providing the tuples from the message that woke it up.

There are several related mechanisms that can be used to accomplish this. POSIX threads can be used to create a new thread into which the main loop can be "moved" while the original thread waits on a condition variable. A listener is attached to the mailbox being waited on, which wakes up the original thread and ensures that one of the threads finishes before the other continues the main loop. In cases where threads are rarely created, the initial thread must be the one to continue, as killing it would terminate the application.

8 Inter-application Cooperation

If all the services of a given machine were provided by a single daemon with various objects to handle different protocols, a bug in any one service could bring down the entire daemon. This lack of isolation would be the death of any such system. To solve this, there must be a way of separating each service into a separate process, while retaining the ability for these daemons to interact with each other.

In order to do this, MetaNet uses IPC in the form of Unix-domain sockets, with one socket per daemon, residing in a central location. The socket is named after the process, avoiding the pain of having a separate TCP port for each service just for message passing. When a given process needs to send messages to another, it opens up a connection to that socket, which is maintained for the life of the processes.

In order to send a message or attach a listeners to a mailbox in another process, the name of the process is appended with a colon, such as `"dhcpd:/dhcpd/lookup"`. Sending a message to that mailbox will cause a packet to be sent across this on-disk socket to the appropriate application, where it will trigger the listeners in that process. Listeners can be added to this mailbox remotely as well, which is done by inserting a dummy listener into the mailboxes stack with the necessary code to route that message back to the originating process.

Using this method, a standalone, fully self-contained daemon such as dhcpd can have its behavior modified from the outside. As such, the dhcpd could be written in a language such as C, while the glue code to bridge to LDAP could be written in Python or Perl, assuming there is a compliant MetaNet implementation for that language.

Longer term, this IPC mechanism can be extended to support sending messages between different daemons on separate machines. This can be used for statistics gathering, remote configuration, or even to create prototypes of new networking protocols.

9 Implementing a Captive Portal

The first test application for the Perl implementation of MetaNet was a captive portal. The central piece of the portal is a web server, which is easily created with:

```
MetaNet::send("/system/http/server/new",
  name => "/captive/server", port => 5280);
```

A listener is attached to capture the requests made by remote clients:

```
MetaNet::append_new_listener(
  "/captive/server/request",
  \&client_request);
```

Some firewall tables rules are put in place to enable this to function:

```
# drop forwarded packets
# unless allowed
iptables -P FORWARD DROP
# masquerade all the clients
iptables -t nat -A POSTROUTING
  -o $pubdev
  -s $pubnet -j SNAT --to $extIP
# allow any packets that have a
# MARK set
iptables -A FORWARD -i $pubdev
  -o $extdev
  -m mark --mark 0x1 -j ACCEPT
# redir all other HTTP connections
# to myself
iptables -t nat -A PREROUTING
  -i $pubdev -p tcp --dport 80
  -m mark ! --mark 0x1
  -j REDIRECT --to-port 5280
```

This blocks all forwarded traffic that doesn't have a firewall mark set, except for HTTP connections which are forwarded to the portal daemon on port 5280. Next, an HTTP request handler has to be constructed to handle two cases: the client requesting a random page from the Internet, and a client that has been redirected to the login server:

```
sub client_request {
  my ($mbox, $listener, %tuples) = @_;

  if ($tuples{Host} =~
      /$tuples{sockhost}:$tuples{sockport}) {
    serve_page($tuples{clientname},
               $tuples{path});
  } else {
   my $url =
      "http://$tuples{Host}$tuples{path}";
   redirect($tuples{clientname},
   "http://$tuples{sockhost}:5280/?url=$url");
  }

  MetaNet::send("$tuples{clientname}/close");
  return 1;
}
```

The first line gathers the arguments that a listener function gets: the mailbox reference, the listener reference, and a hash containing all the tuples sent as part of the message. In this particular message, the tuples are as follows:

- **$tuples{clientname}** The base name for all mailboxes associated with this client connection
- **$tuples{Host}** The "Host" HTTP header, indicating the intended destination
- **$tuples{path}** The path of the requested file on the site
- **$tuples{sockhost}** The host address of this end of the socket: the portal's address
- **$tuples{sockport}** The port connected to on the portal: 5280

The essence of the code is the check to determine whether the client actually intended to connect to the captive portal itself or not. If it did, it serves a page to the client as needed in order to show the client a login webpage, acceptable usage policy, logos, etc. If it intended to go elsewhere, it is redirected to the index page of the portal:

```
sub redirect {
  my ($client,$newurl) = @_;

  print "redirecting browser to '$newurl'\n";
  $headers{code} = 307;
  $headers{'Location'} = $newurl;
  $headers{'Refresh'} = "1;URL=$newurl";
  $headers{'Content-Type'} = "text/html";
  $headers{data} =
     "<html><head><title>Moved!</title>";
  $headers{data} .=
    "<meta http-equiv=\"Refresh\"
    content=\"1;URL=$newurl\"></head>";
  $headers{data} .= "<body>This page has been
    <a href=\"$newurl\">moved.</a></body>
    </html>";

  MetaNet::send("$client/response", %headers);
}
```

The redirect function uses several distinct tricks to try to get the client to jump to the new page. Once the client has done so, gone through the login sequence, etc., it is time to allow the client to surf the web:

```
sub client_login {
  my ($client) = @_;

  system("iptables -t mangle
    -I PREROUTING 1
    -i $pubdev -s $client->{host}
    -j MARK --set-mark 0x1");
}
```

This function obviously assumes the presence of an object containing various information about the client, or at least its IP address. A simple hash of these client objects is stored globally for quick reference, keyed by the final octet of the client's address.

If that were all there was to the captive portal, eventually every IP address in the range would be wide open, and the purpose would be lost. To avoid this we have to implement a timeout mechanism to determine if the client is still actively using their connection, and if they've been idle for some period of time, log them out automatically. To do this we'll create a periodic timeout:

```
MetaNet::add_timeout(time + 5,
                     \&idle_timeout);
```

The idle_timeout function is rather too involved in `iptables` parsing to reproduce here, but the overall structure is along the lines of:

```
sub idle_timeout
  foreach client
    determine bytes used count
    compare to previous count
    if current != previous
      update last-active time
    if (curtime - last-active) >idle_timeout
      log client out
```

The logout function is simply the same as the login function, with -D instead of -I on the iptables commandline.

There is a long list of features that can be added to this basic portal design. The page handler can support status pages displaying the state of each client. The traffic statistics can be logged to an RRDtool database for future graphing. A DHCP server could be integrated to give the portal a head-start on new clients. MAC-based filtering could be done to make sure logged in IP addresses don't get hijacked when someone leaves.

10 Transparent Proxying of POP and IMAP

The spam problem is a little tougher to solve while still allowing legitimate users to go

about their normal business. Not having the SMTP port open for anyone restricts users to webmail or perhaps secure SMTP, and that can be enough of a deterrent to some people to make the whole experience a waste of time. The goal would then be to find a way to only open up SMTP packet forwarding when it is a reasonably certainty that spamming is not going to be done. While this can never be foolproof, the simple fact that the portal is a complete chokepoint for all traffic makes things much easier to regulate.

A common technique used by corporate sites with mobile users, when a VPN is not available, is to open up SMTP relaying on their server for a short duration immediately following a successful POP or IMAP authentication from that IP address. This technique relies on the assumption that remote users will check their mail before sending mail, or can be easily trained to do so if their mail client doesn't already do this.

The same trick can be used to determine whether the SMTP port should be opened for a given client. If it were possible to detect when the client has made a successful connection attempt to a POP or IMAP server, the port can be opened. The task of determining whether this has actually occurred, however, can be problematic. It requires that the daemon in charge is able to watch the POP or IMAP traffic generated by both the client and the server.

The method attempted involves constructing a transparent proxy for the POP and IMAP protocols. In HTTP/1.1, transparent proxying is made possible by the fact that the protocol requires the host to send the intended destination of the connection as part of the connection itself. The proxy acts like a web server up until the point this information is sent (which happily is immediately upon connection), and promptly makes a connection to the final destination. This is required because iptables REDIRECT does not in any way give the server that handles the redirect any indication of the original destination. Unfortunately, neither POP nor IMAP (or almost any other protocol for that matter) are similarly capable of being transparent proxied as currently defined.

In order to accomplish this, we can take advantage of a feature of the netfilter/iptables called "ipq", which is an iptables target that sends packets up to userspace via a netlink socket. There happens to be a Perl module to interface with this socket, making interfacing quite trivial. We can start with the necessary firewall rules:

```
# masquerade all the clients
iptables -t nat -A POSTROUTING -o $pubdev
  -s $pubnet -j SNAT --to $extIP
# redirect all IMAP connections to our proxy
iptables -t nat -A PREROUTING -i $pubdev
  -p tcp --dport 143 -j REDIRECT
  --to-port 65143
# queue all related SYN packets to userspace
iptables -t mangle -A PREROUTING -s $pubnet
  -p tcp --dport 143 --syn -j QUEUE
# by default, block all SMTP traffic outbound
iptables -A FORWARD -i $pubdev -p tcp
  --dport 25
  -j REJECT
```

To start off the transparent proxy we must create a connection to the netlink socket so we can acquire the SYN packets:

```
my $queue =
  new IPTables::IPv4::IPQueue(
    copy_mode => IPQ_COPY_PACKET,
    copy_range => 64);
```

```
MetaNet::add_fd($queue->get_fd(),
  \&ipq_read,
  undef, undef, queue => $queue);
```

The file descriptor for the netlink socket (patch to IPQueue module required) is added to the main loop's list of file descriptors to listen on. Next, we create a socket to listen for the redirected IMAP connections:

```
MetaNet::send("/system/socket/new",
  name => "/transproxy/socket");
MetaNet::append_new_listener(
  "/transproxy/socket/new_client,
  \&new_client);
MetaNet::send(
  "/transproxy/socket/bind",
  protocol => "tcp", port => 65143);
```

Once the socket is bound and listening, we enter the main loop and wait for something to happen:

```
MetaNet::main();
```

When a packet arrives on the netlink socket, the ipq_read function is called:

```
sub ipq_read {
  my ($fd, %tuples) = @_;
  my ($msg, $ip_header, $src_ip, $dest_ip);

  $msg = $tuples->{queue}->get_message(-1);
  $ip_header = NetPacket::IP->decode(
               $msg->payload());
  $src_ip = $ip_header->{src_ip};
  $dest_ip = $ip_header->{dest_ip};
  $TransProxy::syn_attempts{"$src_ip"} =
     $dest_ip;
  $queue->set_verdict($msg->packet_id(),
                      NF_ACCEPT);
}
```

The client's IP address and the original intended destination are associated in a global hash for future reference. The source port would also be stored, but current experiments show that the REDIRECT target seems to cause the source port to change before it gets to the new destination, making it unusable for the purpose. This means that multiple connections on the same port in very close proximity are likely to be confused with each other.

Almost immediately after the SYN packet is processed, a connection will be established with the daemon via the redirect, triggering new_client:

```
sub new_client {
  my ($mbox, $listener, %tuples) = @_;

  $peerhost = $tuples{peerhost};
  if (defined($TransProxy::syn_attempts{
   "$peerhost"})) {
    build_tunnel($tuples{name},
      $TransProxy::syn_attempts{"$peerhost"},
      143);
    undef $TransProxy::syn_attempts{
          "$peerhost"};
  }
}
```

If the client address of the socket is found in the table of previous attempted connections, a socket tunnel is created between the new connection and the originally intended host. This is done by creating a new socket, connecting it to the server, then attaching functions to the two sockets' "read" mailboxes that send a "write" to the peer socket.

In order to determine if a successful authentication has occurred, the function that handles incoming packets from the server checks each packet for the string "OK LOGIN". If the string is found, the firewall is modified to allow SMTP traffic for that client:

```
if ($data =~ /OK LOGIN/) {
  system("iptables -I FORWARD 1
    -i $pubnet
    -p tcp --dport 25
    -s $client->{peerhost}
    -d $client->{desthost}
    -j ACCEPT");
}
```

A timeout mechanism similar to that used in the captive portal can be used to close the port after a certain amount of inactivity. Even more useful would be interaction between the transparent proxy and the captive portal, automatically closing these holes when the client as a whole times out.

11 Performance

The primary goal of MetaNet has been flexibility from the very beginning. It is not intended as the basis for large highly-scalable systems serving hundreds of clients per second. The highly unstructured string-based design doesn't necessarily lend itself to a highly-performant implementation, though it is entirely possible that some caching and other language tricks could be employed to improve the speed. If a highly-scalable server is needed with the ability to send or receive messages, a small subset could be implemented on top of an existing architecture.

12 Conclusion

The MetaNet architecture provides the ability to construct lightweight services very quickly by building off existing code. More importantly, it allows separate services to be integrated with a minimum of code. Completely new services can be built by gluing together otherwise unrelated subsystems.

The M1 platform from Musenki is the current major target of this work, with the goal of replacing all the software on the machine with MetaNet-based daemons capable of being glued together in previously unknown ways. Such a system would consist of a kernel, init, a shell and basic utilities, and the MetaNet-based daemons. Python is a logical choice for this platform, as the interpreter is less than half a megabyte, and implicitly allows developers to script the machine onboard. Such a machine could then be widely deployed to create the fabled city-wide free wireless network.

13 Acknowledgements

The PersonalTelco group deserves a significant amount of credit for getting me to think about these problems, and then actually *do* something about them. Discussions with Professor Jim Binkley at Portland State University, as well as his class on routing protocols, have been quite helpful. The GStreamer crew also deserves some credit for being "patient" while I worked on this project.

References

[Musenki] *Musenki*
`http://www.musenki.com/`

[PTP] *PersonalTelco Project*
`http://www.personatelco.net/`

[Netfilter] *Linux Netfilter*
`http://netfilter.samba.org/`

[ISC] *Internet Software Consortium*
`http://www.isc.org/`