7.2.4 ERROR: packet administratively filtered.
7.2.5 ERROR: packet filtered by destination.
7.3 Local Experiments
7.3.1 I want to call File.open() within my script.
7.3.2 I want to call Kernel.system() within my script.
7.3.3 I want to use gdb to debug a problem.
7.4 Reporting Bugs

8 Conclusion
## Listings

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1 Traceroute</td>
<td>10</td>
</tr>
<tr>
<td>3.1 TCP Traceroute for Scriptroute</td>
<td>16</td>
</tr>
<tr>
<td>3.2 Basic ping for Scriptroute</td>
<td>17</td>
</tr>
<tr>
<td>3.3 Ping as a train for Scriptroute</td>
<td>18</td>
</tr>
<tr>
<td>5.1 A script for managing the remote execution of a Scriptroute measurement.</td>
<td>29</td>
</tr>
</tbody>
</table>
Chapter 1

Introduction

Scriptroute is a distributed platform for flexible, lightweight network measurements. Scriptroute servers host measurement scripts that send network probes, like TTL-limited UDP messages or ICMP timestamp requests, and collect responses, like ICMP time exceeded or timestamp reply. Hosted measurement scripts can then compute performance attributes or send additional probes to further uncover network properties.

This manual consists primarily of example Scriptroute scripts. It is intended for an audience unfamiliar with the Ruby scripting language. Over the course of this manual, you should understand how to write your own measurements, and how to distribute them across Scriptroute servers for execution.

WARNING The descriptions of some of these examples are out of date. As features have been added to Scriptroute, the example scripts have been updated, but many of the descriptions here have not. That is, the listings (figures) are updated directly from the source, while the line-by-line dissection is not. Sorry.

1.1 Architecture

Scriptroute can be broken into three sets of machines: clients, servers, and home. The home servers\(^1\) maintain a list of other servers and destination-specific policy. Clients connect to a home server using DNS or HTTP to discover a Scriptroute server within a geographic region or autonomous system (a certain provider’s network). Clients can then submit measurement scripts using HTTP Post to a front-end webservice running on a server. The front-end passes the measurement to a safe-mode ruby interpreter that runs in a resource-limited chroot sandbox. The interpreter interacts with the Scriptroute daemon, which decides which and when packets should be sent through a raw socket.

The Scriptroute daemon and interpreter software can also be used locally. That is, a user on the local machine can run an interpreter exempt from some of the resource and safe-mode limitations imposed by the unauthenticated execution model of script submission by HTTP. This can be handy for debugging the installation and for developing new scripts, and also provides a different model for network measurement tools that might be installed on your machine, reducing the number of tools that need to setuid root.

The Scriptroute server runs as a small webservice that invokes a resource limited interpreter using CGI. Submitting a script is as easy as posting to a web form, and tools are provided for remotely executing a script with arguments.

1.2 Ruby

Scriptroute measurement scripts are based on Ruby, a general purpose, object oriented scripting language. We chose Ruby because it is easily embedded, easily sandboxed, and easy to use. For a brief description of Ruby, please see http://www.ruby-lang.org/en/whats.html. “Programming Ruby” by Thomas and Hunt is a more detailed guide and reference, and is available online at http://www.rubycentral.com/book/index.html. If you just want to modify an

\(^1\) Support for multiple “home” servers was added in late 2004.
Figure 1.1: System architecture. I usually refer to the “Network Guardian” as “the Scriptroute daemon”; it has the name “scriptrouted.” DNS policy queries (from an older version of this document) have been replaced with a custom registration protocol that disseminates the entire blacklist rather than query DNS for each destination.

existing measurement, learning Ruby should not be a hassle. If you instead need to write a measurement from scratch, you’ll probably need to read the first few chapters of Programming Ruby.

In this guide, the listings package highlights and formats the code. Keywords in bold are usually real keywords (like if, then, etc.). There are very few such keywords in Ruby, so things that are actually class functions in Kernel, like puts, require, and catch, are also in bold.
Chapter 2

Reading Code

In this chapter, we walk through some of the sample scripts. Think of it as an excessively commented script. We suggest that you first stare at the script for a little while, try to figure out what it does and how it works, then consult the line-by-line descriptions for what isn’t immediately obvious. Then, if you have time, skim the rest.

2.1 Traceroute

The Ruby language features you should to know for this exercise:

**Constants are Capitalized** Variables such as ARGV are constants, and cannot be modified. Classes whose names are capitalized are also immutable, though this is unlikely to surprise anyone.

**Blocks** There are two types of blocks, those enclosed between `do ... end` and those enclosed between `{ ... }`. Don’t worry about the difference.

**Data Types** Variables are dynamically typed. Unlike Perl, the prefix of the variable (eg., “$” or “@”) defines its scope, not its type. `$foo` is a global variable, `@foo` is an instance member variable, and an undecorated `foo` is a local variable to the scope in which it was first assigned. Local variables are sufficient here, so you won’t see any `$’s.

**Semicolons** Statements are terminated by a new-line, unless an expression is incomplete. The listings package that formats code in this document may break lines so that they fit – line numbers show the lines in the original script. Examples:

```
z = ( x
    +
y
  )
z = x + y
z = x + y
```

<table>
<thead>
<tr>
<th></th>
<th>works</th>
<th>works</th>
<th>won’t work</th>
</tr>
</thead>
</table>

Line by line, the traceroute script code in Figure 2.1 does the following.

\(^1\)this may be out of sync as the script changes; the figure contains the up-to-date version, the list describes the code
#! /usr/local/bin/srinterpreter

```ruby
probe = Scriptroute::Udp.new(12)

probe.ip_dst = ARGV[0]

unreach = false

puts "Traceroute to #{ARGV[0]} (#{probe.ip_dst})"

begin
  (1..64).each { |ttl|
    (1..3).each { |rep|
      probe.ip_ttl = ttl
      packets = Scriptroute::send_train([Struct::DelayedPacket.new(0, probe)])
      response = (packets[0].response) ? packets[0].response.packet : nil
      if(response)
        puts '%d %s %5.3f ms' % [ttl, response.ip_src, packets[0].rtt * 1000.0]
        if(response.is_a?(Scriptroute::Icmp))
          unreach = true if(response.icmp_type == Scriptroute::Icmp::ICMP_UNREACH)
        end
      else
        puts ttl.to_s + ' *'
      end
      $stdout.flush
    }
  end
end
```

Listing 2.1: Traceroute for Scriptroute
#!/usr/bin/specruter

# This script is a toy. See rockettrace instead if you're planning to execute any sort of network mapping study.
Invoke the interpreter. This should be familiar to those who have written a perl or shell script before, and the scriptroute interpreter is invoked in the same way. Depending on your installation, this will probably be /usr/bin/scriptroute instead.

probe = Scriptroute::Udp.new(12)
Create a new UDP packet with 12 bytes of data, and assign the result to probe. The 12 bytes of data mimic traceroute, it is possible that some routers do not respond to packets of other than 40 bytes. The Udp class is a member of the Scriptroute module.

probe.ip.dst = ARGV[0]
Set the destination of the packet to be the first argument. The script’s arguments (ARGV) are indexed from 0 as in Perl, unlike C. In scriptroute, hostnames are transparently converted to IP addresses and these IP addresses can be assigned as strings to IP address fields of the packet. That is, the code: p.ip.dst = '127.0.0.1' is equivalent to the C function inet_aton(“127.0.0.1”,&p.ip.dst);. And the code p.ip.dst = 'www.scriptroute.org' does what you want it to do.

unreach = false
Declare a flag to be set when we see an ICMP Destination Unreachable message of type Port Unreachable. This will signify that the traceroute is complete. This is declared here so that its scope spans where it is set and where it is read.

puts "Traceroute to #{ARGV[0]} (#{probe.ip.dst})"
Print a header as if to emulate ordinary traceroute. For output more like that of traceroute (with all rtt’s for a ttl on a single line), look at rocketrace. The hash/brace syntax tells the interpreter to substitute variables into double-quoted strings.

catch(unreachable)do
Start a block. If unreachable is thrown, we’ll leave the block. This is roughly like in Perl or C.

(1..64).each { |ttl |
Let the value ttl climb from 1 to 64. While this is essentially the same as a for loop in C, but it should appear at least a little easier to read.

(1..3).each { |rep|
We’re going to run a probe at each ttl three times. The syntax 1..3 expresses a range object. The Range class supports a method each, which takes a block to be invoked for each element in the range.

probe.ip_ttl = ttl
Much like before with ip.dst, we’re setting a field in the header. This uses the BSD-style names for packet fields, and transparently handles network/ host byte order conversions (though for the ttl field, this is unnecessary).

packets = Scriptroute::send_train([Struct::DelayedPacket.new(0,probe)])
This is the heart of the beast. Working from the inside out, we create a new DelayedPacket structure from 0 and the probe. The delay after which the probe can be sent is set to zero, which asks the server to send the probe as soon as possible. The square brackets create an array, in this case it is an array of just one element. Scriptroute::send_train sends this array of packets out, then returns an array of probe, response pairs, which are themselves pairs of timestamp, packet. We assign the result array to the variable packets.

response = (packets[0].response)? packets[0].response.packet : nil
We’re going to make a shortcut: response will point to the response we might have received, or nil if we didn’t see a response. This is the same conditional assignment syntax that obfuscates C code everywhere.

2 there are some exceptions: remote servers may not support name lookup

11
if (response) then
    If we get a response, we can print the rtt. 'nil' is the same as 'false'. BEWARE: nil and false are not the same as a numeric zero or an empty string, so if (0) will execute the block.

puts '%d %s %5.3f ms' % [ ttl, response.ip_src, packets[0].rtt * 1000.0 ]
    Print a line much like traceroute, starting with the hop number (ttl), then the source address that sent the response, followed by the round trip time in milliseconds. puts, as usual, stands for “put string.” It will print a newline after the string. The \% method of the String class applies sprintf-style formatting. The packets[0].rtt is shorthand for: packets[0].response.time – packets[0].probe.time.

if (response.is_a?(Scriptroute::Icmp)) then
    It’s pretty unlikely that we’d see a non-ICMP response to a traceroute probe, but it is possible if the other end responded with UDP. To prevent a type exception when we dig into the response packet’s ICMP fields, this check is a good idea. is_a? is a member function of Object, the base class of everything else. The if/then syntax is a staple, but note that we’re not using curly braces. If/then/else/elsif are built in to the language, not functions that take blocks.

unreach = true if (response.icmp_type == Scriptroute::Icmp::ICMP_UNREACH)
    In Perl-like conditional form, set the unreach flag if the response if of type 3 (ICMP_UNREACH, or destination unreachable). There is a constant for this, Scriptroute::ICMP_UNREACH, and the rest of the ICMP constants defined in ip_icmp.h. One could change this to print !H or !A (host or address unreachable) when the response code (response.icmp_code) is anything other than the expected port unreachable.

end
    End the if-it’s-an-icmp block.

else
    puts ttl .to_s + ' *'
end
    Traceroute prints either the round trip time or a “*” if no response was received.

$stdout.flush
    To get partial responses, especially when executing remotely, output buffering must be overridden. This is the simplest way to force partial output, not necessarily the most efficient.

} 
    End the block that repeats three times

throw :unreachable if(unreach)
    If this was the last hop we had to trace to, throw the :unreachable exception so that we don’t keep going.

}
    End the block that repeats over each ttl value.

end
    End the catch block.

And now you understand your (and the) first scriptroute program. This version of traceroute really does the absolute minimum any traceroute program should, but it serves as an introduction.

This example is written in an imperative style, though Ruby is perhaps better for functional programming. Later examples will start to introduce such concepts in the context of Scriptroute.
2.2 SProbe

SProbe [?] is a packet-pair bandwidth measurement tool that tries to be conservative about when it makes estimates. The packet stream sent by the original SProbe tool was a train of a configurable number of TCP SYN packets. As the SYN flag is not required for the tool to function, and is more likely to raise the eyebrow of a port-scan- or syn-flood-paranoid system administrator, this implementation of SProbe uses only TCP ACKs. SProbe also has a receiver-side mode that we do not implement here.

The SProbe script is shown in Figures ?? and ??, For this code, we won’t go line-by-line, it might be changed over time. The salient features are as follows.

The Ruby language features you should know for this exercise include the following Array functions.

map
Apply a block to each element of an array. In this example, we use an array describing the size of packets, then use map to construct an array of packets.

sort
Sort an array. An optional block specifies the sort order. send_train returns an array of packets in the order responses were received, which is practically undefined, so we sort by a field set in the sent probes.

detect
Walk through an array, evaluating block until it defines to true.

each
Like map, only doesn’t evaluate to a new array.

join
Create a string from an array, if passed an argument, like ”\n”, it will insert the parameter between each of the elements of the array.
Chapter 3

Modifying Code

3.1 TCP Traceroute

3.2 Variations on a Ping
```bash
#!/usr/local/bin/sr_interpreter

probe = Scriptroute::Tcp.new(0)
probe.ip_dst = ARGV[0]
probe.th_dport = 8000
probe.th_win = 1024
tcp_rst = false

puts "TCPTraceroute to #{ARGV[0]} (#{probe.ip_dst})"

catch (:tcp_rst) do
  (1..16).each { |ttl|  
    (1..3).each { |rep|  
      probe.ip_ttl = ttl  
      packets = Scriptroute::send_train([Struct::DelayedPacket.new(0, probe)])  
      if packets[0].response then  
        response = packets[0].response.packet  
        if (response.is_a?(Scriptroute::Icmp)) then  
          puts '%d %s %5.3f ms % [ ttl, response.ip_src, (packets[0].rtt * 1000.0).to_s ]  
        elsif (response.is_a?(Scriptroute::Tcp)) then  
          puts '%d %s %5.3f ms % [ ttl, response.ip_src, (packets[0].rtt * 1000.0).to_s ]  
        end  
        tcp_rst = true  
      else  
        puts ttl.to_s + ' *'  
      end  
      $stdout.flush  
    }  
  }  
throw :tcp_rst if (tcp_rst)
end
```

Listing 3.1: TCP Traceroute for Scriptroute
#!/usr/local/bin/srinterpreter

probe = Scriptroute::Icmp.new(0) # anything else might not get a response.
probe.ip_dst = ARGV[0]
probe.icmp_type = Scriptroute::Icmp::ICMP_ECHO
probe.icmp_code = 0
probe.icmp_seq = 1

last = Time.now - 1.0;

( 1..10).each { |rep|
  probe.icmp_seq = rep
  delay = 1 - (Time.now-last)
  # puts 'delay: ' + delay.to_s + ' last: ' + last.to_s
  packets = Scriptroute::send_train([ Struct::DelayedPacket.new(1 - (Time.now-last), probe) ])
  if (packets[0].response) then
    response = packets[0].response.packet
    rtt = (response) ? ((packets[0].response.time - packets[0].probe.time) * 1000.0) : '*'
    if(response.is_a?(Scriptroute::Icmp)) then
      puts rep.to_s + ' ' + response.ip_src.to_s + ' %5.3f ms' % rtt
    end
    last = Time.at(packets[0].probe.time);
  else
    puts rep.to_s + ' to ' + probe.ip_dst.to_s + ' timed out'
  end
$stdout.flush
}

Listing 3.2: Basic ping for Scriptroute
#!/usr/local/bin/srinterpreter

packets = Scriptroute::send_train(1..10).map { |rep|
  probe = Scriptroute::Icmp.new(16)
  probe.ip_dst = ARGV[0]
  probe.icmp_type = Scriptroute::Icmp::ICMP_ECHO
  probe.icmp_code = 0
  probe.icmp_seq = rep
  Struct::DelayedPacket.new((rep>1) ? 1 : 0, probe)
}

packets.each { |tup|
  response = tup.response.packet
  rtt = (response) ? ((tup.response.time - tup.probe.time) * 1000.0) : '*'
  puts tup.response.packet.ip_len.to_s + ' bytes from ' +
    tup.response.packet.ip_src +
    ': icmp_seq=' + tup.probe.packet.icmp_seq.to_s +
    ' ttl=' + tup.probe.packet.ip_ttl.to_s +
    ' time=' + rtt.to_s + ' ms';
}

Listing 3.3: Ping as a train for Scriptroute
Chapter 4

Writing Code

This section is intended as a reference manual for Scriptroute-specific classes and constants.

4.1 Interpreter Data Types

Packet classes are comprised of BSD-style header variable names. Each subsection heading is a class, followed by its parent class – below, Scriptroute :: IP is a child of Object, and inherits all of its methods. In this presentation, a list of accessor functions is presented. These accessor functions may include “=” meaning that you can also write to the field when creating a new packet. At right is the default value, as read from a newly instantiated packet. Finally, there are descriptive notes, which at the very least describe the role of the field, but may justify why access to a field is read-only.

4.1.1 Scriptroute :: Ip < Object

Scriptroute :: Ip is the base class from which all other packet classes inherit.

Functions

ip_v
  IP Version. IPv6 support will require a different class

ip_hl
  IP Header Length. IP options may eventually be added, but require a compelling use to be worth the effort.

ip_tos, ip_tos=
  IP type of service.

ip_len
  IP length field. Set when a packet is instantiated to the length of the packet.

ip_id, ip_id=
  IP identifier. Will likely be set randomly by the scriptroute daemon to assist in recognizing ICMP error responses.

ip_off, ip_off=
  IP fragmentation offset. Not sure what to do with this, but likely to be regulated by the scriptroute daemon.

ip_ttl, ip_ttl=
  IP time to live.

ip_p
  IP protocol, set when instantiating a derived class.
4.1.2 Scriptroute::Icmp < Scriptroute::Ip

Scriptroute::Icmp consists of methods and constants for writing ICMP packets. On the todo-list is a method for accessing the encapsulated packet of ICMP error messages.

Functions

```
4

+-----------------+-------------------+
| Method          | Description       |
+-----------------+-------------------+
| ip_v            | IP Version. IPv6  |
|                 | support will       |
|                 | require a         |
|                 | different class.  |
| ip_hl           | IP Header Length. |
|                 | IP options may    |
|                 | eventually be     |
|                 | added, but       |
|                 | require a        |
|                 | compelling use   |
|                 | to be worth the  |
|                 | effort.          |
| ip_tos, ip_tos= | IP type of service. |
|                 | 0                 |
| ip_len          | IP length field. |
|                 | Set when a packet|
|                 | is instantiated  |
|                 | to the length of|
|                 | the packet.      |
| ip_id, ip_id=   | IP identifier.   |
|                 | Will likely be   |
|                 | set randomly by  |
|                 | the scriptroute  |
|                 | daemon to assist |
|                 | in recognizing   |
|                 | ICMP error       |
|                 | responses.       |
| ip_off, ip_off= | IP fragmentation |
|                 | offset. Not sure |
|                 | what to do with |
|                 | this, but likely |
|                 | to be regulated |
|                 | by the scriptroute|
|                 | daemon.          |
| ip_ttl, ip_ttl= | IP time to live. |
|                 | 255               |
| ip_p            | IP protocol, set |
|                 | when instantiating|
|                 | a derived class. |
| ip_sum, ip_sum= | IP header checksum. |
|                 | Set by the        |
|                 | scriptroute       |
|                 | daemon or the     |
|                 | kernel.           |
| ip_src          | Source IP address. |
|                 | Set by the        |
|                 | scriptroute       |
|                 | daemon.           |
| ip_dst, ip_dst= | Destination IP    |
|                 | address. You will|
|                 | want to set this.|
|                 | '0.0.0.0'         |
```
icmp_type, icmp_type= 255
    The ICMP type. You are expected to set this in outgoing probes to one of ICMP_ECHO, ICMP_TSTAMP, ICMP_IREQ. It is set by default to -1 to remind you.

icmp_code, icmp_code= 0
    This will be set to zero or one of the UNREACH, TIMXCEED codes below in responses.

icmp_cksum
    Set by the scriptroute daemon.

icmp_id
    Used in echo and timestamp protocols for matching replies. This is set by the scriptroute daemon, because this field identifies the process sending the packet.

icmp_seq, icmp_seq= 0
    Used in echo for matching replies; may be set by the scriptroute daemon to help in matching, but you should really set it yourself.

icmp_nextmtu, icmp_nextmtu= 0
    Used in Fragmentation Needed messages.

icmp_otime, icmp_otime= 0
    Used in ICMP timestamps.

icmp_rtime, icmp_rtime= 0
    Used in ICMP timestamps.

icmp_ttime, icmp_ttime= 0
    Used in ICMP timestamps.

ip_df, ip_df = false
    (undocumented – send nspring mail; a lack of documentation is not intentional)

to_bytes
    (undocumented – send nspring mail; a lack of documentation is not intentional)

to_s
    (binary)
    Invokes tcpdump’s packet decoding functions to present a string version of the packet. Note that this is in the IP packet class, from which other packets are derived. If invoked on a TimedPacket, the time and the packet are concatenated. If invoked only on the packet, obviously, the timestamp is not present.
### Constants

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ICMP_ECHOREPLY</td>
<td>0</td>
</tr>
<tr>
<td>ICMP_REDIRECT</td>
<td>5</td>
</tr>
<tr>
<td>ICMP_ECHO</td>
<td>8</td>
</tr>
<tr>
<td>ICMP_UNREACH</td>
<td>3</td>
</tr>
<tr>
<td>ICMP_SOURCEQUENCH</td>
<td>4</td>
</tr>
<tr>
<td>ICMP_TIMXCEED</td>
<td>11</td>
</tr>
<tr>
<td>ICMP_PARAMPROB</td>
<td>12</td>
</tr>
<tr>
<td>ICMP_TSTAMP</td>
<td>13</td>
</tr>
<tr>
<td>ICMP_TSTAMPREPLY</td>
<td>14</td>
</tr>
<tr>
<td>ICMP_I REQ</td>
<td>15</td>
</tr>
<tr>
<td>ICMP_I REQREPLY</td>
<td>16</td>
</tr>
<tr>
<td>ICMP_MASKREQ</td>
<td>17</td>
</tr>
<tr>
<td>ICMP_MASKREPLY</td>
<td>18</td>
</tr>
<tr>
<td>ICMP_UNREACH_NET</td>
<td>0</td>
</tr>
<tr>
<td>ICMP_UNREACH_HOST</td>
<td>1</td>
</tr>
<tr>
<td>ICMP_UNREACH_PROTO</td>
<td>2</td>
</tr>
<tr>
<td>ICMP_UNREACH_PORT</td>
<td>3</td>
</tr>
<tr>
<td>ICMP_UNREACH_NEEDFRAG</td>
<td>4</td>
</tr>
<tr>
<td>ICMP_UNREACH_SRCFAIL</td>
<td>5</td>
</tr>
<tr>
<td>ICMP_UNREACH_NET_UNKNOWN</td>
<td>6</td>
</tr>
<tr>
<td>ICMP_UNREACH_HOST_UNKNOWN</td>
<td>7</td>
</tr>
<tr>
<td>ICMP_UNREACH_ISO</td>
<td>8</td>
</tr>
<tr>
<td>ICMP_UNREACH_NET_PROHIB</td>
<td>9</td>
</tr>
<tr>
<td>ICMP_UNREACH_HOST_PROHIB</td>
<td>10</td>
</tr>
<tr>
<td>ICMP_UNREACH_TOSNET</td>
<td>11</td>
</tr>
<tr>
<td>ICMP_UNREACH_TOSHOST</td>
<td>12</td>
</tr>
<tr>
<td>ICMP_UNREACH_FILTER_PROHIB</td>
<td>13</td>
</tr>
<tr>
<td>ICMP_UNREACH_HOST_PRECEDENCE</td>
<td>14</td>
</tr>
<tr>
<td>ICMP_UNREACH_PRECEDENCE_CUTOFF</td>
<td>15</td>
</tr>
<tr>
<td>ICMP_TIMXCEED_INTR</td>
<td>0</td>
</tr>
<tr>
<td>ICMP_TIMXCEED_REASS</td>
<td>1</td>
</tr>
</tbody>
</table>

#### 4.1.3 Scriptroute::Udp < Scriptroute::Ip

Methods to access UDP packet header fields.
Functions

ip_v  
IP Version. IPv6 support will require a different class

ip.hl  
IP Header Length. IP options may eventually be added, but require a compelling use to be worth the effort.

ip_tos , ip.tos =  
IP type of service.

ip_len  
IP length field. Set when a packet is instantiated to the length of the packet.

ip_id , ip.id =  
IP identifier. Will likely be set randomly by the scriptroute daemon to assist in recognizing ICMP error responses.

ip.off , ip.off =  
IP fragmentation offset. Not sure what to do with this, but likely to be regulated by the scriptroute daemon.

ip_ttl , ip.ttl =  
IP time to live.

ip.p  
IP protocol, set when instantiating a derived class.

ip_sum, ip.sum =  
IP header checksum. Set by the scriptroute daemon or the kernel.

ip.src  
Source IP address. Set by the scriptroute daemon.

ip.dst , ip.dst =  
Destination IP address. You will want to set this.

uh.sport  
Set by the scriptroute daemon. The interpreter binds a valid port as it shares code for packet instantiation with the scriptroute daemon. The valid port assigned by the interpreter is not trusted, however, and a different one will be used for the outgoing packet.

uh.dport , uh.dport =  
Destination port. Set to a traceroute-like value by default. May be changed, but likely not to a low numbered port.

uh.ulen  
UDP length. Set when the packet is instantiated to the length of the user data plus the length of the UDP header (8). This packet was instantiated with length 12.

uh.sum, uh.sum =  
As before, checksums are handled by the scriptroute daemon and kernel

tos  
Invokes tcpdump's packet decoding functions to present a string version of the packet. Note that this is in the IP packet class, from which other packets are derived. If invoked on a TimedPacket, the time and the packet are concatenated. If invoked only on the packet, obviously, the timestamp is not present.

to bytes  
(undocumented – send nspring mail; a lack of documentation is not intentional)

ip.df , ip.df =  
(undocumented – send nspring mail; a lack of documentation is not intentional)
4.1.4  Scriptroute :: Tcp < Scriptroute::Ip

Methods to access option-free TCP packet header fields.

Functions

ip_v
   IP Version. IPv6 support will require a different class

ip_hl
   IP Header Length. IP options may eventually be added, but require a compelling use to be worth the effort.

ip_tos, ip_tos=
   IP type of service.

ip_len
   IP length field. Set when a packet is instantiated to the length of the packet.

ip_id, ip_id=
   IP identifier. Will likely be set randomly by the scriptroute daemon to assist in recognizing ICMP error responses.

ip_off, ip_off=
   IP fragmentation offset. Not sure what to do with this, but likely to be regulated by the scriptroute daemon.

ip_ttl, ip_ttl=
   IP time to live.

ip_p
   IP protocol, set when instantiating a derived class.

ip_sum, ip_sum=
   IP header checksum. Set by the scriptroute daemon or the kernel.

ip_src
   Source IP address. Set by the scriptroute daemon.

ip_dst, ip_dst=
   Destination IP address. You will want to set this.

th_sport
   As for uh_sport, set by the scriptroute daemon.

th_dport, th_dport=
   As for uh_dport. Likely policies will exclude traffic to low numbered ports other than 80.

th_seq, th_seq=
   Set the sequence number to something unique if possible.

th_ack, th_ack=
   Set the acknowledgement sequence number to something unique if possible.

th_flags, th_flags=
   Can be set numerically, or using the flag booleans listed below.

flag_fin, flag_fin=
   FIN.

flag_syn, flag_syn=
   SYN. Note that SYN packets are rate limited to a lower level. Do not use SYN packets unless necessary.

flag_rst, flag_rst=
   RST. Note that RST segments do not solicit a response, so are useless in the current scriptroute implementation. They will be filtered
flag_push, flag_push= false
  PSH. If anyone finds a use for this bit, let me know.

flag_ack, flag_ack= true
  ACK. Set by default as acks are generally useful. They solicit a RST, and nobody seems to care when they re-
  ceive em, because they do not create or destroy state.

flag_urg, flag_urg= false
  URG. The urgent pointer is set to zero; this may be useless.

th_win, th_win= 0
  Set the advertised window to something unique if possible.

th_sum, th_sum= 0
  Set by the scriptroute daemon.

ip_df, ip_df= false
  (undocumented – send nspring mail; a lack of documentation is not intentional)

to_bytes
  (undocumented – send nspring mail; a lack of documentation is not intentional)

to_s
  Invokes tcpdump’s packet decoding functions to present a string version of the packet. Note that this is in the
  IP packet class, from which other packets are derived. If invoked on a TimedPacket, the time and the packet are
  concatenated. If invoked only on the packet, obviously, the timestamp is not present.

4.1.5 Scriptroute :: ProbeResponse < Object

send_train returns an array of objects of this type, which include matched probes and responses with timestamps. The
timestamps are given by libpcap, which means they are the same timestamps you might see when running tcpdump.
On Linux, the kernel timestamps packets as they are received, then hands these timestamped packets to libpcap.¹

Functions

probe, probe=
  Extract the probe timestamp and packet as a class TimedPacket. Append .time to access the timestamp, or ap-
  pend .packet to access the packet contents.

response, response=
  Extract the response timestamp and packet. Append .time and .packet as for the probe.

rtt
  Calculate the round trip time in seconds of this packet exchange. Returns nil if there was no response. A com-
  mon idiom might be (packets[0].rtt or "*") to get traceroute like behavior. rtt is essentially sugar for:
  packets[0].response and (packets[0].response.time – packets[0].probe.time) or nil

rtt_s
  (undocumented – send nspring mail; a lack of documentation is not intentional)

to_s
  Invokes tcpdump’s packet decoding functions to present a string version of the packet. Note that this is in the
  IP packet class, from which other packets are derived. If invoked on a TimedPacket, the time and the packet are
  concatenated. If invoked only on the packet, obviously, the timestamp is not present.

¹Feel free to tell me about the accuracy of such timestamps, and how outgoing packets are stamped by libpcap.
4.1.6 Scriptroute :: TimedPacket < Object

Scriptroute :: TimedPacket is the class in which probes and responses are returned to your script by the daemon with associated timestamps.

Functions

packet
Grab the packet object.

time
Grab the time at which the packet was sent (for probes) or received (for responses). This is stored in the builtin class Time, but is easily converted to a string or to a floating point value.

to_s
Invokes tcpdump’s packet decoding functions to present a string version of the packet. Note that this is in the IP packet class, from which other packets are derived. If invoked on a TimedPacket, the time and the packet are concatenated. If invoked only on the packet, obviously, the timestamp is not present.

tsc
(undocumented – send nspring mail; a lack of documentation is not intentional)

4.1.7 Scriptroute

Top-level module environment, including version accessors and the send_train method

Functions

Scriptroute::DaemonVersion
Read a version structure from the interpreter. The version structure consists of major, minor, and revision fields. In this case, DaemonVersion involves a query to the scriptroute daemon. If at some point features are introduced, it may be necessary to change behavior based on the interpreter or daemon version.

Scriptroute::DaemonConfig
Read the daemon configuration file as a hash. Boolean, numeric, and string values are given their appropriate type (not left as strings). For a description of the configuration variables, see the Scriptroute Administrator’s guide. The intent is to allow your script to sanity check the environment its in and catch any problems before time is wasted.

Constants

Scriptroute::InterpreterVersion
It probably violates some principle of object oriented programming to let the InterpreterVersion be a constant while DaemonVersion is not. In this case, the version is an attribute of the interpreter software, so it seems fine for it to be a constant.

4.2 Techniques

This section is a collection of useful techniques discovered over time while using the system.

When running remotely, use $stdout.flush as needed to push intermediate measurement results back to the client. Alternately, use an ensure block to clean up at the end of a measurement and send back collected results. The ensure block is even executed if the script runs out of time and is interrupted (sigint) by the web server.
Chapter 5

Running Code

In this chapter, we describe how to run your scripts, and how to build distributed measurements. The exciting part of scriptroute’s remote access protocol is that you could write distributed measurement tools in any language you please – all you have to be able to do is post to a web form.

5.1 Local

In this section, we describe a how to develop your scripts using the local Scriptroute server. With control over your own processes, you can do different things: lookup hostnames, avoid process state and resource limits, run under gdb when something goes wrong, and print extra debugging messages.

The interpreter takes a few command line arguments.

-c Compile only, don’t execute the code. Ruby is dynamically typed, so this is unlikely to catch anything but parse errors.

-d Skip all resource limits. This makes it easier to run the interpreter under gdb, as gdb will require some file handles, etc.

-h Print a help message.

-u Unsafe mode. Scriptroute is designed around scripts that can be remotely executed, using Ruby’s safe mode. That’s great, but sometimes you might want to write a script that also directly manipulates file without redirecting stdout. Scripts that require unsafe mode can’t be executed remotely via the CGI interface.

-v Print verbose debugging messages.

-V Print the version.

5.2 Remote

In this section, we describe “remotely.rb,” shown in Figure 5.1, which is a script to execute Scriptroute measurements remotely. Larger tools might query several remote servers at a time, but will likely retain the same essential procedure:

1. Take a Scriptroute-Ruby measurement script
2. Look-up all hostnames needed
3. URL-encode the script for transmission
4. Connect to a server at port 3355
5. Submit it via HTTP POST to srruby cgi
6. Collect and process or print the response.

Line by line, “sr-remotely.rb” in Figure 5.1 does the following.

#!/usr/bin/ruby
Invoke the real Ruby interpreter. This script (and the srclient library it invokes) could be reimplemented in whichever language suits you.

require "srclient"
Load the srclient library. If you’ve downloaded and installed ruby and scriptroute, srclient will be in your include path. If not, copy srclient.rb from the scriptroute source directory. To modify Ruby’s load path, use the -I command line option.

ServerName = ARGV[0]
The first parameter is the hostname of the server.

ScriptFileName =
if ( FileTest.exist?(ARGV[1])) then
ARGV[1]
If ScriptFileName, the second command line argument, exists in the local directory, great, but we’ll look for it in PATH otherwise.
else
path = ENV['PATH'].split(':').detect{|dir|
FileTest.exist?(dir + '/' + ARGV[1])
}
Search through the environment variable PATH where each directory is separated by ':'s. The split method in class String creates an array of strings delimited by the parameter. The detect method of an array returns the first element (directory) for which the block evaluates to true.

if ( path == nil ) then
puts "Script file '#{ARGV[1]}’ not found"
exit 1;
We did not find the script file anywhere. We could have written if ( !path ) instead.
end
path + '/' + ARGV[1]
We found the script file; the result of this if/then/else block is assigned to ScriptFileName above, much as in ML.
end

ScriptArgv = ARGV[2..-1]
Assign the rest of the arguments to be the arguments of the script. That is, arguments 2 through last.

puts "host: #{ServerName}’
puts "file : #{ScriptFileName}’
puts "args: #{ScriptArgv.join(‘ ’)} ”
We print out the arguments here as a header to simplify debugging and building larger analyses. The #\{x\} syntax converts x to a string and embeds it in the double-quoted string. In Ruby, single-quoted strings receive no substitution, double quoted strings have their backslashes processed and will evaluate expressions inside #\{…\}. ruby-mode will probably be your friend while you gain a feel for this. join is an Array method that concatenates each of the string elements, the opposite of split.

begin
Start a block that can fail due to timeout.

28
#!/usr/bin/ruby

require "srcclient"

ARGV.length > 1 or exit 1;

ServerName = ARGV[0]

ScriptFileName = 
if ( FileTest.exist?(ARGV[1]) ) then
  ARGV[1]
else
  path = ENV['PATH'].split(':').detect { |dir|
    FileTest.exist?( dir + '/' + ARGV[1])
  }
  if ( path == nil ) then
    puts "Script file '#{ARGV[1]}' not found"
    exit 1;
  end
  path + '/' + ARGV[1]
end

ScriptArgv = ARGV[2..-1]

puts "host: #{ServerName}"
puts "file: #{ScriptFileName}"
puts "args: #{ScriptArgv.join(' ')}"

begin
  puts ScriptrouteClient.new(ServerName).
  set_reverse_dns_lookup.
  query_file(ScriptFileName, ScriptArgv.map { |a|
    (/[A-Za-z][^-w.]+w$/m.match(a) != nil) ?
    (IPSocket.getaddress(a) or a) : a
  }).join("\n")
rescue TimeoutError
  puts "ERROR: timed out — scriptroute should never take longer than 60 seconds."
  exit 1; # we failed.
end

Listing 5.1: A script for managing the remote execution of a Scriptroute measurement.
puts ScriptrouteClient.
new(ServerName).
Create a new ScriptrouteClient (from the srclient.rb library) that connects to the server named by ServerName.
Note the trailing dot: we’re not done yet.

set_reverse_dns_lookup.
ScriptrouteClient objects can process the output of the remotely executed script to automatically do reverse-name lookups. That is, it matches IP addresses and calls gethostbyaddr on each. This method enables this feature on the newly created client. Again, the trailing dot will take the return value from the set_reverse_dns_lookup method, (which in the method is "self", evaluating here to the object).

query_file (ScriptFileName, ScriptArgv.map { |a|
  (/^[A-Za-z][-$w.]+\w$/.match(a)! nil)?
  (IPSocket.getaddress(a)or a) : a
}).
If any of the script arguments look like they could be hostnames (start with alpha characters, and consist of letters, numbers, dots and dashes), we try to lookup the hostname using IPSocket.getaddress. If the lookup fails, pass the text unaltered.

join ("n")
The output of query_file is an array of output lines. We concatenate them together to be printed, rather than process them further. That’s a backslash n, as in C for newlines; Latex sometimes doesn’t preserve the backslash when formatting.

rescue TimeoutError
query_file can throw a TimeoutError if it appears that the query (or connection to the server) has timed out. We’ll handle this particular error using a rescue statement.

puts "ERROR: timed out — scriptroute should never take longer than 60 seconds."
Scriptroute servers are designed around kicking out remotely executed scripts after about 30 seconds. Therefore, if we’ve had to wait for a response longer than a minute, something has gone wrong.

exit 1; # we failed.
Exit with non-zero status to indicate an error.

end
End the error handling block.

5.3 require ’srclient’

A little utility library is included. After gaining some experience using Scriptroute, srclient.rb was written to refactor common code to a central module. It is installed in Ruby’s path, if you’re able to install Scriptroute. Otherwise, copy srclient.rb to your working directory.

5.3.1 Class Methods

ScriptrouteClient . serverlist
Returns an array of all public Scriptroute servers listed at www.scriptroute.org:3967. This list is cached to avoid having to ask the server again within the same process. This list is probably not as useful as sitelist, below.

ScriptrouteClient . sitelist
Returns an array of public Scriptroute servers listed at www.scriptroute.org:3967 that are hosted in different /24 prefixes. (A /24 is a unit of address allocation that frequently corresponds to a single site. Servers within a /24 are likely to be redundant and unnecessary.)

ScriptrouteClient . new(server)
Returns a new ScriptrouteClient object, connected to a particular server. This connection may or may not be persistent (I don’t know and haven’t had occasion to test).
5.3.2 Object Methods

set_reverse_dns_lookup
Sets a flag that causes your script’s output to be post-processed to insert hostnames with IP addresses. That is, when it sees 128.95.2.30, it will replace it with fretless.cs.washington.edu (128.95.2.30).

query_script(string)
Sends a script to the server using HTTP POST. A 100-second timeout is used to avoid blocking too long on an unresponsive server. (so query_script may throw a TimeoutError) This function returns an array of lines output by the script.

query_file(filename, arguments)
Sends a script, read from a file, to a remote server, passing arguments as ARGV[0], ARGV[1], etc. The underlying mechanism is to define ScriptRemoteArgs to be the arguments passed to query_file, then substitute ScriptRemoteArgs in place of ARGV. The composed script is then passed to query_script, meaning that query_file also returns an array of lines output by the script. This function is the heart of sr-remotely.rb, but can be used to build larger tools.
Chapter 6

Policy

There has been some confusion about what behaviors are permitted and what behaviors are forbidden, and how to circumvent any policy restrictions. “Policy” refers to any administrative limit that prevents a measurement script from doing anything that is otherwise permitted by the implementation. For example, packets are prohibited from being sent to the local subnet in the default policy, but the implementation doesn’t care what the destination is.

It is confusing because there are two convolved issues. First, why is behavior X prohibited? (“but I want to send TCP SYN packets to everyone in the internet!”) Second, which scriptroute component applies the policy?

The decomposition is straightforward. The scriptroute core daemon protects the network. It does not care who you are, whether you’re running locally (using the local interpreter) or remotely (via the web interface). This is a feature of Scriptroute’s security model. If it’s okay to do, anyone should be able to do it. If it’s not okay, it should be forbidden.

The interpreter protects the host. It is the environment for remote code execution and must apply resource limits and safe mode (no filesystem access) restrictions. Now, if you’re running locally, under your own user account, these restrictions are not necessary. You can disable them by using the “-u” (unsafe mode) and “-d” (no resource limits)\(^1\) options. You can disable these options because it’s only your account that is hurt when a script does something unsafe. Using features enabled by the “-ud” option set will make it impossible to run your script remotely, which is why these features are disabled by default.

The front end web-server (thttpd) protects the availability of remote service. It will kill off any scripts executed on behalf of remote users after they’ve taken too long. This is a feature compiled into the webserver and is difficult to change or disable.

But I’m running as root, shouldn’t I be able to send whatever I want? To do so, you’ll need to change the policy in scriptrouted.conf to include what you want to do as “safe” behavior. You accept responsibility for any damage this change in policy causes. That means, if you violate your service agreement with your provider by sending bad traffic, don’t blame us.

The scriptroute daemon doesn’t automatically give you license to send whatever you want when running as root philosophically because unsafe traffic is unsafe traffic. The implementation doesn’t support any user-based authentication, so whether you’re running as an ordinary user, as root, or as nobody, makes no difference to the daemon, and consequently to the traffic you are permitted to send.

\(^1\)The “-d” flag is needed when running under the debugger, hence its name.
Chapter 7

Troubleshooting

7.1 Remote Execution

7.1.1 ERROR: You’re already running a measurement

You, or one of your friends on the same machine, is already running a measurement on a remote Scriptroute server. Since the number of concurrent experiments is limited, each client is only allowed one at a time.

If you see this error in normal operation, it means some state was left around at the server. It should time out in a few seconds as your last experiment completes. If not, report it as a bug.

7.1.2 ./srclient.rb:7:in ‘require’: No such file to load – net/http

Your installation of ruby is incomplete. Ruby is actually looking for net/http.rb (though if it found net/http.so, it would use that instead, so its error message isn’t specific).

You have the following choices, in decreasing order of preference.

- Get ruby installed by your system administrator. It should be a simple package that is part of your distribution.
- Install ruby yourself, using make install from the ruby directory.
- Install ruby locally, using ./configure --prefix=$HOME/ruby; make INSTALL, then add $HOME/ruby/bin to your path.

7.1.3 ERROR: timed out – scriptroute should never take longer than 60 seconds.

The Scriptroute server will limit the duration of your script execution. When it does, your script will be “Interrupted” (sent SIGINT) and a stack trace will result. The client can take advantage of this time limit to set a timeout of, currently, 100 seconds – if the server takes longer than that to respond, it is not worth waiting any longer, the server is likely dead. If you see this “timed out” error, it means that the remote server didn’t bother to respond when given much more than a minute – essentially that the connection timed out, not the experiment.

If this happens when connecting to several servers listed at http://www.scriptroute.org:3967/, it may mean that your client is behind a firewall, and is unable to connect to port 3355 on the remote server. To test this, run telnet <server ip> 3355. If you see a connection timed out message, your traffic is probably being filtered. In a future version of the Scriptroute client, you may be able to set the $HTTP_PROXY environment variable so that you can traverse the firewall using a web proxy.
7.2 Sending Packets

7.2.1 ERROR: Packet was not actually sent: pcap overloaded?

Also, “didn’t see the packet leave: pcap overloaded?”

Every once in a while, libpcap will fail to capture some packets. This happens when the buffer between kernel and application fills up. This buffer can fill when the rate of traffic is high and the Scriptroute daemon is waiting for a timeslice.

Scriptroute can observe that something went wrong when it calls sendto() to send your packet but never sees the outgoing packet using libpcap. In contrast, it can’t tell if a response packet was missed. Consider this error to be a sign that the machine you’re using is too heavily loaded to be of much use.

send_train will return a valid ProbeResponse object, but both probe and response values will be nil. It seemed likely that any processing of a packet exchange would start with checking if a response was received; these semantics simplify a script that checks for a response before processing further. However, it means that to estimate loss rate, you’ll need to check that the probe packet was actually sent. Note that it may have been (and probably was) sent, it’s just that Pcap missed it.

7.2.2 I asked for 1ms spaced packets and get 10ms spaced clumps!

You’re running on a machine with processor contention: there’s too much other stuff running on the machine. If scriptroute has to wait for another CPU-bound process to take its timeslice (10ms), it can’t be awake to send your packet when you want it.

The time when your probe packet was sent is returned by send_train, so if spacing is important, check the eventual spacing and discard unsatisfactory measurements.

Scriptroute could busy-wait for short intervals to try and fight other processes to be awake when you scheduled a packet. Such a patch would be welcome as a configuration option, but it only postpones and does not actually solve the problem. Operating systems such as RTLinux have features to schedule fine-grained events that could also be used to enhance Scriptroute.

7.2.3 ArgumentError: Invalid address in reserved 0.0.0.0/8

Scriptroute is unable to send packets to the network 0.0.0.0/8. This prefix is reserved, and other applications face a similar problem (just try to telnet 0.0.0.1.)

This exception prevents you from trying to send to an invalid address.

7.2.4 ERROR: packet administratively filtered.

This Scriptroute host is configured not to send your packet. More detail about the specific policy is not provided, though it is likely that the filtering policy has not been changed from the default.

7.2.5 ERROR: packet filtered by destination.

The destination of your probe has requested not to receive any Scriptroute traffic, or not to receive traffic of the specific type you wanted to send.

7.3 Local Experiments

If you’re running Scriptrouted on your own machine, you can run a “local” experiment. The site security policy still applies: you can only send packets of an approved type at an approved rate, however, you can get around the resource limits as srinterpreter will run as your own user.
7.3.1 I want to call File.open() within my script.

To disable Ruby’s safe mode, use `srinterpreter -u` or change the line at the top of your script to:

```
#!/usr/bin/srinterpreter -u
```

While you’re writing a local-only script, you may also want to use the `-d` option to disable resource limits, some of which are used to back up Ruby’s safe mode.

7.3.2 I want to call Kernel.system() within my script.

You’ll also have to disable the resource limits:

```
#!/usr/bin/srinterpreter -ud
```

A resource limit is used to control the number of processes spawned, which acts as a backup limit to ruby’s safe mode. This is why there is a resource limit that prevents the use of fork (as used by system).

7.3.3 I want to use gdb to debug a problem.

GDB requires some extra file handles, so `srinterpreter` must be run in debug mode:

```
gdb /usr/bin/srinterpreter
   run -d /usr/bin/sr-traceroute www.cnn.com
```

7.4 Reporting Bugs

Send mail to mailto:bugs@scriptroute.org.

Please include the Scriptroute version number, a copy of the script that triggered the bug, relevant lines from syslog, and anything else you feel would help reproduce the bug or find a way to fix it. However, the above address is a mailing list, so do not send large attachments.
Chapter 8

Conclusion

Wasn’t this fun? I thought so.

For feedback, contact mailto:nspring@cs.washington.edu.