xh

Spencer Tipping

June 8, 2014

Contents

Part I

xh runtime

Chapter 1

Self-replication

Listing 1.1 boot/xh-header

```
#!/usr/bin/env perl
BEGIN {eval(our $xh_bootstrap = q{
# xh | https://github.com/spencertipping/xh
# Copyright (C) 2014, Spencer Tipping
# Licensed under the terms of the MIT source code license

# For the benefit of HTML viewers (long story):
# <body style=’display:none’>
# <script src=’http://spencertipping.com/xh/page.js’></script>
use 5.014;
package xh;
our %modules;
our @module_ordering;
our %eval_numbers = (1 => ’$xh_bootstrap’);

sub with_eval_rewriting(&) {
  my @result = eval {$_[0]->(@_[1..$#_])};
  $@ =~ s/\(eval (\d+)\)/$eval_numbers{$1}/eg if $@;
  die $@ if $@;
  @result;
}

sub named_eval {
  my ($name, $code) = @_;
  $eval_numbers{$1 + 1} = $name if eval(’__FILE__’) =~ /\(eval (\d+)\)/;
  with_eval_rewriting {eval $code};
}

our %compilers = (pl => sub {
  my $package = $_[0] =~ s/\./::/gr;
  named_eval $_[0], "{package ::$package;\n$_[1]\n}";
  die "error compiling module $_[0]: $@" if $@;
});

sub defmodule {
  my ($name, $code, @args) = @_;
  chomp($modules{$name} = $code);
  push @module_ordering, $name;
  my ($base, $extension) = split /\.(\w+$)/, $name;
  die "undefined module extension ’$extension’ for $name"
    unless exists $compilers{$extension};
  $compilers{$extension}->($base, $code, @args);
}

chomp($modules{bootstrap} = $::xh_bootstrap);
undef $::xh_bootstrap;
```

At this point we need a way to reproduce the image. Since the bootstrap code is already stored, we can just wrap it and each defined module into an appropriate BEGIN block.

Listing 1.2 boot/xh-header (continued)

```
sub image {
  my @pieces = "#!/usr/bin/env perl";
  push @pieces, "BEGIN {eval(our \$xh_bootstrap = <<’_’)}",
                $modules{bootstrap},
                ’_’;
  push @pieces, "BEGIN {xh::defmodule(’$_’, <<’_’)}",
                $modules{$_},
                ’_’ for @module_ordering;
  push @pieces, "xh::main::main;\n__DATA__";
  join "\n", @pieces;
}
})}
```

Chapter 2

SSH routing fabric

xh does all of its distributed communication over SSH stdin/stdout tunnels (since remote hosts may have port forwarding disabled), which means that we need to implement a datagram format, routing logic, and a priority-aware traffic scheduler.

For simplicity, the only session type that's supported is RPC. The request and response each must fit into a single packet, which is size-limited to 64 kilobytes excluding the packet header. The fabric client will deal with larger requests and responses, but it will cause additional round-trips.

Listing 2.1 src/fabric.pl

```
package xh::fabric;
use Sys::Hostname;
use Time::HiRes qw/time/;
use Digest::SHA qw/sha256/;
# Mutable state space definition for the routing fabric. You should create
# one of these for every separate xh network you plan to interface with.

sub fabric_client {
  my ($name, $bindings) = @_;
  $name //= $ENV{USER} . ’@’ . hostname . ’.local’;
  return {rpc_bindings     => $bindings,
          instance_name    => $name,
          instance_id      => 0,
          edge_pipes       => {},
          network_topology => {},
          send_queue       => [],
          blocked_rpcs     => {},
          routing_cache    => {}};
}

sub fabric_rpc_bind {
  my ($state, %bindings) = @_;
  my $bindings = $state->{rpc_bindings};
  $bindings->{$_} = $bindings{$_} for keys %bindings;
  $state;
}

sub fabric_is_initialized {
  my ($state) = @_;
  !!$state->{instance_id};
}
use constant header_pack_format        => ’C32 N N d n C C n’;
use constant signed_header_pack_format => ’C32 ’ . header_pack_format;

use constant signed_header_length    => 32 + 32+4+4+8+2+1+1+2;
use constant header_signature_length => 32;

our $nonce_state = sha256(time . hostname);
sub packet_nonce {$nonce_state = sha256(time . $nonce_state)}

sub encode_packet {
  my ($state, $destination_name, $message_type, $priority, $deadline) = @_;
  die "data is too long: " . length($_[5]) . " (max is 65535 bytes)"
    if length $_[5] >= 65536;

  my $destination_id = $state->{routing_cache}{$destination_name};
  return undef unless defined $destination_id;

  my $header = pack header_pack_format, packet_nonce,
                                        $state->{instance_id},
                                        $destination_id->{endpoint_id},
                                        time,
                                        length $_[5],
                                        $message_type,
                                        $priority,
                                        $deadline;
  my $packet = $header . $_[5];
  sha256($packet) . $packet;
}

sub decode_packet_header {
  unpack signed_header_pack_format, $_[0];
}

sub signature_is_valid {
  my ($sha) = decode_packet_header $_[0];
  $sha eq sha256(substr $_[0], header_signature_length);
```

```
}
use constant {forgetful_rpc  => 0,
              functional_rpc => 1,
              rpc_reply      => 2,
              rpc_error      => 3,
              routing_error  => 4};
use constant {realtime_priority => 0,
              high_priority     => 16,
              normal_priority   => 256,
              low_priority      => 32768};

use constant {realtime_deadline          => 0,
              imperceptible_deadline     => 20,
              short_interactive_deadline => 50,
              long_interactive_deadline  => 100,
              process_blocking_deadline  => 250,
              background_deadline        => 2000,
              far_deadline               => 32768};



\lstset{caption={src/fabric/dependencies.pl},name={src/fabric/dependencies.pl}}\begin{perlcode}
use Sys::Hostname;
use Time::HiRes qw/time/;
use Digest::SHA qw/sha256/;
```

Listing 2.2 src/fabric/state.pl

```
# Mutable state space definition for the routing fabric. You should create
# one of these for every separate xh network you plan to interface with.

sub fabric_client {
  my ($name, $bindings) = @_;
  $name //= $ENV{USER} . ’@’ . hostname . ’.local’;
  return {rpc_bindings     => $bindings,
          instance_name    => $name,
          instance_id      => 0,
          edge_pipes       => {},
          network_topology => {},
          send_queue       => [],
          blocked_rpcs     => {},
          routing_cache    => {}};
}

sub fabric_rpc_bind {
  my ($state, %bindings) = @_;
  my $bindings = $state->{rpc_bindings};
  $bindings->{$_} = $bindings{$_} for keys %bindings;
  $state;
}

sub fabric_is_initialized {
  my ($state) = @_;
  !!$state->{instance_id};
}
```

2.1 Packet format

Packets and headers are written in binary, and all multibyte numbers are big-endian. The structure of a packet is:

```
data+header SHA-256:        32 bytes
packet identity nonce:      32 bytes         \
source xh instance ID:      4 bytes          |
destination xh instance ID: 4 bytes          |
packet creation time:       8 bytes (double) |
data length:                2 bytes          | SHA applies to these bytes
message type:               1 byte           |
priority:                   1 byte           |
deadline:                   2 bytes          |
data:                       <= 65535 bytes   /
```

The only reason we represent packet creation time as a double rather than as a 64-bit integer is that 64-bit integer support is not guaranteed within Perl. As a result, we have a somewhat awkward situation where all absolute times are encoded as doubles and all deltas as integers.

Listing 2.3 src/fabric/packet-format.pl

```
use constant header_pack_format        => 'C32 N N d n C C n';
use constant signed_header_pack_format => 'C32 ' . header_pack_format;

use constant signed_header_length    => 32 + 32+4+4+8+2+1+1+2;
use constant header_signature_length => 32;

our $nonce_state = sha256(time . hostname);
sub packet_nonce {$nonce_state = sha256(time . $nonce_state)}

sub encode_packet {
  my ($state, $destination_name, $message_type, $priority, $deadline) = @_;
  die "data is too long: " . length($_[5]) . " (max is 65535 bytes)"
    if length $_[5] >= 65536;

  my $destination_id = $state->{routing_cache}{$destination_name};
  return undef unless defined $destination_id;

  my $header = pack header_pack_format, packet_nonce,
                                        $state->{instance_id},
                                        $destination_id->{endpoint_id},
                                        time,
                                        length $_[5],
                                        $message_type,
                                        $priority,
                                        $deadline;
  my $packet = $header . $_[5];
  sha256($packet) . $packet;
}

sub decode_packet_header {
  unpack signed_header_pack_format, $_[0];
}

sub signature_is_valid {
  my ($sha) = decode_packet_header $_[0];
  $sha eq sha256(substr $_[0], header_signature_length);
}
```

`message type` is one of the following values:

0 Forgetful RPC request. The receiver should execute the code, but the sender will not await a reply. This is used internally by xh to maintain routing graph information and clock offsets.

1 Functional RPC request. This indicates that the receiver should execute the given code, encoded as text, and send a reply. The code may contain references that require further RPCs to be issued.

2 RPC reply after a successful invocation. The return value of the function is encoded in quoted form, and may require further dereferencing via RPC.

3 Callee-side RPC error; the reply is a partially-evaluated quoted value, where any unevaluated pieces represent errors.

4 Routing error or timeout; the routing fabric generates this to indicate that it has given up on getting a successful reply. If this happens, the sender will automatically re-send the RPC unless the deadline has expired.

Listing 2.4 `src/fabric/message-types.pl`

```
use constant {forgetful_rpc  => 0,
              functional_rpc => 1,
              rpc_reply      => 2,
              rpc_error      => 3,
              routing_error  => 4};
```

`priority` and `deadline` are used for scheduling purposes. Zero is the highest priority, 65535 is the lowest. The deadline is used to indicate how time-sensitive the packet is; the queueing order function used by the scheduler is $\frac{2^c}{s}$, where:

$$\begin{aligned}
c &= \frac{\Delta t - d}{16 + p} \\
\Delta t &= \text{ms since packet was originally sent} \\
d &= \text{the deadline} \\
p &= \text{the priority} \\
s &= \text{header + data size in bytes}
\end{aligned}$$

Δt is an estimated quantity, since hosts will not, in general, have synchronized clocks. However, xh uses a protocol similar to NTP to estimate clock offsets for each instance. These clock offsets are used to coordinate instances on different hosts. (See 2.3.)

Listing 2.5 `src/fabric/standard-priorities.pl`

```
use constant {realtime_priority => 0,
              high_priority     => 16,
              normal_priority   => 256,
              low_priority      => 32768};

use constant {realtime_deadline          => 0,
              imperceptible_deadline     => 20,
              short_interactive_deadline => 50,
              long_interactive_deadline  => 100,
              process_blocking_deadline  => 250,
              background_deadline        => 2000,
              far_deadline               => 32768};
```

2.2 Routing logic

I assume the topology of xh instances will fit into memory. This won't be a problem for most installations; in practice, xh should be able to easily manage (and transfer data between) many hundreds of machines without slowing down. Each xh instance maintains a copy of the full routing graph, which includes information about edge timings.

The routing logic's job is to decide how to most effectively get packets from point A to point B, which, more formally, means minimizing the expected sum of delay costs. Doing this well involves a few factors:

1. An edge's average latency and throughput.
2. The variance in an edge's latency and throughput, absent xh traffic.
3. The impact of traffic on an edge's latency and throughput.

All of these are continuously measured and periodically propagated as network topology metadata.

Listing 2.6 `src/fabric/routing-rpcs.pl`

2.3 Clock offset estimation