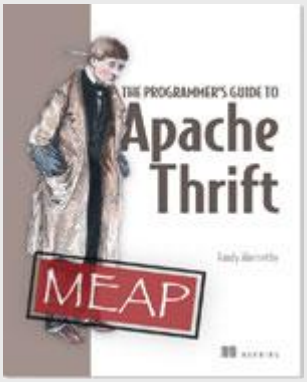

Apache Thrift and Ruby

By Randy Abernethy

In this article, excerpted from The Programmer's Guide to Apache Thrift, we will install Apache Thrift support for Ruby and build a simple Ruby RPC client and server.

Ruby is a flexible programming language used for systems administration scripting, web programming and general purpose coding. Many important DevOps tools are written in Ruby, such as Puppet, Chef, and Vagrant. Ruby is behind one of the most popular web frameworks, Rails and Ruby is also a key language in many test frameworks, including RSpec and Cucumber. Ruby provides the best features of Perl's string processing and adds objects, functional programming features, reflection and automatic memory management. When combined with Apache Thrift Ruby becomes an effective RPC service platform as well as an incredibly versatile language for RPC service mocking and testing.

To build Apache Thrift clients and servers in Ruby we will need to install the Apache Thrift IDL Compiler and the Apache Thrift Ruby library package. In Ruby packages are called gems and can be installed using the RubyGems "gem" package manager. There are over 90,000 Ruby gems hosted on RubyGems.org, the central repository for publicly available Ruby packages. A search for "thrift" on the RubyGems.org site produces a list of over 30 gems, including the official Apache Thrift Ruby library, which is named "thrift."

If you have a working Ruby installation, it is fairly easy to install support for Apache Thrift. Here's an example:

```
$ sudo gem install thrift
Building native extensions.  This could take a while...
Successfully installed thrift-0.9.2.0
1 gem installed
Installing ri documentation for thrift-0.9.2.0...
Enclosing class/module 'thrift_module' for class BinaryProtocolAccelerated
      not known
Installing RDoc documentation for thrift-0.9.2.0...
Enclosing class/module 'thrift_module' for class BinaryProtocolAccelerated
```

```
not known
```

The example above installs the latest version of the Apache Thrift gem, which contains the library code needed to execute Ruby based Apache Thrift clients and servers. The Apache Thrift Ruby library includes an optional C language native extension designed to improve serialization performance. This is automatically built (if possible) by the gem installer. The Ruby Development packages are typically required to build native extensions. To install full Ruby support on an Ubuntu 14.04 system you could use a command something like the following: `$ sudo apt-get install ruby ruby-dev`

Once we have the thrift gem installed, our next step in building the hello client and server is to generate client and server stubs for our helloSvc service.

```
$ thrift --gen rb hello.thrift                                    #A
$ ls -l gen-rb                                                    #B total 12
-rw-r--r-- 1 randy randy  161 Oct 26 17:37 hello_constants.rb
-rw-r--r-- 1 randy randy 1647 Oct 26 17:37 hello_svc.rb
-rw-r--r-- 1 randy randy  139 Oct 26 17:37 hello_types.rb
  $ head gen-rb/hello_svc.rb                                      #C
#
# Autogenerated by Thrift Compiler (1.0.0-dev)
#
# DO NOT EDIT UNLESS YOU ARE SURE THAT YOU KNOW WHAT YOU ARE DOING
#
require 'thrift' require
'hello_types'

  module HelloSvc                                                          #D
```

The thrift command above #A generates Ruby code to support all of the Apache Thrift interface components described in hello.thrift. The generated code will be emitted in a subdirectory called "gen-rb", as displayed in the listing above #B. The top few lines of the hello_svc.rb are displayed above with the head command #C. As you can see from the comments, this file should be treated as read only, to change it we should change the IDL source and recompile with the thrift compiler. The output also show that Ruby packages each service within a module, module HelloSvc in our case #C.

The IDL compiler generates three files to support our IDL definitions, *_constants.rb files contain all of the constants defined in the IDL (none in our case), *_types.rb files contain all of the user defined types defined in the IDL (again, none in our case) and *.svc.rb files define the service client and server stubs required to support the services defined in the IDL.

In Ruby the svc file contains a module for each IDL service, "HelloSvc" in our case. A service module contains a "Client" class used by service clients and a "Processor" class used by servers to process service requests. Each service method will also have a class definition for the arguments passed to the server and the result returned from the server. These classes are for use by the Client and the Processor and are not typically accessed directly by user code.

Ruby Server

Now that we have the Thrift libraries installed and our service Client/Processor code generated we can code up a quick RPC server. Here's an example:

Listing 1 ~thriftbook/part3/script/ruby/hello_server.rb

```
#!/usr/bin/env ruby


require 'thrift'
$:.push('gen-rb')
require 'hello_svc'


class HelloHandler  def
getMessage(name)
return 'Hello ' + name
end end

  port =

9095

handler = HelloHandler.new() proc =
HelloSvc::Processor.new(handler) trans_ep =
Thrift::ServerSocket.new(port) trans_buf_fac =
Thrift::BufferedTransportFactory.new() proto_fac
= Thrift::BinaryProtocolFactory.new()
server = Thrift::SimpleServer.new(proc, trans_ep, trans_buf_fac, proto_fac)


puts "Starting server on port #{port}..."
server.serve()
```

The server source begins with a require statement to include Apache Thrift library support #A, then adds the gen-rb directory to the library path #B before requiring the IDL Compiler generated hello_svc.rb file. This gives us access to both the core thrift library and the code needed to implement the helloSvc.

Thrift takes care of all of the service wiring leaving only the implementation of the service methods to us. The HelloHandler class provides a method for each method defined in the Apache Thrift IDL helloSvc #D.

After defining the service methods in the Handler we can create a server to host the service. In this example the server is configured to run on port 9095 #E. The IDL Compiler generated processor for the helloSvc is initialized #F with a new instance of the handler class #E. The server will use the processor to pass network calls to the handler methods and return the responses.

This example server uses a common Apache Thrift protocol stack:

- TSocket
- TBufferedTransport
- TBinaryProtocol

The TSocket layer provides TCP communications with the client and on the server side it is implemented with the ServerSocket class in Ruby #H. The server transport will listen for new connections and create a new socket for each connecting client. The BufferedTransportFactory allows the server to create transport buffers for each connection, collecting response data together until such time as a complete RPC response is serialized and ready to transmit back to the connected client #I. The BinaryProtocolFactory is similar in that is allows the server to create a new binary protocol serializer for each connecting client #J.

With the I/O stack created we are ready to construct the server which will orchestrate all of these components. In this example we use the SimpleServer #K. The SimpleServer will only accept one client connection at a time, queuing connection requests until the current client disconnects. After constructing the server with an initialized processor, transport stack and protocol we can run it by calling the serve() method #L.

Running the Ruby server should look something like this:

```
$ ruby hello_server.rb
Starting server on port 9095...
```

A Ruby Client

Next we'll take a look at a basic Ruby client we can use to test our server.

Listing 2 ~thriftbook/part3/script/ruby/hello_server.rb

```
#!/usr/bin/env ruby

require 'thrift'                                                    #A
$:.push('gen-rb')                                                   #A
require 'hello_svc'                                                 #A

begin                                                               #B
  trans_ep = Thrift::Socket.new('localhost', 9095)                  #C
  trans_buf = Thrift::BufferedTransport.new(trans_ep)               #C
  proto = Thrift::BinaryProtocol.new(trans_buf)                     #C
  client = HelloSvc::Client.new(proto)                              #D

  trans_ep.open()                                                   #E
  res = client.getMessage('world')                                  #F
  puts 'Message from server: ' + res
  trans_ep.close()                                                  #G
rescue Thrift::Exception => tx                                      #H
  print 'Thrift::Exception: ', tx.message, "\n"
end
```

Things should look fairly familiar. The client has the same dependencies as the server #A, requiring the thrift library and the hello_svc boilerplate. The I/O stack is also identical, including the Socket end point transport and a transport buffer layer, as well as the binary serialization protocol #C.

The main client code listing is wrapped in a begin rescue block #B. The rescue clause will trap any Thrift exceptions and display them to the console #H.

On the client side, to connect with the server we open the socket, using the transport open() method #E. With the connection open, we can use the new helloSvc client instance #D to make calls to the server using the service interface #F. When the client's session is complete the connection can be closed with the transport close() method #G.

Ruby Features

Ruby is a fairly high profile language and is one of the more frequently used Apache Thrift languages. The Apache Thrift Ruby libraries support most of the top shelf Apache Thrift features.

When using Apache Thrift IDL the type mappings from Ruby to Apache Thrift are intuitive:

Apache Thrift Type	**Ruby Type**
bool	True or False
byte, i16, i32, i64	Integer (e.g. 8, 0, -394)
double	Float (e.g. 3.1415, -42.42)
binary	String (e.g. "\xE5\xA5\xBD")
string	String (e.g. 'hi Mom')
list	Array (e.g. [1,2,3,4,5])
map	Hash (e.g. {'red' => 'FF0000', 'blue' => '0000FF' })
set	Set (e.g. Set.new([1,2,3,4,5]))
struct	Object (e.g. myType.new({'name'=>'Bob, 'age'=>24}))

union	Object (e.g. myUnion.new({'color'=>'red'}))

Ruby also supports IDL namespaces. In Ruby an IDL namespace generates a module. For example placing the statement "namespace rb FishCo" at the top of an IDL file would place all of the generated code in the Ruby "module FishCo". Ruby will also create modules for wildcard namespaces, for example "namespace * FishCo".

Ruby supports most of the common Apache Thrift End Point Transports

- TCP Sockets Thrift::Socket
- HTTP Thrift::HTTPClientTransport
- Unix Domain Sockets Thrift::UNIXSocket
- Memory Buffers Thrift::MemoryBufferTransport
- Arbitrary Streams Thrift::IOStreamTransport

Ruby I/O stacks usually layer either the buffered transport or the framed transport over the endpoints. The buffered transport (Thrift::BufferedTransport) should be used by default for performance to avoid transmitting incomplete messages. Some servers (typically nonblocking servers) require a frame header, in which case the framed transport should be used (Thrift::FramedTransport). The framed transport provides its own buffering. Ruby also supply all three common serialization protocols

- Binary Thrift::BinaryProtocol
  Thrift::BinaryProtocolAccelerated
- Compact Thrift::CompactProtocol
- JSON Thrift::JsonProtocol

A native implementation of the binary protocol can be accessed through BinaryProtocolAccelerated and BinaryProtocolAcceleratedFactory classes. The native extension is a compiled C language drop in replacement for the Ruby based BinaryProtocol. In the client program, you could replace the BinaryProtocol type with the BinaryProtocolAccelerated type to use the faster native extension. On the server you would replace the BinaryProtocolFactory with the BinaryProtocolAcceleratedFactory. The speed improvement varies by platform and application.

Here are some time trials to give you a sense of the general magnitude of performance improvement represented by BinaryProtocolAccelerated. When running the helloSvc::getMessage() method 100,000 times in a tight loop against a local server using the Ruby implementation of BinaryProtocol, the total run time consumed was about 14 seconds on my test system. Changing the server to the accelerated native extension reduced this to

around 12.5 seconds and changing the client and server to native produced a run time of about 11 seconds. So while the performance boost is modest, if you have the native extension lying around, it is probably worth using. The native extension may not build successfully on all target systems, which might be a good reason not to use it. Whether you choose the Ruby or the C implementation of BinaryProtocol the bits on the wire are the same and the client or server on the other end of the connection will not be impacted by your choice.

Ruby also supplies a useful set of prebuilt servers you can quickly deploy.

- Single Threaded Server — Thrift::SimpleServer
- Multi-Threaded Server — Thrift::ThreadedServer
- Thread Pool Server — Thrift::ThreadPoolServer
- Nonblocking server — Thrift::NonblockingServer
- HTTP server — Thrift::ThinHTTPServer
  Thrift::MongrelHTTPServer

The single threaded simple server handles one client at a time. It is simple and fast but only useful in special cases. The multi-threaded server creates a new thread for each client, allowing multiple clients to connect in parallel. The standard Ruby interpreter does not allow more than one thread to run at a time. This means that I/O and system operations can take place in parallel but the Ruby code associated with the server cannot.

The thread pool server creates a pool of threads (configurable through the last parameter of the server's initialize method, 20 by default) and assigns each thread to a connecting client. When all threads are in use, connections queue until a thread is freed by a closing connection.

The nonblocking server is the most complex of the servers. It uses a pool of worker threads to process inbound user requests from an arbitrary number of clients. The thread pool can be set in the server's initialize method and defaults to 20.

The Ruby library offers two HTTP server implementations. The first is based on the Thin Ruby gem. Thin is a Ruby web server that combines the Mongrel parser, the Event Machine network I/O library and the Rack webserver interface. The second implementation is based on Mongrel, another popular Ruby web server.

For more details regarding servers and threading models, see the Servers Chapter in my book, The Programmer's Guide to Apache Thrift.