



THE PROGRAMMER'S GUIDE TO

Apache Thrift

Randy Abernethy

 MANNING



**MEAP Edition
Manning Early Access Program
The Programmer's Guide to Apache Thrift
Version**

Copyright 2014 Manning Publications

For more information on this and other Manning titles go to
www.manning.com

brief contents

PART 1: APACHE THRIFT OVERVIEW

1 Introduction to Apache Thrift

2 Apache Thrift Architecture

PART 2: PROGRAMMING APACHE THRIFT

3 Moving Bytes with Transports

4 Handling Exceptions

5 Serializing Data with Protocols

6 Apache Thrift IDL

7 User Defined Types

8 Implementing Services

9 Servers

PART 3: POLYGLOT APPLICATION DEVELOPMENT

10 A Thrift based Enterprise

11 Apache Thrift and the Web

12 Building Node.js Clients and Servers

13 Building C# Clients and Servers with .Net and Windows

14 Advanced C++ Clients and Servers

15 Advanced Java Clients and Servers

16 Python, Perl, PHP & Ruby

17 Apache Thrift Mobile Clients

18 The Big Picture

Glossary

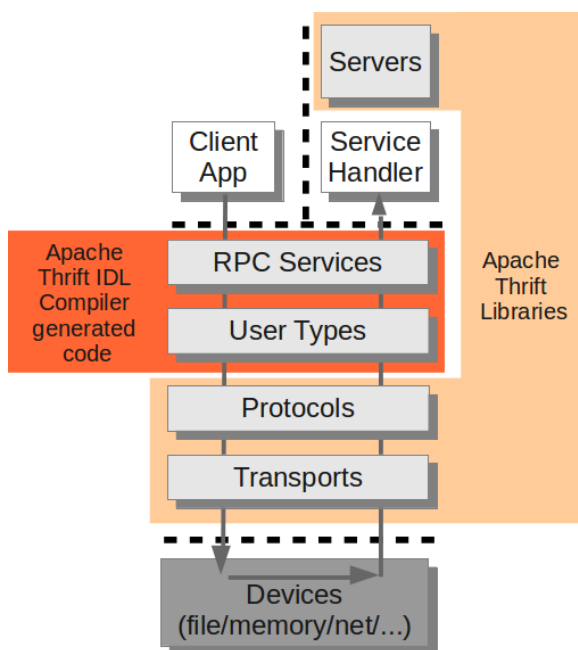
Index

Bibliography

Part 1

Apache Thrift Overview

Apache Thrift is an open source cross language serialization and RPC framework. With support for over 15 programming languages, Apache Thrift can play an important role in a range of distributed application development environments. As a serialization platform Apache Thrift enables efficient cross language storage and retrieval of a wide range of data structures. As an RPC framework, Apache Thrift enables rapid development of complete polyglot services in a few lines of code. Part 1 of this book takes you on a guided tour through the range of distributed development solutions empowered by Apache Thrift. You'll see how the Apache Thrift framework fits into various communications schemes and also get a high level picture of the overall Apache Thrift architecture.



Introduction to Apache Thrift

This chapter covers

- Using Apache Thrift to unify polyglot systems
- How Apache Thrift simplifies the creation of networked services
- An introduction to the Apache Thrift modular serialization system
- How to create a simple Apache Thrift multilanguage application

This chapter introduces the Apache Thrift framework. We take a look at why Apache Thrift was created and how it helps programmers build high performance cross language services. We begin with a look at the growing need for multilanguage integration and examine the role Apache Thrift plays in distributed application development. The chapter also includes a tutorial walk through of a simple Apache Thrift application, demonstrating how easily cross language networked services can be created with Apache Thrift.

1.1 Polyglotism, the pleasure and the pain

The number of programming languages in common commercial use has grown considerably in recent years. In 2003 80% of the Tiobe Index was attributed to six programming languages: Java, C, C++, Perl, Visual Basic and PHP. In 2013 it took twice as many languages to capture the same 80%, adding Objective-C, C#, Python, JavaScript and Ruby to the list. Increasingly developers and architects choose the programming language most suitable for the task at hand. A developer working on a Big Data project might decide Clojure is the best language to use, meanwhile folks down the hall may be doing front end work in JavaScript, while programmers upstairs might be using C++ with embedded systems. Years ago this type of diversity would be rare at a single company, now it can be found within a single team.

Choosing a programming language uniquely suited to solving a particular problem can lead to productivity gains and better quality software. When the language fits the problem, friction is reduced, programming becomes more direct and code becomes	2003 Tiobe Index top four quintiles			2013 Tiobe Index top four quintiles		
	Language	Rating	Cumulative	Language	Rating	Cumulative
	Java	23.08%	23.08%	C	17.81%	17.81%
	C	18.47%	41.55%	Java	16.66%	34.47%
	C++	15.56%	57.12%	Objective-C	10.36%	44.82%
	Perl	9.42%	66.54%	C++	8.82%	53.64%
	(Visual) Basic	7.81%	74.35%	PHP	5.99%	59.63%
	PHP	4.76%	79.11%	C#	5.78%	65.41%
	During the ten years from 2003 to 2013 the number of languages comprising the top 80% of the Tiobe Index doubled. With the exception of PHP, every language in the 2003 index made up a smaller percentage of the index in 2013.			(Visual) Basic	4.35%	69.76%
				Python	4.18%	73.94%
				Perl	2.27%	76.21%
				JavaScript	1.65%	77.87%
				Ruby	1.48%	79.35%

Figure 1.1 - The Tiobe Index uses web search results to track programming language popularity (www.tiobe.com)

simpler and easier to maintain. For example, in large scale data analysis, horizontal scaling is instrumental in achieving performance. Functional programming languages like Haskell, Scala and Clojure tend to fit naturally here, allowing analytic systems to scale without complex concurrency concerns.

New platforms drive language adoption as well. Objective-C exploded in popularity when Apple released the iPhone, most programming for Android will be biased toward Java and the Windows Phone folks will likely be using C#. Those coding for the browser will have teams competent with JavaScript. Embedded systems shops are going to have strong C programming groups and high performance GUI applications will often be written in C++. These choices are driven by history as well as compelling technology underpinnings. Even when such groups are internally monoglots, languages mix and mingle as they collaborate across business boundaries.

Many organizations who claim monoglotism make use of a range of support languages for testing and prototyping. Dynamic programming languages such as Groovy and Ruby are often used for test and behavioral driven development solutions, while Perl and Python are popular for prototyping and PHP has a long history on the server side of the web in IT support roles. Platforms such as the Groovy based Gradle and the Ruby based Rake provide innovative build capabilities. Thus, even firms that think they are monoglots may not be, given the proliferation of innovative language driven tools around the periphery or core application development.

The Polyglot story is not all wine and song, however. Mastering a programming language is no small feat, not to mention the tools and libraries that come with it. As this burden is multiplied with each new language, firms may experience diminishing returns. Introducing multiple languages into a product initiative can have numerous costs associated with cross language integration, developer training, and complexity in build and test. If managed improperly, these costs can quickly overshadow the benefits of a multilanguage strategy.



Figure 1.2 - The growing number of programming languages in commercial use creates important cross language interoperability challenges

There are many degrees of Polyglotism. Some go to the extreme, coding individual components of a single process in different languages, relying on a runtime layer, such as the JVM or CLR, to provide interoperability. Others are purists, sticking with a single language for everything and only dealing with other languages when forced to by partners and clients. Either way, if trends continue, only the rarest of software teams will be truly isolated within the bounds of a single programming language in the years ahead. While Polyglotism

may be bane or boon, depending on your point of view, it is a condition not likely to go away. The more our programs mirror the dialog on the floor of the United Nations General Assembly, the more we will need professional translators to communicate across languages.

1.2 The Apache Thrift Framework from 30,000 feet

Apache Thrift solves many of the problems associated with building applications which collaborate across language boundaries. In addition to normalizing data for cross language communications, Apache Thrift also provides a complete remoting framework making it trivial to build cross language networked services.

The challenges of polyglot development come in many flavors. Some developers need a way to allow two processes written in different languages to communicate, for example a web tier application written in PHP communicating with an enterprise service written in C++. Others may have a program or service which they would like to expose to an unknown range of clients, for example a cloud based database written in Java with a public API which anyone using any language should be able to access. Commercial systems such as EverNote and open source projects such as Cassandra have adopted Apache Thrift as their principle API provider.

The Apache Thrift framework is also a powerful tool for distributed application development, even when only one programming language is involved. For example, Apache Thrift can facilitate scenarios where two C++ applications need to communicate across machine boundaries, resolving 32 bit and 64 bit differences during remote procedure calls. Whether the problem involves one, two, or ten languages, Apache Thrift can supply the cross language glue needed to enable collaboration.

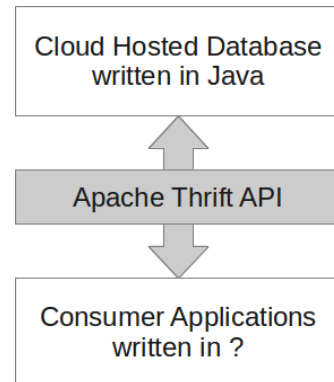


Figure 1.3 - Apache Thrift can be used to export a single service API to a wide range of client languages

1.3 Reasons to consider Apache Thrift

There are several key benefits associated with using Apache Thrift to develop network services or perform cross language serialization tasks.

- Full RPC Implementation - Apache Thrift supplies a complete RPC solution
- Modularity - Apache Thrift supports plug-in serialization protocols
- Performance - Apache Thrift is fast and efficient
- Reach - Apache Thrift supports a wide range of languages and platforms
- Flexibility - Apache Thrift supports interface evolution

1.3.1 RPC Services

Remote Procedure Call (RPC) servers are modular application components which provide services accessible over a network. Apache Thrift provides a complete cross language RPC solution. The RPC service facilities of Apache Thrift are Interface Definition Language (IDL) based (see Listing 1.1). Services are described using a simple IDL syntax and compiled to generate stub code used to connect clients and servers in a wide range of languages.

For example, imagine you have a C++ module which tracks and computes sailboat team statistics for the

America's Cup and that this module is built into a C++ GUI application. However, your company's web site team would also like to use the sail stats module. The problem might be that the web site team builds everything in Ruby, PHP and Python and their programs are based around the Service Oriented Architecture (SOA) model, which would require the sail stats to be deployed as a network service of some sort.

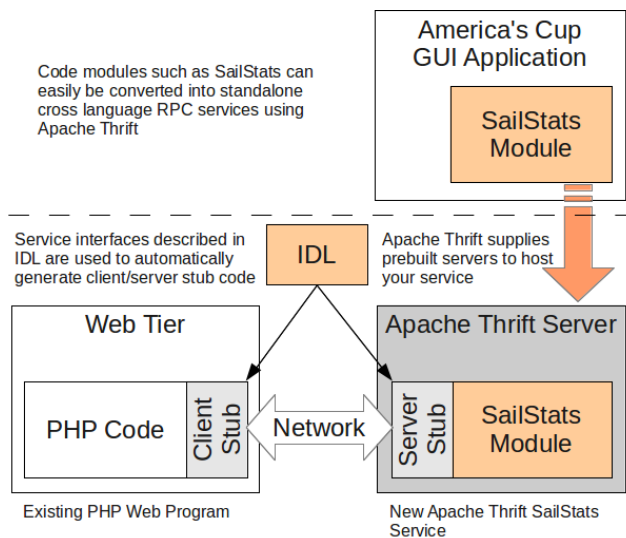


Figure 1.4 - Converting a module from an old application (above dotted line) into a distributed network application (below dotted line)

Service Oriented Architecture (SOA)

The SOA approach breaks distributed applications down into services, which are remotely accessible autonomous modules composed of a set of closely related functions. SOA based systems generally provide their features over language agnostic interfaces, allowing clients to be constructed in the most appropriate language and on the most appropriate platform, independent of the service implementation. SOA services are typically stateless and loosely coupled, communicating with clients through a formal interface contract. SOA services may be internal to an organization or support clients across business boundaries.

Encapsulating the SailStats module in a SOA service will make it easy for any part of the company's enterprise to access the service. There are several common ways to build SOA services using web oriented technologies, however many of these would require the installation of web or application servers, possibly a material amount of additional coding, and/or the use of HTTP communications schemes like REST, which are broadly supported but not famous for being fast or compact.

A better approach in this case may be to use Apache Thrift. Using Apache Thrift IDL we can define a service interface with the functions we want to expose. We can then use the Apache Thrift compiler to generate RPC code for our SailStats service in Ruby, PHP, Python and C++. The web team can now use code generated in their language of choice to call the functions offered by the SailStats service, exactly as if the functions were defined locally (see Figure 4).

Apache Thrift supplies a complete library of RPC servers. This means that you can use one of the powerful multithreaded servers provided by Apache Thrift to handle all of the server RPC processing and concurrency matters. The C++ code generated from our IDL will make it easy for us to take our existing module and wire it up using one of the Apache Thrift RPC servers.

Listing 1.1 ~/thriftbook/hello/SailStats.thrift

```
service SailStats {
    double GetSailorRating(1: string SailorName),
    double GetTeamRating(1: string TeamName),
    double GetBoatRating(1: i64 BoatSerialNumber),
    list<string> GetSailorsOnTeam(1: string TeamName),
    list<string> GetSailorsRatedBetween(1: double MinRating,
                                     2: double MaxRating),
    string GetTeamCaptain(1: string TeamName),
}
```

In summary, to turn a code library or module into a high performance RPC service with Apache Thrift, all we need to do is:

1. Define the service interface in IDL
2. Compile the IDL to generate client and server RPC stub code in the desired languages
3. On the client side call the remote functions as if they were local using the client stubs

4. On the Server side connect the server stubs to the desired module functionality
5. Choose one of the prebuilt Apache Thrift servers to host the service

In exchange for a fairly small amount of work, we can turn almost any set of existing functions into a high performance Apache Thrift service, accessible from a broad range of client languages. We will build a demonstration RPC service later in the chapter using Java, C++ and Python.

1.3.2 Modular Serialization

To make a function call from a client to a server, both client and server must agree on the representation of data exchanged. The typical approach to solving this problem is to select an interchange format and then to transform all data to be exchanged into this interchange format. The process of transforming data to and from an interchange format is called serialization. In essence, taking a complex memory object and turning it into a serial bit stream.

The Apache Thrift framework provides a complete, modular, cross language serialization layer which supports RPC functionality but can also be used independently.

Serialization frameworks make it easy to store data to disk for later retrieval by another application. For example, a service written in C which captures live earthquake data in a C struct could serialize this data to disk using Apache Thrift (see figure 1.5). The serialization process converts the C struct into a generic Apache Thrift serialized object. At a later time a Ruby earthquake analysis application could use Apache Thrift to restore the serialized object. The serialization layer takes care of the various differences in data representation between the languages automatically.

A fairly unique feature of the Apache Thrift serialization framework is that it is not hard wired to a single serialization protocol. The serialization layer provided by Apache Thrift is modular, making it possible to choose from an assortment of serialization protocols, or even to create custom serialization protocols. Out of the box, Apache Thrift supports an efficient binary serialization protocol, a compact protocol which reduces the size of serialized objects and a JSON protocol which provides broad interoperability with JavaScript and the web.

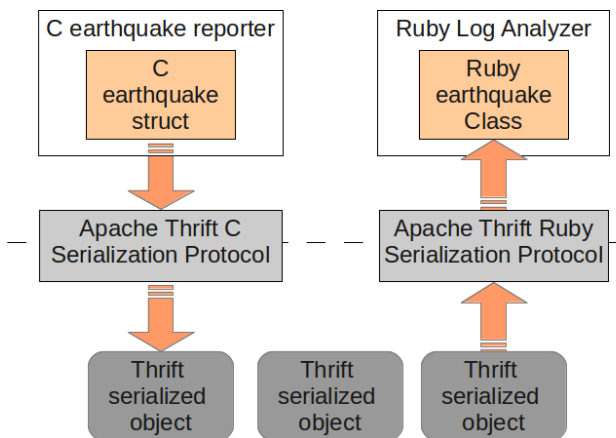


Figure 1.5 - Apache Thrift serialization protocols enable different programming languages to share abstract data types

1.3.3 Performance

Apache Thrift is a good fit in many distributed computing settings, however it excels in the area of high performance backend service development. The Apache Thrift framework offers a choice of prebuilt and custom protocols for serialization. This allows the



Figure 1.6 - Thrift and the RPC service landscape

application designer to choose the most appropriate serialization protocol for the needs of the application, balancing transmission size and speed.

Apache Thrift supports compiled languages such as C, C++, Java and C#, which generally have a performance edge over interpreted languages. This allows performance critical services to be built in the appropriate language while still providing support for scripting clients.

Apache Thrift RPC servers are lightweight, performing only the task of hosting Apache Thrift services. A selection of servers is available in various languages giving application designers the flexibility to choose a concurrency model well suited to their application requirements.

Apache Thrift covers a wide performance range in between the two extremes represented by custom communication development on one end and REST on the other (see figure 1.6). The light weight nature of Apache Thrift combined with a choice of efficient serialization protocols allows Apache Thrift to meet a range of performance requirements while offering support for an impressive breadth of languages and platforms.

1.3.4 Reach

The Apache Thrift framework supports a number of programming languages, operating systems and hardware platforms in both serialization and RPC capacities. By adopting Apache Thrift as a means to provide interfaces internally and externally, organizations gain substantial optionality. In a

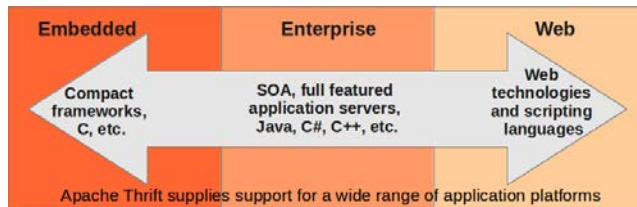


Figure 1.7 - Apache Thrift supports embedded, enterprise and web technology platforms

company that is growing and changing rapidly, giving teams the flexibility to integrate with new languages and platforms effortlessly can be a significant business advantage.

The table below provides a list of the languages supported directly by Apache Thrift 1.0. Note that support for C# enables other .Net/CLR languages, such as F#, VisualBasic and IronPython, to easily integrate with Apache Thrift services. By the same token, support for Java enables a host of JVM based languages, such as Scala, Clojure and Groovy, to interoperate with

Apache Thrift. JavaScript support is provided for browser based applications and Node.js. Other projects found on the web expand this list further.

Table 1.1 - Languages supported by Apache Thrift

C	C++	C#	D
Delphi	Erlang	Go	Haskell
Java	JavaScript	Objective-C	OCaml
Perl	PHP	Python	Ruby
Smalltalk			

Apache Thrift also supports a range of platforms including Windows, iOS, OS X, Linux, Android and many other Unix-like systems. Apache Thrift is compact and supports C/C++ and JavaME, making it appropriate for some embedded systems. At the other end of the spectrum, Apache Thrift supports Perl, PHP, Python, Ruby and JavaScript, making it viable in web oriented environments. Few frameworks can supply the breadth of reach in languages and platforms offered by Apache Thrift.

1.3.5 Interface Evolution

Interface evolution is the process of changing the elements of an interface gradually over time without breaking interoperability with modules built around older versions of the interface. For example, consider the previously described earthquake application where a C language program writes a C language struct to disk each time a tremor is reported. Let's assume that the earthquake struct contains fields for the date, time, position and magnitude of the earthquake and that this struct is shared with a Ruby data analysis program. The interface evolution features of Apache Thrift allow new fields, say the earthquake's nearest city and state, to be added to the earthquake struct without breaking the Ruby application. The Ruby program will continue to read old and new earthquake files, simply ignoring fields it does not recognize. Should the Ruby programmers require the new fields they may add support for them at their leisure.

Early RPC systems like SunRPC, DCE, CORBA and MSRPC supplied little or no support for interface evolution. As platforms grow and requirements change rigid interfaces can make it hard to extend and maintain RPC based services. Modern RPC systems such as Apache Thrift provide a number of features which allow interfaces to evolve over time without breaking compatibility with existing systems. Functions can be extended with new parameters, old parameters can be removed, and default values can be supplied. Properly applied these changes can be made without impacting peers using older versions of the interface.

Continuous Integration (CI) & Continuous Delivery (CD)

Continuous integration is an approach to software development wherein changes to a system are merged into the central code base frequently. These changes are continuously built and tested, usually by automated systems, providing developers with rapid feedback when patches create conflicts or fail tests. Taking CI to its logical conclusion involves migrating successfully merged code to evaluation and ultimately production systems at a high frequency. Often these processes occur multiple times per day. The goal of continuous systems is to take many small risks and to provide immediate feedback rather than taking large risks and delaying feedback over long release cycles. The longer integration is delayed the more patches are involved in the integration task, making it more difficult to identify and repair the source of conflicts and bugs.

Support for interface evolution greatly simplifies the task of ongoing software maintenance and extension, particularly in a large enterprise. Modern engineering sensibilities such as Continuous Integration (CI) and Continuous Delivery (CD) require systems to support incremental improvements. Systems which do not supply some form of interface evolution tend to “break the world” when changed. That is to say that changing the interface usually means that all of the clients and servers using that interface must be rewritten and/or recompiled. Apache Thrift interface evolution features allow multiple interface versions to coexist, making incremental updates viable. We will take an in depth look at interface evolution in later chapters.

1.4 Building a Simple RPC Example

Now that we have covered some of the key features and benefits of Apache Thrift, let’s take a look at a very simple Apache Thrift RPC example. If you already have a working Thrift installation on your system you can build the example as you read. If you do not have a working Thrift installation you may want to take a few minutes to setup Apache Thrift (see the Appendices at the end of this book).

For this example we’ll build a simple server designed to supply various parts of our enterprise with a daily greeting. Our service will expose one function which takes no parameters and returns our greeting string. To see how Apache Thrift works across languages we will build clients in C++, Python and Java.

1.4.1 Describing services with Apache Thrift IDL

Most projects involving Apache Thrift begin with careful consideration of the services that will be consumed. Service interfaces are the basis for communications between clients and servers in Apache Thrift. Apache Thrift services are defined using an Interface Definition Language (IDL) similar to C in its notation. IDL code is saved in plain text files with a “.thrift” extension.

Here is the IDL file we will use for our service.

Listing 1.2 ~/thriftbook/hello/hello.thrift

```
service HelloSvc {           #A
    string hello_func()      #B
}
```

This IDL file declares a single service interface called HelloSvc #A. HelloSvc has one function, hello_func() which accepts no parameters and returns a string #B. To use this interface in an RPC application we can compile it with the Apache Thrift IDL Compiler. The IDL Compiler will generate stub code for both clients using the interface and servers implementing the interface. In this example we will begin by using the compiler to generate Python stubs for the HelloSvc. The IDL Compiler binary is called “thrift” on UNIX like systems and “thrift.exe” on Windows. The IDL Compiler accepts an IDL file and a target language to generate code for. Here’s an example:

```
~/thriftbook/hello $ ls -l
-rw-r--r-- 1 randy randy 95 Mar 26 16:28 hello.thrift
~/thriftbook/hello $ thrift -gen py hello.thrift           #A
~/thriftbook/hello $ ls -l
drwxr-xr-x 3 randy randy 4096 Mar 26 16:31 gen-py          #B
-rw-r--r-- 1 randy randy 95 Mar 26 16:28 hello.thrift
~/thriftbook/hello $ ls -l gen-py
drwxr-xr-x 2 randy randy 4096 Mar 26 16:31 hello           #C
-rw-r--r-- 1 randy randy 0 Mar 26 16:31 __init__.py
~/thriftbook/hello $ ls -l gen-py/hello
-rw-r--r-- 1 randy randy 248 Mar 26 16:31 constants.py     #D
-rw-r--r-- 1 randy randy 5707 Mar 26 16:31 HelloSvc.py     #E
-rwxr-xr-x 1 randy randy 1896 Mar 26 16:31 HelloSvc-remote #F
-rw-r--r-- 1 randy randy 46 Mar 26 16:31 __init__.py       #G
-rw-r--r-- 1 randy randy 398 Mar 26 16:31 ttypes.py        #H
~/thriftbook/hello $
```

In this example we specify Python as our target by providing the gen switch with the “py” argument #A. The IDL Compiler creates a gen-py directory to house all of the emitted Python code #B. Within the gen-py directory the IDL Compiler creates a package with the same name as our IDL file #C. Inside the hello package we find support files for each service defined in our IDL #E, a file for any types we may have defined #H and a file for any constants we may have defined #D. The HelloSvc-remote file is a sample executable Python test client for our HelloSvc service #F. Python packages also typically include an __init__.py file with package configuration instructions, which the IDL Compiler also generates #G.

1.4.2 Building a Python Server

Since we already have a test client built for us, let’s construct an Apache Thrift Python RPC server as our first project. We will need to implement the HelloSvc service and define a server to make our service available on the network. In this example, for simplicity, we create all of our Python source files in the same directory as the hello.thrift IDL file.

NOTE Apache Thrift does not currently support Python 3.x. To build the Python examples in this book you will need Python 2.5 – 2.7.x.

Here's the code for our starter Python server.

Listing 1.3 ~/thriftbook/hello/hello_server.py

```
import sys                                #A
sys.path.append("gen-py")                #A
from hello import HelloSvc                #A

from thrift.transport import TSocket       #B
from thrift.server import TServer         #B

class HelloHandler:                       #C
    def hello_func(self):
        print("[Server] Handling client request")
        return "Hello from the python server"

handler = HelloHandler()                  #D
processor = HelloSvc.Processor(handler)    #D

listening_socket = TSocket.TServerSocket(port=8585) #E
server = TServer.TSimpleServer(processor, listening_socket) #E
server.serve()                            #E
```

Our listing begins with the resolution of several dependencies using Python's import statement. We use the built in Python sys module to add the gen-py directory to the Python Path. This allows us to import the IDL Compiler generated RPC stubs for our HelloSvc service #A.

Our next step is to import an end point network transport for our clients to connect to and a server to run our service from the Apache Thrift library. The Apache Thrift library is broken up into packages/directories. Here we import a server module from the thrift.server package and the socket module from the thrift.transport package. This will allow us to create a basic RPC server using TCP/IP sockets for communications #B.

The next block of code implements the HelloSvc service. This class is called a handler in Apache Thrift parlance. All of the Service methods must be represented in the Handler class, in our case this is just the hello_func() method #C.

Next we create an instance of our handler and use it to initialize a processor for our service. The processor is the server side stub generated by the IDL Compiler which turns network RPC messages into calls to the handler #D.

Our RPC server will need a service processor and an end point transport for clients to connect to. The Apache Thrift library offers end points for use with files, memory buffers and networks. The example here creates a TCP server socket end point to accept client connections on TCP port 8585. Apache Thrift also provides a range of servers to choose from, each with unique features. The server used here is an instance of the TSimpleServer class, which, as its

name implies, provides basic RPC server functionality. Once constructed, we can run the server by calling the `serve()` method #E.

The Apache Thrift Tutorial

In addition to the code examples included with this text, the Apache Thrift source tree provides a tutorial with samples in each supported language. The tutorial is based on a central IDL file defining a calculator service from which client and server samples in each language are built. This tutorial is simple but demonstrates many of the capabilities of Apache Thrift in every supported language. The tutorials can be found under the `thrift/tutorial` directory of the source tree. The Apache Thrift IDL files used by all of the language tutorial examples are located in the root of the tutorial tree with `.thrift` extensions. Each language specific tutorial is found in a subdirectory named for the language. A Makefile is provided to build the tutorial examples in languages which require compilation.

```
~thrift-1.0.0/thrift/tutorial $ ls
```

```
cpp      go      Makefile.am  php      README
csharp   hs      Makefile.in  py      shared.thrift
d        java    nodejs      py.tornado  tutorial.thrift
delphi   js      ocaml      py.twisted
erl      Makefile  perl      rb
```

To test our server program we can run the code at the command line using the Python interpreter.

```
~/thriftbook/hello $ ls -l
drwxr-xr-x 4 randy randy 4096 Jan 27 02:34 gen-py
-rw-r--r-- 1 randy randy 732 Jan 27 03:44 hello_server.py
-rw-r--r-- 1 randy randy 99 Jan 27 02:24 hello.thrift
~/thriftbook/hello $ python hello_server.py
```

Our server is now up and running and waiting to serve client requests. Excluding import statements and the handler implementation, creating our RPC server took 5 lines of Python code. The story is similar in C++, Java and a number of other languages. This is a very basic server but the example should give you some sense as to how much leverage Apache Thrift gives you when it comes to quickly creating cross language RPC services.

1.4.3 Building a Python Client

Next let's code up a Python client in the same directory as our hello server to take a look at the client side of Apache Thrift RPC. Our client will simply call the `hello_func()` service method, display the result and exit.

Listing 1.4 ~/thriftbook/hello/hello_client.py

```

import sys
sys.path.append("gen-py")

from thrift.transport import TSocket           #A
from thrift.protocol import TBinaryProtocol    #B
from hello import HelloSvc                    #C

socket = TSocket.TSocket("localhost", 8585)    #D
socket.open()                                #D
protocol = TBinaryProtocol.TBinaryProtocol(socket) #E

client = HelloSvc.Client(protocol)             #F
msg = client.hello_func()                      #F
print("[Client] received: %s" % msg)

```

Our client requires two modules from the Apache Thrift Python Library. The first is TSocket which is used on the client side to make a TCP connection to a server socket #A. Our next import pulls in the TBinaryProtocol which we will use to serialize our messages to the server #B. Finally we import the IDL Compiler generated client side stubs for our HelloSvc #C.

We initialize our TSocket with the host and port to connect to and call the socket's open() method to make the connection with the server #D. To transmit RPC messages over the socket transport we will need to serialize them using a protocol. While there are a several protocols to choose from, the binary protocol is the Apache Thrift default. To construct our binary protocol instance we pass it the underlying transport that it will use to deliver serialized RPC messages #E.

The final layer of our I/O stack is the IDL Compiler generated client stub for our HelloSvc. The IDL Compiler creates a Client class for each service which acts as a proxy for the remote service. After constructing the client object and supplying it with a protocol object we can make calls to the service using the client. Invoking the hello_func() method on the Client object serializes our call request with the binary protocol and transmits it over the socket to the server #F.

Here is a sample session running the above client. The Python server must be running in another shell to respond.

```

~/thriftbook/hello $ ls -l
drwxr-xr-x 3 randy randy 4096 Mar 26 21:45 gen-py
-rw-r--r-- 1 randy randy 386 Mar 26 21:59 hello_client.py
-rw-r--r-- 1 randy randy 535 Mar 26 16:50 hello_server.py
-rw-r--r-- 1 randy randy 95 Mar 26 16:28 hello.thrift
~/thriftbook/hello $ python hello_client.py
[Client] received: Hello from the python server
~/thriftbook/hello $

```

While a bit more work than your run of the mill hello world program, a few lines of IDL and Python have allowed us to create a language agnostic, OS agnostic and platform agnostic service API with a working client and server. Not bad.

1.4.4 Building a C++ Client

To broaden our perspective and demonstrate the cross language aspects of Apache Thrift we will build two more clients for the hello server, one in C++ and one in Java. Let's start with the C++ client.

As a first step we will compile our existing IDL for C++.

```
~/thriftbook/hello $ thrift -gen cpp hello.thrift #A
~/thriftbook/hello $ ls -l
drwxr-xr-x 2 randy randy 4096 Mar 26 22:25 gen-cpp
drwxr-xr-x 3 randy randy 4096 Mar 26 21:45 gen-py
-rw-r--r-- 1 randy randy 386 Mar 26 21:59 hello_client.py
-rw-r--r-- 1 randy randy 535 Mar 26 16:50 hello_server.py
-rw-r--r-- 1 randy randy 95 Mar 26 16:28 hello.thrift
~/thriftbook/hello $ ls -l gen-cpp
-rw-r--r-- 1 randy randy 255 Mar 26 22:25 hello_constants.cpp #B
-rw-r--r-- 1 randy randy 337 Mar 26 22:25 hello_constants.h #B
-rw-r--r-- 1 randy randy 8187 Mar 26 22:25 HelloSvc.cpp #B
-rw-r--r-- 1 randy randy 6219 Mar 26 22:25 HelloSvc.h #B
-rw-r--r-- 1 randy randy 1311 Mar 26 22:25 HelloSvc_server.skeleton.cpp #B
-rw-r--r-- 1 randy randy 193 Mar 26 22:25 hello_types.cpp #B
-rw-r--r-- 1 randy randy 350 Mar 26 22:25 hello_types.h #B
~/thriftbook/hello $
```

Running the IDL Compiler with the “-gen cpp” switch causes it to emit C++ files roughly equivalent to those generated for Python #A. We now have headers (.h) and source files (.cpp) for our hello.thrift constants and types, as well as HelloSvc RPC service stubs #B. We did not define any constants or user defined types in our IDL so the “*constants.*” and “*types.*” files will contain only boiler plate code. The HelloSvc.h header contains the declarations for our service and the HelloSvc.cpp source file contains the implementation of the RPC stub components. Note that the C++ generator created a server skeleton for us but no sample client. The Python generator did just the opposite. Apache Thrift language libraries each have their own subtleties.

Here's the code for a HelloSvc C++ client with the same functionality as the Python client above.

Listing 1.5 ~/thriftbook/hello/hello_client.cpp

```
#include <iostream>
#include <string>
#include <boost/shared_ptr.hpp> #A
#include <thrift/transport/TSocket.h> #B
#include <thrift/protocol/TBinaryProtocol.h> #B
#include "gen-cpp/HelloSvc.h" #C

using namespace apache::thrift::transport; #D
using namespace apache::thrift::protocol; #D

int main() { #E
    boost::shared_ptr<TSocket> socket(new TSocket("localhost", 8585));
    socket->open();
```

```

boost::shared_ptr<TBinaryProtocol> protocol(new
                                           TBinaryProtocol(socket));

HelloSvcClient client(protocol);
std::string msg;
client.hello_func(msg);
std::cout << "[Client] received: " << msg << std::endl;
}

```

Our C++ client code is structurally identical to the Python client code. With few exceptions, the Apache Thrift meta-model is consistent from language implementation to language implementation, making it easier for developers to work across languages.

The C++ main() function corresponds line for line with the Python code with one exception, hello_func() does not return a string conventionally, rather it returns the string through an out parameter reference #E. The Apache Thrift language libraries are generally wrapped in namespaces to avoid conflicts in the global namespace. Each language has its own unique namespace mechanisms. In C++ all of the Apache Thrift library code is located within the lengthy but intuitive apache.thrift namespace. The using statements listed here provide the namespaces for TSocket and TBinaryProtocol #D.

Our C++ program uses #include to resolve compile time dependencies. The Apache Thrift library headers included provide us support for sockets transports and the binary protocol #B. We also include our generated service stubs which are housed in the HelloSvc.h file #C.

While Apache Thrift strives to maintain as few dependencies as possible to keep the development environment simple and portable, some things are indispensable. The Apache Thrift C++ Library relies fairly heavily on the open source Boost Library #B. In our example above both the socket transport and the binary protocol objects are wrapped in boost::shared_ptr. Apache Thrift uses shared_ptr to manage almost all of the key objects involved in C++ RPC.

Here is a session which builds and runs our C++ client.

```

~/thriftbook/hello $ ls -l
drwxr-xr-x 2 randy randy 4096 Mar 26 22:25 gen-cpp
drwxr-xr-x 3 randy randy 4096 Mar 26 21:45 gen-py
-rw-r--r-- 1 randy randy 641 Mar 26 22:36 hello_client.cpp
-rw-r--r-- 1 randy randy 386 Mar 26 21:59 hello_client.py
-rw-r--r-- 1 randy randy 535 Mar 26 16:50 hello_server.py
-rw-r--r-- 1 randy randy 95 Mar 26 16:28 hello.thrift
~/thriftbook/hello $ g++ hello_client.cpp gen-cpp/HelloSvc.cpp -lthrift #A
~/thriftbook/hello $ ls -l
-rwxr-xr-x 1 randy randy 136508 Mar 26 22:38 a.out
drwxr-xr-x 2 randy randy 4096 Mar 26 22:25 gen-cpp
drwxr-xr-x 3 randy randy 4096 Mar 26 21:45 gen-py
-rw-r--r-- 1 randy randy 641 Mar 26 22:36 hello_client.cpp
-rw-r--r-- 1 randy randy 386 Mar 26 21:59 hello_client.py
-rw-r--r-- 1 randy randy 535 Mar 26 16:50 hello_server.py
-rw-r--r-- 1 randy randy 95 Mar 26 16:28 hello.thrift
~/thriftbook/hello $ ./a.out #B
[Client] received: Hello thrift, from the python server
~/thriftbook/hello $

```

In this example we use the standard gnu C++ compiler to build our `hello_client.cpp` file into an executable program #A. Clang, Visual C++ and other compilers are also commonly used to build Apache Thrift C++ applications.

For the C++ compile phase we specify our `hello_client.cpp` file but must also compile the generated RPC client stub found in the `HelloSvc.cpp` source file. During the link phase the “`-lthrift`” switch tells the linker to scan the standard Apache Thrift C++ library to resolve the `TSocket` and `TBinaryProtocol` library dependencies (this switch must follow the list of `.cpp` files when using `g++` or it will be ignored causing link errors).

Assuming the Python Hello server is still up we can run our executable C++ client and make a cross language RPC call. The C++ compiler builds our source into an `a.out` file which produces the same result as the Python client when executed #B.

1.4.5 Building a Java Client

Finally let’s put together a Java client. Our first step is to generate Java stubs for our service.

```
~/thriftbook/hello $ thrift -gen java hello.thrift #A
~/thriftbook/hello $ ls -l
-rwxr-xr-x 1 randy randy 136508 Mar 26 23:07 a.out
drwxr-xr-x 2 randy randy 4096 Mar 26 22:25 gen-cpp
drwxr-xr-x 2 randy randy 4096 Mar 26 23:23 gen-java
drwxr-xr-x 3 randy randy 4096 Mar 26 21:45 gen-py
-rw-r--r-- 1 randy randy 641 Mar 26 22:36 hello_client.cpp
-rw-r--r-- 1 randy randy 386 Mar 26 21:59 hello_client.py
-rw-r--r-- 1 randy randy 535 Mar 26 16:50 hello_server.py
-rw-r--r-- 1 randy randy 95 Mar 26 16:28 hello.thrift
~/thriftbook/hello $ ls -l gen-java
-rw-r--r-- 1 randy randy 25984 Mar 26 23:23 HelloSvc.java #B
~/thriftbook/hello $
```

The “`-gen java`” switch causes the IDL Compiler to emit Java code for our interface #A. A single Java file is emitted when building our IDL for Java #B. The Java `HelloSvc` class contains nested client and server stub classes.

Here is the source for a Java client which parallels the prior Python and C++ clients.

Listing 1.6 ~/thriftbook/hello/HelloClient.java

```
import org.apache.thrift.protocol.TBinaryProtocol;
import org.apache.thrift.transport.TSocket;
import org.apache.thrift.TException;

public class HelloClient {
    public static void main(String[] args) throws TException {
        TSocket socket = new TSocket("localhost", 8585);
        socket.open();
        TBinaryProtocol protocol = new TBinaryProtocol(socket);

        HelloSvc.Client client = new HelloSvc.Client(protocol);
        String str = client.hello_func();
        System.out.println("[Client] received: " + str);
    }
}
```

```

    }
}

```

This simple program has the required Java application structure, but beyond that, the code is very close to the Python and C++ code we have already seen. We use the Java style import statements and place our main() method inside a class with the same name as the file. The rest is a rehash of our previous clients. Here is a build and run session for the Java client.

```

~/thriftbook/hello $ javac -cp /usr/local/lib/libthrift-1.0.0.jar:      #A
                        /usr/local/lib/slf4j-api-1.7.2.jar:          #A
                        /usr/local/lib/slf4j-nop-1.7.2.jar          #A
                        HelloClient.java gen-java/HelloSvc.java    #A
Note: gen-java/HelloSvc.java uses unchecked or unsafe operations.
Note: Recompile with -Xlint:unchecked for details.
~/thriftbook/hello $ ls -l
-rwxr-xr-x 1 randy randy 136508 Mar 26 23:07 a.out
drwxr-xr-x 2 randy randy  4096 Mar 26 22:25 gen-cpp
drwxr-xr-x 2 randy randy  4096 Mar 26 23:34 gen-java
drwxr-xr-x 3 randy randy  4096 Mar 26 21:45 gen-py
-rw-r--r-- 1 randy randy  1080 Mar 30 00:04 HelloClient.class
-rw-r--r-- 1 randy randy   607 Mar 29 23:48 hello_client.cpp
-rw-r--r-- 1 randy randy   657 Mar 30 00:04 HelloClient.java
-rw-r--r-- 1 randy randy   384 Mar 29 23:48 hello_client.py
-rw-r--r-- 1 randy randy   535 Mar 26 16:50 hello_server.py
-rw-r--r-- 1 randy randy    95 Mar 26 16:28 hello.thrift
~/thriftbook/hello $ java -cp /usr/local/lib/libthrift-1.0.0.jar:
                        /usr/local/lib/slf4j-api-1.7.2.jar:
                        /usr/local/lib/slf4j-nop-1.7.2.jar:
                        ./gen-java:                                #B
                        .                                           #B
                        HelloClient
[Client] received: Hello thrift, from the python server
~/thriftbook/hello $

```

As you can see in the session transcript, our Java compile includes three dependencies #A. The first is the Thrift Java Library jar. The IDL generated code for our service also depends on SLF4J, a popular Java logging façade. The slf4j-api jar is the façade and the slf4j-nop jar is the nop logger, which ignores logging requests. We must generate byte code .class files for our HelloClient class as well as the generated HelloSvc class.

The unchecked warning emitted by the javac compiler is triggered by method return type casts in the Apache Thrift generated code. The Apache Thrift Java libraries are designed to work with JDK 1.5 and newer, so while the code compiles clean in older JDKs, newer JDKs produce this warnings, which can be safely ignored in this case. The Java Setup appendix provides complete setup and configuration information for Java development if you have yet to setup Apache Thrift for Java.

To run our Java HelloClient class we must modify the Java class path as we did in the compilation step, adding the current directory and the gen-java directory, where the HelloClient class and HelloSvr classes will be found #B. Running the client produces the same result we saw with Python and C++.

Beyond running the standard language build tools, it took very little effort to produce this server and the three clients. We now have a server which can handle requests from clients created in a number of different languages with just a few lines of code.

Now that we have seen how basic Apache Thrift RPC programs are created, let's take a look at how Apache Thrift fits into the overall distributed application landscape.

1.5 *Apache Thrift's role in Distributed Applications*

Distributed applications are applications which have been broken down into subsystems that can be deployed on separate computers, yet still collaborate to accomplish the purpose of the application. When subsystems are autonomous and export flexible APIs they are often called services. Compared to large monolithic systems, distributed applications benefit from smaller more focused processes which are easier to scale, reuse, maintain and test. Distributed applications generally use one or more of three key types of inter-process communications:

- **Streaming** – Communications characterized by an ongoing flow of bytes from a server to one or more clients.
 - Example: An internet radio broadcast where the client receives bytes over time transmitted by the server in an ongoing sequence of small packets.
- **Messaging** – Message passing involves one way asynchronous, often queued, communications, producing loosely coupled systems.
 - Example: Sending an email message where you may get a response or you may not, and if you do get a response you don't know exactly when you will get it.
- **RPC** – Remote Procedure Call systems allow function calls to be made between processes on different computers.
 - Example: An iPhone app calling a service on the Internet which returns the weather forecast.

Scalability

Scalability describes a system's ability to increase, or scale, its workload. There are two common means of scaling a system, vertical and horizontal. Vertical scaling is often referred to as scaling up and horizontal scaling is often referred to as scaling out.

Vertical scaling, in simple terms, involves buying a faster computer. Vertical scaling places little burden on the application and is an easy way to increase capacity. However, modern CPUs are no longer increasing in performance at the rates they once did, making it expensive, or impossible in many cases, to scale vertically beyond a given threshold.

Horizontal scaling involves adding more computers to a pool or cluster of systems which run an application collectively. Horizontally scaled applications take advantage of multiple CPUs and/or multiple systems to grow performance. Applications must be designed for distribution across multiple computers to take advantage of horizontal scaling. Extreme examples of

horizontal scaling allow applications to harness thousands of CPUs to perform demanding tasks in very short periods of time. Apache Thrift is a tool particularly well suited to building horizontally scaled distributed applications.

These three communications paradigms can be used to tackle just about any inter-process communication task. Let's look at how Apache Thrift applies in each of these settings.

1.5.1 Streaming

Streaming systems deal in continuous flows of bytes. Streaming servers may transmit streams to one or more clients. Some streams have temporal requirements, for instance streaming movies which require frames to arrive at least as fast as they are viewed. Some streams are more batch oriented, for example a background file transfer. Streaming systems are typically designed for communications where data transfers flow in one direction and are of large or undefined size.

Streaming systems are frequently low overhead in nature. They tend to be large bandwidth consumers and therefore strive for efficiency over ease of use. Unicasting is often used to eliminate connection and retransmission overhead, and in some cases multicasting is used to allow the



Figure1. 8 - Streaming systems are often purpose built to meet performance needs

server to send a single message to multiple clients. Streaming systems may use data compression mechanisms to reduce network impact as well.

Apache thrift does not typically play a role in streaming data services. However control APIs used to subscribe to streams and perform other setup and configuration tasks may be a good fit for RPC. Apache Thrift supports one way messages which may suffice to stream information from a client to a server in some applications. Apache Thrift serialization may be useful in streaming solutions which require streaming structured data across languages.

1.5.2 Messaging

Messaging is an asynchronous communications model allowing queued communications to take place independently of the speed of the producer or consumer. Full service messaging systems support reliable communications over unreliable links with features such as store and forward, transactions, multicasting and publish/subscribe. Systems such as WebsphereMQ, ActiveMQ, RabbitMQ and JMS fit into this category.

Lightweight messaging systems are more appropriate for messaging at high data rates with minimum latency as a design imperative. Open source systems such as LCM and ZeroMQ, and commercial systems such as TIBCO Rendezvous implement a lightweight framework supporting many standard messaging features, with performance as an overriding design goal. Such high speed messaging systems strike a balance between the performance of streaming systems and the features of heavier weight messaging systems.

Apache Thrift is not a message queuing platform but it can fulfill the serialization responsibilities associated with cross language messaging. For example, if you are interested in using RabbitMQ to send messages between a C++ and a Java application, you may need a common serialization format. User defined message types can be described in Apache Thrift IDL and then compiled to support serialization in any Apache Thrift language. A C# program could serialize a C# object and then send it as a message through the messaging system, whereupon an Objective-C application could receive the message and deserialize it into a native Objective-C object.

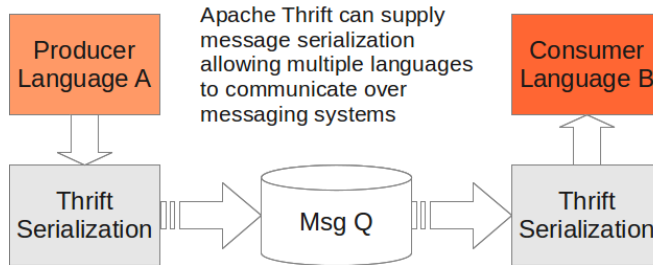


Figure 1.9 - Messaging systems can make use of Apache Thrift serialization

RPC systems, under the covers, send messages between clients and servers to make function calls and return results. Therefore it is possible to implement RPC systems on top of messaging systems. There is an experimental Apache Thrift transport in the thrift/contrib folder which uses ZeroMQ, allowing Apache Thrift RPC to operate over the ZeroMQ messaging platform.

1.5.3 Remote Procedure Calls

Making function calls to complete the work of a program is fairly natural in most languages. Remote Procedure Call systems allow function callers and function implementers to live in different processes, as demonstrated by our sample RPC application earlier in this chapter. Systems such as Apache Thrift, JavaRMI, COM, DCE, SOAP and CORBA provide RPC style functionality.

Unlike messaging systems, the client and the server in an RPC exchange must be up and running at the same time. The client waits for the server's response in many RPC environments, just as if the client were calling a local function. This couples the client to the server in a much closer way than that of a messaging system. However SOA platforms, such as Apache Thrift, lend flexibility to the client and server relationship in several ways.

Most Apache Thrift languages support asynchronous client interfaces. This allows the client to call the server and then go about other business, checking back later to see if the response is available. This is similar to the way a client and server would interact over a messaging platform.

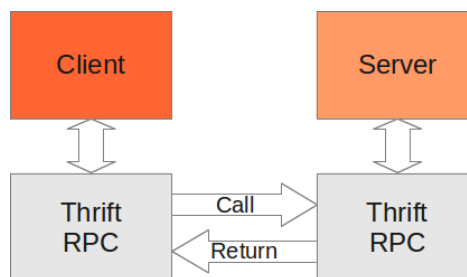


Figure 1.10 - Apache Thrift allows clients to call functions hosted in remote servers

Apache Thrift also supports one way messages. One way messages are “fire and forget” style communications, the client calls the one way function with the appropriate parameters and then goes about its business. The server receives the message but does not reply. This is akin to the way single direction messages are sent in a streaming environment or a messaging environment without queuing.

Choosing the right communications platform often involves a combination of RPC, messaging and streaming style solutions. Thrift is well suited to such hybrid environments, easily adapting to an assortment of languages and communications platforms. Thrift provides a rich RPC framework and can provide the serialization needs of messaging and streaming applications.

1.6 The cross language communications landscape

Apache Thrift was originally developed at Facebook to address performance and functionality challenges facing programmers working with REST interfaces and the LAMP stack (an acronym for Linux, Apache Web Server, MySQL database and programming languages starting with P [PHP/Perl/Python]). The Facebook team examined several potential 3rd party options looking for an efficient cross language communications system. To quote the 2007 Facebook white paper, their list of possibilities and conclusions were:

SOAP. XML-based. Designed for web services via HTTP, excessive XML parsing overhead.

CORBA. Relatively comprehensive, debatably overdesigned and heavyweight. Comparably cumbersome software installation.

COM. Embraced mainly in Windows client software. Not an entirely open solution.

Pillar (Facebook internal). Lightweight and high-performance, but missing versioning and abstraction.

Protocol Buffers. Closed-source, owned by Google. Described in Sawzall paper. *

NOTE Protocol Buffers, while listed in the Facebook white paper as closed source, has since been open sourced.

REST, SOAP, Protocol Buffers and the Apache Avro platform are perhaps the technologies most often considered as alternatives to Apache Thrift. Each of these technologies is unique and all have their place.

1.6.1 REST, SOAP and XML-RPC

REST is an acronym for Representational State Transfer and is the typical means for web browsers to retrieve content from web servers. RESTful web services use the REST communication technique to make RPC style function calls over HTTP. The well understood and widely supported HTTP protocol gives REST based services broad reach. RESTful services often use XML, JSON or similar text based formats for data interchange.

Simple Object Access Protocol (SOAP) and XML-RPC are both closely related to the RESTful approach. SOAP and XML-RPC rely on XML for carrying their payload between the client and the server and are frequently built upon HTTP, though other transports can also be used. Optimizations are available which attempt to reduce the burden of transmitting XML and there are versions of SOAP which use JSON, among other off shoots.

The key benefit of RESTful services and other HTTP friendly RPC technologies is their broad interoperability. By transmitting standards based text documents (XML, JSON, etc.) over the ubiquitous HTTP protocol, almost any application or language can be engaged. The principal drawback of these approaches is that they tend to vastly underperform more native platforms, such as Apache Thrift. The serialization burden created by generating and decoding text documents, and the transmission overhead associated with them, can make them a nonstarter in performance sensitive settings.

1.6.2 Google Protocol Buffers

Google Protocol Buffers and Apache Thrift are similar in performance and from a serialization and IDL stand point. Official Google Protocol Buffer language support is limited to C++, Java and Python. Google Protocol Buffers are used by a large community of developers however, and many projects on the web expand the languages supported by Protocol Buffers to a selection comparable to that of Apache Thrift.

Apache Thrift supports a full RPC client and server framework in one integrated distribution. Google Protocol Buffers has a narrower message serialization focus, though several RPC style systems for Protocol Buffers are available in other projects.

Another difference between the platforms is support for transmission of collections. Apache Thrift supports transmission of three common containers types: lists, sets and maps. Protocol Buffers supply a repeating field feature rather than support for containers, producing similar capabilities through a lower level construct.

Protocol Buffers are robust, well documented and backed by a large corporation which contrasts with the open source nature of Apache Thrift.

1.6.3 Apache Avro

Apache Avro is a serialization framework designed to package the serialization schema with the data serialized. This contrasts with the Apache Thrift and Protocol Buffers approach of describing the schema (data types and service interfaces) in IDL. Apache Avro interprets the schema on the fly and Apache Thrift generates code to interpret the schema at compile time. In general, combining the schema with the data works well for long lived objects serialized to disk. However, such a model can add complexity and overhead to real time RPC style communications. Arguments and optimizations can be made to turn these observations on their head of course.

Apache Avro supports eight programming languages at the time of this writing and offers an RPC framework as well. Avro supports the same containers present in Apache Thrift although Apache Avro maps only allow strings as keys. The use of dynamically interpreted embedded

schemas in Apache Avro and the use of compiled IDL in Apache Thrift is the key distinction between these two platforms.

Apache Thrift Versions

The Thrift framework was originally developed at Facebook and released as open source in 2007. The project became an Apache Software Foundation incubator project in 2008, after which four early versions were released.

0.2.0 released 2009-12-12

0.3.0 released 2010-08-05

0.4.0 released 2010-08-23

0.5.0 released 2010-10-07

In 2010 the project was moved to top level status where several additional versions have been released.

0.6.0 released 2011-02-08

0.6.1 released 2011-04-25

0.7.0 released 2011-08-13

0.8.0 released 2011-11-29

0.9.0 released 2012-10-15

0.9.1 released 2013-07-16

0.9.2 released 2014-04-01

1.0.0 released 2014-10-15

1.7 What Apache Thrift doesn't do

Apache Thrift is a powerful cross language serialization platform packaged with a robust RPC framework, however it is not a fit for all applications.

While the Apache Thrift type system has many nice features, including support for constants, unions, sets, maps and lists, Apache Thrift does not support self-referential structures. This means that trees, and other graphs will have to be reorganized to transit an Apache Thrift Service interface. With some creativity any data structure can be recomposed with lists and maps. Many experienced distributed system interface designers prefer to avoid passing uncontrolled data structures through RPC interfaces, making this shortfall a potential boon for inexperienced interface designers.

Thrift is not a good choice for transmission of large datasets. If you need to move large blocks of data between processes it may be best to consider other options. Apache Thrift RPC and serialization are designed to be fast and efficient in the context of traditional function call payloads, unit data sizes much more than a megabyte may become unwieldy. Again, this is less of an Apache Thrift failing and more of a general RPC best practice.

Apache Thrift is not a messaging system. The Apache Thrift serialization system can be an effective way to encode messages transmitted over messaging systems, however Apache Thrift

RPC is not designed to provide publish/subscribe, queuing, multicast, broadcast or other messaging specific features.

If absolute performance is your goal and you have no need for serialization, the cross language and interface evolution overhead added by Thrift may be unnecessary. Thrift fares well in performance comparisons with competing platforms but it will not be as fast as rigid purpose built code.

1.8 Summary

Here are the most important points to take away from the chapter:

Apache Thrift is a cross language serialization and RPC framework

Apache Thrift supports a wide array of languages and platforms

Apache Thrift makes it easy to build high performance RPC services

Apache Thrift is a good fit for service oriented architectures (SOA)

Apache Thrift is an Interface Definition Language (IDL) based framework

IDLs allow you to describe interfaces, automating implementation generation

The Apache Thrift IDL Compiler generates code in over 15 programming languages

Apache Thrift includes a modular serialization system, providing several built in serialization protocols and support for custom serialization solutions

Apache Thrift includes a modular transport system, providing built in memory, disk and network transports and making it easy to add additional transports