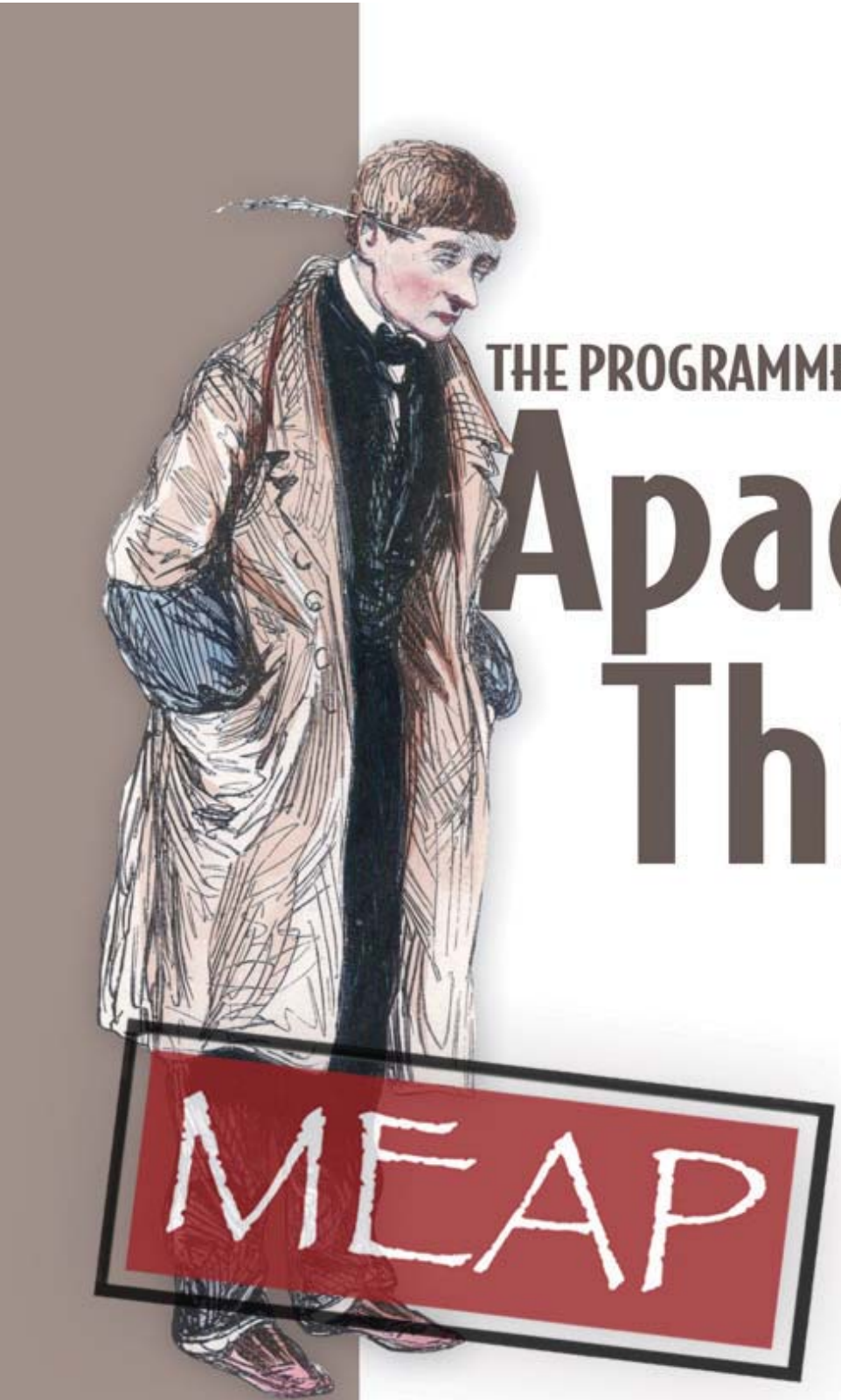


THE PROGRAMMER'S GUIDE TO Apache Thrift

MEAP

Randy Abernethy

MANNING

MEAP Edition
Manning Early Access Program
The Programmer's Guide to Apache Thrift
Version 17

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

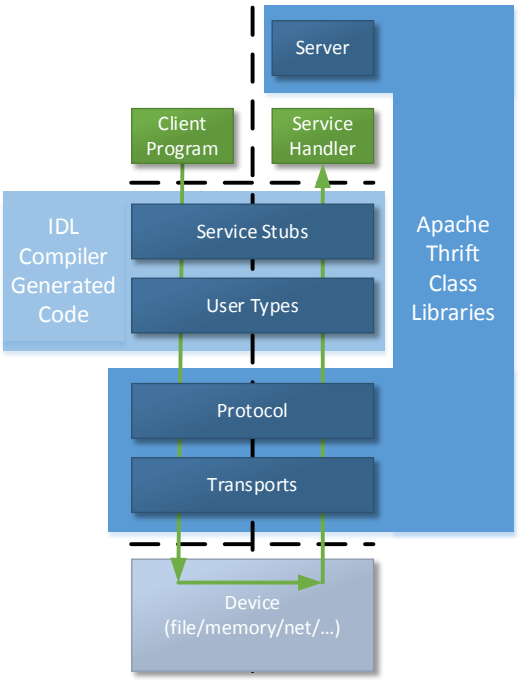
brief contents

Part 1

Apache Thrift Overview

Apache Thrift is an open source cross language serialization and remote procedure call (RPC) framework. With support for over 20 programming languages, Apache Thrift can play an important role in many distributed application solutions. 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 cross language services with little more than a few lines of code.

Part 1 of this book will help you understand how Apache Thrift fits into modern cloud based application models while imparting a high level understanding of the Apache Thrift framework architecture. Part 1 also gets you started with basic Apache Thrift setup and debugging and a look at building a simple cross language hello world service.



1

Introduction to Apache Thrift

This chapter covers

- Using Apache Thrift to unify polyglot systems
- Simplifying the creation of high-performance networked services
- Introducing the Apache Thrift modular serialization system
- Creating a simple Apache Thrift cross-language microservice
- Comparing Apache Thrift with other cross-language communications frameworks

Modern software systems live in a networked world. Network communications are critical to the tiniest embedded systems in the Internet of Things through to the weightiest of relational databases anchoring traditional multitier applications. As new software systems increasingly embrace dynamically scheduled, containerized microservices, lightweight, high-performance, language-agnostic network communications are ever more important.

But how to wire all these things together, the old and the new, the big and the small? How do you package a message from a service written in one language in such a way that a program written in any other language can read it? How do you design services that are fast enough for high-performance, backend cloud systems but accessible by front-end scripting technologies? How do you keep things lightweight to support efficient containers and embedded systems? How do you create interfaces that can evolve over time without breaking existing components? How do you do all of this in an open, vendor-neutral way and, perhaps most important, how can you do it all precisely once, reusing the same communications primitives across a broad platform? For companies such as Facebook, Evernote, and Twitter, the answer is Apache Thrift.

This chapter introduces the Apache Thrift framework and its role in modern distributed applications. You'll look at why Apache Thrift was created and how it helps programmers build high-performance, cross-language services. To begin, you'll consider the growing need for

multi-language integration and examine the role Apache Thrift plays in polyglot application development. Next, you'll look at the two key functions of Apache Thrift and walk through the construction of a simple Apache Thrift service. At the end of the chapter we'll compare Apache Thrift to several other tools offering similar features to help you determine when Apache Thrift might be a good fit.

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 (http://www.tiobe.com/index.php/tiobe_index) was attributed to six programming languages: Java, C, C++, Perl, Visual Basic, and PHP. In 2013, it took nearly twice as many languages to capture the same 80%, adding Objective-C, C#, Python, JavaScript, and Ruby to the list (see figure 1.1). In early 2016 the entire Tiobe top 20 didn't add up to 80% of the mind share. In Q4 2015, Github reported 19 languages all having more than 10,000 active repositories (http://githut.info/), adding Swift, Go, Scala, and others to the list.

2003 Tiobe Index top four quintiles

Language	Rating	Cumulative
Java	23.08%	23.08%
C	18.47%	41.55%
C++	15.56%	57.12%
Perl	9.42%	66.54%
(Visual) Basic	7.81%	74.35%
PHP	4.76%	79.11%

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.

2013 Tiobe Index top four quintiles

Language	Rating	Cumulative
C	17.81%	17.81%
Java	16.66%	34.47%
Objective-C	10.36%	44.82%
C++	8.82%	53.64%
PHP	5.99%	59.63%
C#	5.78%	65.41%
(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 (http://www.tiobe.com).

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 TypeScript, while programmers in the basement might be using C with embedded systems (no aversion to sunlight implied). 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 simpler and easier to maintain. For example, in large-scale data analysis, horizontal scaling is instrumental to achieving acceptable performance. Functional programming languages such as Haskell, Scala, and Clojure tend to fit naturally here, allowing analytic systems to scale out without complex concurrency concerns.

Platforms drive language adoption as well. Objective-C exploded in popularity when Apple released the iPhone, and Swift is following suit. Go is the language of the booming container ecosystem, responsible for Docker, Kubernetes, etcd, and other essentials. Those programming for the browser will have teams competent with JavaScript, TypeScript, and/or Dart, while the game and GUI world still often codes in C++ for top-performing graphics. 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 (figure 1.2). Dynamic programming languages such as Groovy and Ruby are often used for testing, while Lua, Perl, and Python are popular for prototyping, and PHP has a long history with the web. Build systems such as the Groovy-based Gradle and the Ruby-based Rake also provide innovative capabilities.

Figure1.1 Efficient translators are a core asset of any multi-language assembly.

The polyglot story isn't 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 when building and testing. If managed improperly, these costs can quickly overshadow the benefits of a multi-language strategy.

One of the key strengths of Apache Thrift is its ability to simplify, centralize, and encapsulate the cross-language aspects of a system. Apache Thrift offers broad support, in tree, for polyglot application development. Every language mentioned previously is supported by the Apache Thrift project, more than 20 languages in all, and growing (see table 1.1). This unrivaled direct support for existing languages and the Apache Thrift community's rapid addition of support for new languages can help organizations maximize the potential of polyglotism while minimizing the downsides. The more your programs mirror the dialog on the floor of the United Nations General Assembly, the more you'll need professional translators such as Apache Thrift to streamline communications.

Table 1.1 Languages Supported by Apache Thrift

AS3	C	C++	C#
D	Dart	Delphi	Erlang
Go	Haskell	Haxe	Java
JavaScript	Lua	Node.js	Objective-C
OCaml	Perl	PHP	Python
Ruby	SRust	Smalltalk	TypeScript

1.2 Application integration with Apache Thrift

Whether your application uses multiple platforms and languages or not, it's likely that its operations span multiple processes over networks and time. At times these processes will need to communicate, either through a file on disk, through a buffer in memory, or across networks. Two central concerns are associated with inter-process communications:

- Type serialization
- Service implementation

Let's consider each in turn.

1.2.1 Type serialization

Serialization is a basic function in any cross-platform/language exchange. For example, imagine an application for the music industry that uses NATS as a messaging system to move song data between processes (see figure 1.3). Using NATS, the team can send/receive messages rapidly between their remote processes written in Java and Python. The question is, can the programs read the musical messages when sent by another language? Python objects are represented differently in memory than Java objects. If a Python program sent the raw memory bits for its music track data to a Java program, fireworks would ensue.

Figure 1.3 Apache Thrift can be used to serialize data in cross-platform messaging scenarios.

To solve this problem, you need a data serialization layer on top of the messaging platform. Why not send everything back and forth in JSON, one might ask? Using a standard format such as JSON is part of a solution; however, you must still answer questions such as: how are data fields ordered when sending multi-field messages, what happens when fields are missing, and what does a language that doesn't directly support a data type do when receiving that data type? These and many other questions cannot be answered by a data layout specification such as JSON, YAML, or XML. Different languages frequently produce different, though legally formatted, documents for the same dataset.

IDL AND TYPES

Apache Thrift provides a modular serialization framework that addresses these issues. With Apache Thrift, developers define abstract data types in an Interface Definition Language (IDL). This IDL can then be compiled into source code for any supported language. The generated code provides complete serialization and deserialization logic for all of the user's defined types. Apache Thrift ensures that types written by any language can be read by any other language. The following listing shows Apache Thrift IDL type definitions.

Listing 1.1 Apache Thrift IDL type definitions

```
namespace * music

enum PerfRightsOrg {
    ASCAP = 1
    BMI = 2
    SESAC = 3
    Other = 4
}

typedef double Minutes

struct MusicTrack {
    1: string title
    2: string artist
    3: string publisher
    4: string composer
    5: Minutes duration
    6: PerfRightsOrg pro
```

```
}
```

Certain people complain that creating IDL is an extra step, slowing the development process. I've found that it's the opposite. IDL forces you to carefully consider your interfaces in isolation, free of noisy implementation code. This may be the most important time you spend on a system design. IDL is also lightweight, easy to modify and experiment with, and often useful as a communications tool on the business side as well.

Users may say schemaless systems are more flexible and that IDL is brittle. The truth is, whether you document your schema or not, you still have a schema if you're reading and interpreting data. Implied (undocumented) schemas can be the source of fairly treacherous application errors and create a burden on developers who need to interact with the data or extend the system. If you have no definition for the data layout you read and write except the code that reads and writes it, it will be slow going when you want to extend the system. How many bits of code throughout the system depend on this implied schema? How do you change such a thing?

The popularity of NoSQL systems, many of which are schemaless, creates another role for IDL. You can now document your types in a single place and use those types in service calls, with messaging systems and in storage systems such as Redis, MongoDB, and others.

Several systems reverse the process and generate their schema from a given coded solution. Annotation driven systems, such as Java's JAX-RS, can work this way. This approach makes it easy to allow implementation details to bias the interface definition, straining portability and clarity. It's generally much more work to modify implementation code than it is to modify IDL. Also, you have no guarantee that another vendor's code generator will create compatible code from a foreign schema. This is a problem any time multiple vendors are involved in a communications solution.

Apache Thrift sidesteps many of these problems by providing a single source of truth, the IDL. Apache Thrift supplies vendor-independent support for a single IDL across a wide array of programming languages, and the Apache Thrift cross-language test suit is constantly at work verifying interoperability as the framework grows.

INTERFACE EVOLUTION

IDL creates a contract that all parties can rely upon and that code generators can use to create working serialization operations, ensuring the contract is adhered to. Yet IDL schemas need not be brittle. Apache Thrift IDL supports a range of interface evolution features which, when used properly, allow fields to be added and removed, types to be changed, and more.

Support for interface evolution greatly simplifies the task of ongoing software maintenance and extension. Modern engineering sensibilities such as microservices, Continuous Integration (CI), and Continuous Delivery (CD) require systems to support incremental improvements without impacting the rest of the platform. Tools that supply no form of interface evolution tend to "break the world" when changed. In such systems, changing an interface means all the clients and servers using that interface must be rewritten and/or recompiled, then redeployed in a big bang.

Apache Thrift interface evolution features allow multiple interface versions to coexist seamlessly in a single operating environment. This makes incremental updates viable, enabling CI/CD pipelines and empowering individual Agile teams to deliver business value at their own cadence.

Continuous Integration (CI) and 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. Continuous Delivery takes CI one step further, migrating successfully merged code to evaluation/staging systems and ultimately into production, many times per day. The goal of continuous systems is to take many small risks and 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, making it more difficult to identify and repair conflicts and bugs.

MODULAR SERIALIZATION

Apache Thrift provides pluggable serializers, known as protocols, allowing you to use any one of several serialization formats for data exchange, including binary for speed, compact for size, and JSON for readability. The same contract (IDL) can remain in place even as you change serialization protocols. This modular approach allows custom serialization protocols to be added as well. Because Apache Thrift is community managed and open source, you can easily change or enhance functionality and push it upstream when needed (patches are always welcome at the Apache Thrift project).

1.2.2 Service implementation

Services are modular application components that provide interfaces accessible over a network. Apache Thrift IDL allows you to define services in addition to types (see listing 1.2). Like types, IDL services can be compiled to generate stub code. Service stubs are used to connect clients and servers in a wide range of languages.

Listing 1.2 /ThriftBook/part1/hello/sail_stats.thrift

```
service SailStats {
   double get_sailor_rating(1: string sailor_name)
   double get_team_rating(1: string team_name)
   double get_boat_rating(1: i64 boat_serial_number)
   list<string> get_sailors_on_team(1: string team_name)
   list<string> get_sailors_rated_between(1: double min_rating,
                                          2: double max_rating)
   string get_team_captain(1: string team_name)
}
```

Imagine you have a module that tracks and computes sailing team statistics and that this module is built into a Windows C++ GUI application designed to visualize wind flow dynamics. As it happens, your company's web dev team wants to use the sail stats module to enhance a

client-facing, Node.js-based web application on Linux. Faced with multiple languages and platforms and the "laziness" axiom (wanting to write as little code as possible), Apache Thrift could be a good solution (see figure 1.4).

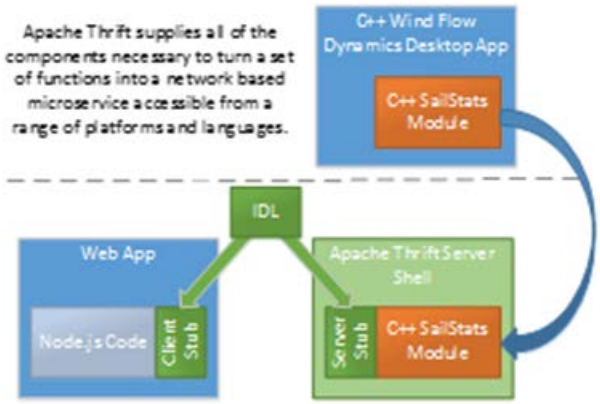


Figure 1.4 The Apache Thrift RPC framework enables cross-platform services.

With Apache Thrift you could repackage the sail stats functions as a microservice and provide the Node.js programmers with access to the service through an easy-to-use Node.js client stub. To create the sail stats microservice you need only define the service interface in IDL, compile the IDL to create client and server stubs for the service, select one of the prebuilt Apache Thrift servers to host the service, and then assemble the parts.

PREBUILT SERVER SHELLS

It's important to note that, unlike standalone serialization solutions, Apache Thrift comes with a complete set of server shells, ready to use, in almost all the supported languages. This sidesteps the difficult and repetitive process of building custom network servers. The prebuilt Apache Thrift servers are also small and focused, providing only the functionality necessary to host Apache Thrift services. A typical Apache Thrift server will consume an order of magnitude less memory than an equivalent Tomcat deployment. This makes Apache Thrift servers a good choice for containerized microservices and embedded systems that don't have the resources necessary to run full-blown web or application servers.

Microservices and Service Oriented Architecture (SOA)

The microservice and SOA approaches to distributed application design break applications down into services, which are remotely accessible, autonomous modules composed of a set of closely related functions. Such systems 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. These services are typically (and in the best case) stateless and loosely coupled, communicating with clients through a formal interface contract. Services

may be internal to an organization or support clients across business boundaries. The distinction between SOA services and microservices is subtle, but most agree that microservices are a subset of SOA services in which the services are more atomic and independently deployable.

MODULAR TRANSPORTS

Apache Thrift also offers a pluggable transport system. Apache Thrift clients and servers communicate over transports that adapt Apache Thrift data flows to the outside world. For example, the TSocket transport allows Apache Thrift applications to communicate over TCP/IP sockets. You can use prebuilt transports for other communications schemes, such as named pipes and UNIX domain sockets. Custom transports are easy to craft as well. Apache Thrift also supports offline transports that allow data to be serialized to disk, memory, and other devices.

A particularly elegant aspect of the Apache Thrift transport model is support for layered transports. Protocols serialize application data into a bit stream. Transports read and write the bytes, making any type of manipulation possible. For example, the TZLibTransport is available in many Apache Thrift language libraries and can be layered on top of any other transport to achieve high-ratio data compression. You can branch data to loggers, fork requests to parallel servers, encrypt, and perform any other manner of manipulation with custom-layered transports.

1.3 Building a simple service

To get a better understanding of the practical aspects of Apache Thrift, you'll build a simple microservice. Your service will be designed to supply various parts of your enterprise with a daily greeting. The service will expose a single "hello_func" function that takes no parameters and returns a greeting string. To see how Apache Thrift works across languages you'll build clients in C++, Python, and Java.

1.3.1 The Hello IDL

Most projects involving Apache Thrift begin with careful consideration of the interface components involved. Apache Thrift IDL is similar to C in its notation and makes it easy to define types and services shared across systems. Apache Thrift IDL is plain text saved in files with a ".thrift" extension (see the following listing).

Listing 1.3 /ThriftBook/part1/hello/hello.thrift

```
service HelloSvc {                  #1
   string hello_func()     #2
}
```

Your hello.thrift IDL file declares a single service interface called HelloSvc (#1) with a single function, hello_func() (#2). The function accepts no parameters and returns a string. To use this interface you can compile it with the Apache Thrift IDL compiler. The IDL compiler

binary is named “thrift” on UNIX-like systems and “thrift.exe” on Windows. The compiler expects two command line arguments, an IDL file to compile and one (or more) target languages to generate code for. Here’s an example session that generates Python stubs for your HelloSvc:

```
/ThriftBook/part1/hello$ ls -l
-rw-r--r-- 1 root root     88 Feb 16 17:01 hello.thrift
/ThriftBook/part1/hello$ thrift --gen py hello.thrift            #1
/ThriftBook/part1/hello$ ls -l
drwxr-xr-x 4 root root   4096 Feb 17 00:16 gen-py                #2
-rw-r--r-- 1 root root     88 Feb 16 17:01 hello.thrift
```

In the previous session the IDL Compiler is invoked with the --gen py switch (#1), which causes the compiler to create a gen-py directory (#2) to house the emitted Python code for your hello.thrift IDL. The directory contains client/server stubs for all the services defined and serialization code for all the user-defined types.

1.3.2 The Hello server

Now that you have your support code generated, you can implement your service and use a prebuilt Apache Thrift server to house it. The following listing provides a sample server coded in Python.

Listing 1.4 /ThriftBook/part1/hello/hello_server.py

```
import sys                                                       #1
sys.path.append("gen-py")                                        #1
from hello import HelloSvc                                       #1

from thrift.transport import TSocket                             #2
from thrift.transport import TTransport                          #2
from thrift.protocol import TBinaryProtocol                      #2
from thrift.server import TServer                                #2

class HelloHandler:                                              #3
    def hello_func(self):
        print("[Server] Handling client request")
        return "Hello from the python server"

handler = HelloHandler()                                         #4
proc = HelloSvc.Processor(handler)                               #4

trans_svr = TSocket.TServerSocket(port=9090)                     #5
trans_fac = TTransport.TBufferedTransportFactory()               #6
proto_fac = TBinaryProtocol.TBinaryProtocolFactory()             #7
server = TServer.TSimpleServer(proc, trans_svr, trans_fac, proto_fac)  #8
server.serve()                                                   #9
```

At the top of your server listing you use the built-in Python sys module to add the gen-py directory to the Python Path. This allows you to import the generated service stubs for your HelloSvc service (#1).

Your next step is to import several Apache Thrift library packages. TSocket provides an endpoint for your clients to connect to, TTransport provides a buffering layer, TBinaryProtocol

will handle data serialization, and TServer will give you access to the prebuilt Python server classes (#2).

The next block of code implements the HelloSvc service itself through the HelloHandler class. This class is called a handler in Apache Thrift because is handles all the calls made to the service. All the service methods must be represented in the Handler class; in your case this is the `hello_func()` method (#3). In real projects, almost all your time and effort is spent here, implementing your services. Apache Thrift takes care of all of the wiring and boilerplate code.

Next you create an instance of your handler and use it to initialize a processor for your service. The processor is the server side stub generated by the IDL compiler that turns network service requests into calls to the appropriate handler function (#4).

The Apache Thrift library offers endpoint transports for use with files, memory, and various network types. The example here creates a TCP server socket endpoint to accept client connections on TCP port 9090 (#5). The buffering layer ensures that you make efficient use of the underlying network, transmitting bits only when an entire message has been serialized (#6). The binary serialization protocol transmits your data in a fast binary format with little overhead (#7).

Apache Thrift 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 the most basic server functionality (#8). Once constructed, you run the server by calling the `serve()` method (#9).

The following example session runs your Python server:

```
/ThriftBook/part1/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/part1/hello$ python hello_server.py
```

The Python server took approximately seven lines of code, excluding imports and the service implementation. The story is similar in C++, Java, and most other languages. This is a basic server but the example should help you see how much leverage Apache Thrift gives you when it comes to quickly creating cross-language microservices.

1.3.3 A Python client

Now that you have your server running, let's create a simple Python client to test it, as shown in the following listing.

Listing 1.5 /ThriftBook/part1/hello/hello_client.py

```
import sys
sys.path.append("gen-py")
from hello import HelloSvc                                  #1

from thrift.transport import TSocket                        #2
from thrift.transport import TTransport                     #3
from thrift.protocol import TBinaryProtocol                 #4
```

```
trans = TSocket.TSocket("localhost", 9090)                    #5
trans = TTransport.TBufferedTransport(trans)                  #6
proto = TBinaryProtocol.TBinaryProtocol(trans)                #7
client = HelloSvc.Client(proto)                               #8

trans.open()                                                  #9
msg = client.hello_func()                                     #10
print("[Client] received: %s" % msg)                          #11
trans.close()                                                 #12
```

The Python client begins by importing the same HelloSvc module used by the server, but the client will use the client-side stubs for the hello service (#1). You'll also import three modules from the Apache Thrift Python library. The first is TSocket, which is used on the client side to make a TCP connection to the server socket (#2), because you might guess the client must use a client side transport compatible with the server transport. The next import pulls in TTransport which will provide a network buffer (#3), and the TBinaryProtocol import allows you to serialize messages to the server (#4). Again, this must match the server implementation.

Your next block of code initializes the TSocket with the host and port to connect to (#5). You'll wrap the socket transport in a buffer (#6) and finally wrap the entire transport stack in the TBinaryProtocol (#7), creating an I/O stack that can serialize data to and from the server.

The I/O stack is used by the client stub, which acts as a proxy for the remote service (#8). Opening the transport causes the client to connect to the server (#9). Invoking the `hello_func()` method on the Client object serializes your call request with the binary protocol and transmits it over the socket to the server, then deserializes the returned result (#10). The program prints out the result (#11) and then closes the connection using the transport `close()` method (#12).

Here's a sample session running the above client (the Python server must be running in another shell to respond).

```
/ThriftBook/part1/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/part1/hello$ python hello_client.py
[Client] received: Hello from the python server
```

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

1.3.4 A C++ client

To broaden your perspective and demonstrate the cross-language aspects of Apache Thrift, let's build two more clients for the hello server, one in C++ and one in Java. You'll start with the C++ client.

First you need to compile the service definition again, this time generating C++ stubs:

```
/ThriftBook/part1/hello$ thrift --gen cpp hello.thrift              #1
/ThriftBook/part1/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
```

Running the IDL Compiler with the "--gen cpp" switch (#1) causes it to emit C++ files in the gen-cpp directory, roughly equivalent to those generated for Python, producing C++ headers (.h) and source files (.cpp). The gen-cpp/HelloSvc.h header (#1) contains the declarations for your service, and the gen-cpp/HelloSvc.cpp source file contains the implementation of the service stub components.

The code for a HelloSvc C++ client with the same functionality as the Python client appears in the following listing.

Listing 1.6 /ThriftBook/part1/hello/hello_client.cpp

```
#include "gen-cpp/HelloSvc.h"
#include <thrift/transport/TSocket.h>
#include <thrift/transport/TBufferTransports.h>
#include <thrift/protocol/TBinaryProtocol.h>
#include <boost/make_shared.hpp>
#include <iostream>
#include <string>

using namespace apache::thrift::transport;                  #1
using namespace apache::thrift::protocol;                   #1
using boost::make_shared;                                   #2

int main() {
    auto trans_ep = make_shared<TSocket>("localhost", 9090);
    auto trans_buf = make_shared<TBufferedTransport>(trans_ep);
    auto proto = make_shared<TBinaryProtocol>(trans_buf);
    HelloSvcClient client(proto);

    trans_ep->open();
    std::string msg;
    client.hello_func(msg);                                 #3
    std::cout << "[Client] received: " << msg << std::endl;
    trans_ep->close();
}
```

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

The C++ `main()` function corresponds line for line with the Python code with one exception; `hello_func()` doesn't return a string conventionally, rather it returns the string through an out parameter reference (#3).

The Apache Thrift language libraries are generally wrapped in namespaces to avoid conflicts in the global namespace. In C++ all of the Apache Thrift library code is located within the “apache::thrift” namespace. The using statements here provide implicit access to the necessary Apache Thrift library code (#1).

Apache Thrift strives to maintain as few dependencies as possible to keep the development environment simple and portable; however, exceptions do exist. For example, the Apache Thrift C++ library relies on the open source Boost library. In this example, several objects are wrapped in boost::shared_ptr (#2). Apache Thrift uses shared_ptr to manage the lifetimes of almost all of the key objects involved in C++ service operations.

Those familiar with C++ will know that shared_ptr has been part of the standard library since C++11. While the sample code is written in C++11, Apache Thrift supports C++98 as well, requiring the use of the Boost version of shared_ptr (C++98 support will likely be dropped in the future, moving all Boost namespace elements to the std namespace).

The following listing shows a Bash session that builds and runs the C++ client.

Listing 1.7 Bash session running C++ client

```
$ 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
$ g++ --std=c++11 hello_client.cpp gen-cpp/HelloSvc.cpp -lthrift              #1
$ 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
$ ./a.out                                                                     #2
[Client] received: Hello thrift, from the python server
```

Here you use the Gnu C++ compiler to build the hello_client.cpp file into an executable program (#1). Clang, Visual C++, and other compilers are also commonly used to build Apache Thrift C++ applications.

For the C++ build you must compile the generated client stubs 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, you can run your executable C++ client and make a cross-language RPC call. The C++ compiler builds your source into an a.out file that produces the same result as the Python client when executed (#2).

1.3.5 A Java client

As a final example let's put together a Java client for the service. Your first step is to generate Java stubs for the service, as shown in the following listing.

Listing 1.8 Generating Java stubs

```
/ThriftBook/part1/hello$ thrift --gen java hello.thrift              #1
/ThriftBook/part1/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
```

The "–-gen java" switch causes the IDL Compiler to emit Java code for your interface in the gen-java directory (#1), creating a HelloSvc class with nested client and server stub classes. The following listing provides the source for a Java client that parallels the prior Python and C++ clients.

Listing 1.9 /ThriftBook/part1/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 trans = new TSocket("localhost", 9090);
        TBinaryProtocol protocol = new TBinaryProtocol(trans);
        HelloSvc.Client client = new HelloSvc.Client(protocol);

        trans.open();
        String str = client.hello_func();
        System.out.println("[Client] received: " + str);
        trans.close();
    }
}
```

In typical Java form, the `main()` method lives inside a class with the same name as the containing file and the rest of the code is a rehash of the previous clients. The one noticeable difference is that the Java client has no buffering layer above the endpoint transport because the socket implementation in Java is based on a stream class that buffers internally, so no additional buffering is required.

The following listing shows a build and run session for the Java client.

Listing 1.10 Build and run session for the Java client

```
/ThriftBook/part1/hello$ javac -cp /usr/local/lib/libthrift-1.0.0.jar:\  #1
                               /usr/share/java/slf4j-api.jar:\           #2
                               /usr/share/java/slf4j-nop.jar \           #3
```

```
                              HelloClient.java gen-java/HelloSvc.java   #4
/ThriftBook/part1/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/part1/hello$ java -cp /usr/local/lib/libthrift-1.0.0.jar:\
                                  /usr/share/java/slf4j-api.jar:\
                                  /usr/share/java/slf4j-nop.jar:\
                                  ./gen-java:\                        #5
                                  . \                                 #5
                              HelloClient
[Client] received: Hello thrift, from the python server
```

The Java compile includes three dependencies; the first is the Apache Thrift Java library jar (#1). The IDL generated code for your service also depends on SLF4J, a popular Java logging façade. The slf4j-api jar (#2) is the façade and the slf4j-nop jar (#3) is the nonoperational logger that discards logging output. The Java files generate byte code in .class files for your HelloClient class as well as the HelloSvc class (#4).

To run your Java HelloClient class under the JVM you must modify the Java class path as you did in the compilation step, adding the current directory and the gen-java directory, where the HelloClient class and HelloSvr class files will be found (#5). Running the client produces the same result you saw with Python and C++.

Beyond running standard build tools in our respective languages, it didn't take much effort to produce your Apache Thrift server and the three clients. In short order, you've built a microservice that can handle requests from clients created in a wide range of languages. Now that you've seen how basic Apache Thrift programs are created, let's look at how Apache Thrift fits into the overall application integration landscape.

The Apache Thrift Tutorial

In addition to the code examples included with this text, the Apache Thrift source tree provides a tutorial. The tutorial is based on a central tutorial 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 tutorial directory below the root of the Apache Thrift source tree. Each language-specific tutorial is found in a subdirectory named for the language. A Makefile is provided to build the tutorial examples in languages that require compilation. The source tree also provides many tests throughout the tree, all of which provide useful examples.

```
/thrift/tutorial$ ls
as3         c_glib      cpp         csharp      d
dart        delphi      erl         gen-html    go
haxe        hs          java        js          netcore
nodejs      ocaml       perl        php         py.tornado
```

```
py.twisted py rb rs shared.thrift
tutorial.thrift
```

1.4 The communications toolkit landscape

SOAP, REST, Protocol Buffers, and Apache Avro are perhaps the technologies most often considered as alternatives to Apache Thrift, though many others exist. Each technology is unique and each has its place. The following sections provide a brief overview of the key players in the software communications landscape, followed by a summary of the features fielded by Apache Thrift and a discussion of where Apache Thrift fits in the milieu.

1.4.1 SOAP

Simple Object Access Protocol (SOAP) is a W3C recommendation (https://www.w3.org/TR/2007/REC-soap12-part1-20070427/) specifying a Service Oriented Architecture (SOA) style remote procedure call (RPC) system over HTTP. SOAP relies on XML for carrying its payload between the client and server and is typically deployed over HTTP, though other transports can also be used. Optimizations are available that attempt to reduce the burden of transmitting XML, and SOAP has versions that use JSON, among other offshoots. Related technologies, such as XML-RPC, operate on similar principles. Unlike RESTful services, which directly use HTTP headers, verbs, and status codes, SOAP and XML-RPC systems tunnel function calls through HTTP POST operations, missing out on most of the caching and system layering benefits found in RESTful services.

The key benefit of HTTP-friendly technologies is their broad interoperability. By transmitting standards-based text documents (XML, JSON, and others) over the ubiquitous HTTP protocol, almost any application or language can be engaged. Human-readable XML/JSON payloads also greatly simplify prototyping, testing, and debugging. On the downside, each language, vendor and, often, each company, provides their own scheme for generating stubs. You have no guarantees that code generated by different SOAP WSDL (Web Service Description Language) tools will collaborate.

SOAP was one of the principle technologies used during the evolution of Service Orientation and is still widely used in older systems. SOAP offers a number of WS-* standards established by the Oasis standards body, addressing authentication, transactions, and other concerns (https://www.oasis-open.org/standards). Few new SOAP services appear to be coming online, and most considering SOAP today find REST simpler, faster at scale, and more compelling as a public API solution.

1.4.2 REST

REST is an acronym for REpresentational State Transfer, a term coined by Dr. Roy Fielding in his 2000 dissertation (https://www.ics.uci.edu/~fielding/pubs/dissertation/rest_arch_style.htm). REST is the typical means for web browsers to retrieve content from web servers. RESTful web services use the

REST architectural style to leverage the infrastructure of the web. The well-understood and widely supported HTTP protocol gives REST-based services broad reach. REST-based services typically use the JSON format for payload transmission, making client/server requests human-readable and easy to work with.

RESTful services are unique in that their interfaces are based on resources accessed through URIs and manipulated through HTTP verbs, such as GET, PUT, POST, and DELETE. When done well, this is referred to as a Resource Oriented Architecture (ROA). ROAs produce significant benefits when scaling over the web. For example, standard web-based caching systems can cache resources acquired using the GET verb, firewalls can make more intelligent decisions about HTTP delivered traffic, and applications can leverage the wealth of technology associated with existing web server infrastructure. HTTP headers can negotiate payload formats, cache expirations, security features, and more. In-browser clients can leverage the native features of the browser, and the list goes on.

One concern with ROA is that monolithic applications are composed of modules that expose functions or methods internally. Module operations don't typically map naturally to resource based interfaces. This can make decomposing a monolith into RESTful microservices more work than decomposing the same code into RPC-based microservices.

When developers refer to APIs or services today, they're usually talking about REST APIs/services. The RESTful approach has become nearly ubiquitous when it comes to implementing public interfaces. The ecosystem is vast and the developer skills are widespread. REST, however, does have its drawbacks.

It's important to keep in mind that REST is an architectural style, not a standard or a technology framework. Two different teams might build the same REST service in different and incompatible ways. While this might be said of any solution, it's particularly true of REST due to the broad set of perspectives on how REST should be done and the several toolkits, schema mechanisms, and documentation systems in use. For example, the RESTful world offers at least three competing platforms for service definition and code generation: RAML, Swagger, and API Blueprint, though the more recent Swagger-based Open API Initiative (OAI) appears like it may unify the space.

Several communications models are not addressed by REST. REST is, by definition, a client/server architecture and, in practice, it's implemented over HTTP, a request/response-based protocol. REST doesn't address serialization concerns or support messaging or data streaming.

One of the most important issues with RESTful interfaces is their overhead in backend systems. The advent of HTTP/2 (https://http2.github.io/) does much to address the overhead associated with HTTP header and JSON text transmission; however, no amount of external optimization is likely to allow a REST service to perform at the level of a purpose-built binary solution such as Apache Thrift. In fact, Protocol Buffers and Thrift were created by Google and Facebook respectively to alleviate the performance issues associated with RESTful services in high-load server systems.

1.4.3 Protocol Buffers

Google Protocol Buffers (PB) (https://developers.google.com/protocol-buffers/) and Apache Thrift are similar in function, performance, and from a serialization and IDL standpoint. They were built by different companies (but by several of the same people) to do the same thing. Official Google Protocol Buffer language support is limited to Java, Python, Objective-C, C++, Go, Ruby, and C#. This is a moving target and support for new languages is added over time. Protocol Buffers are used by a large community of developers.

Google Protocol Buffers focuses on providing a monolithic integrated message serialization system through the main project. Several RPC style systems for Protocol Buffers are available in other projects, in particular, the HTTP/2-based gRPC (grpc.io). The gRPC system trades web platform integration through HTTP/2 for speed; Apache Thrift and Protocol Buffer TCP-based services typically run 4-6 times faster. Many developers feel the modular serialization and transport features of the Apache Thrift framework and the in tree language and server support provide an advantage. Others prefer the simple integrated serialization scheme offered by PB.

Another difference between the platforms is support for transmission of collections. Apache Thrift supports transmission of three common container types: lists, sets, and maps. Protocol Buffers supplies a repeating field feature rather than support for containers, producing similar capabilities through a lower-level construct. Newer versions of PB add map simulation with several restrictions. Protocol Buffers supports signed and unsigned integers, while Apache Thrift supports only signed integers. Apache Thrift, however, supports unions and other minor IDL features not found in Protocol Buffers.

Protocol Buffers are robust, well-documented, and backed by a large corporation, which contrasts with the community driven nature of Apache Thrift. This is evident most clearly in the quality of the documentation for the two projects, Google's being noticeably superior (and I'm being kind).

1.4.4 Apache Avro

Apache Avro (https://avro.apache.org/) is a serialization framework designed to package the serialization schema with the data serialized. This contrasts with Apache Thrift and Protocol Buffers, both of which describe the schema (data types and service interfaces) in IDL. Apache Avro interprets the schema on the fly while most other systems generate 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, but most practical use of Apache Avro has been focused on serializing objects to disk; Avro isn't used for RPC in the wild.

Apache Thrift versions

The Thrift framework was open sourced by Facebook in 2007 and became an Apache Software Foundation incubator project in 2008.

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
Project moved to top-level status in 2010
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 2014-11-16
0.9.3 released 2015-10-11
0.10.0 released 2017-01-06

1.4.5 Strengths of Apache Thrift

The strength of the Apache Thrift platform lies in the completeness of its package, its performance and flexibility, as well as the expressiveness of its IDL. Apache Thrift was created to provide cross-language capabilities comparable to REST but with dramatically improved performance and a significantly smaller footprint.

PERFORMANCE

To get a sense for the relative performance of several of the communications approaches described here, look at the test results in figure 1.5 (these tests are created and covered in detail in chapter 17). The chart displays the time required to make one million API calls to a single service implemented with several communications technologies. All of the servers were coded in Java and the same client, also coded in Java, was used in all cases, though the necessary bindings are used to call the service backend under test. Each bar shows the number of seconds the requests took to complete against a different implementation running on the same machine. The tests were performed in isolation over the local loopback on a system with no other activity. Multiple runs of each test were completed and no outliers were discovered. The sole service function accepts a string and returns a small struct. The service implementation is identical in all cases, performs no logic, and returns a static struct to highlight the service and serialization overhead.

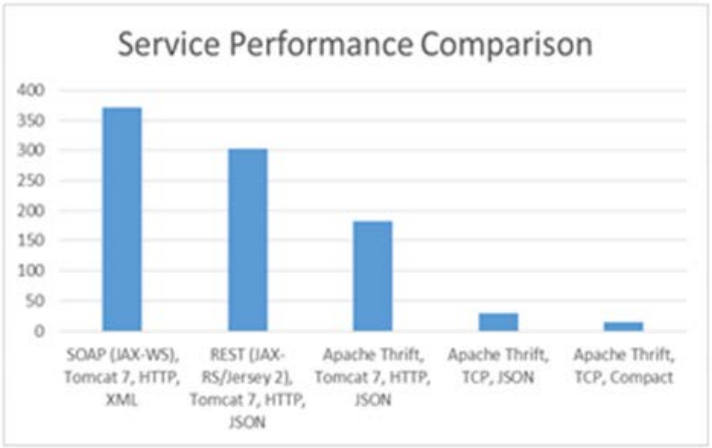


Figure 1.5 Time to complete 1 million service requests for various Java servers

The first bar shows the elapsed time for the service when implemented with SOAP. A standard Java SOAP service coded in JAX-WS, deployed on Tomcat 7, was used for the test. The serialization overhead associated with XML and the load incurred by Tomcat and HTTP make this the worst performer in the group, at more than 350 seconds.

The second bar shows the results of the same test but against a REST service created with Java and JAX-RS. Though the comparison normalizes as many variables as possible, REST-based services are defined with HTTP verbs and IRIs, not functions. The implementation here is a simple GET request (no caching), passing the input string as a query parameter and receiving the resultant struct in a JSON payload. This is noticeably faster than the SOAP example at about 300 seconds, largely due to the lack of a caller payload, improved serialization performance of JSON over XML, and the significantly smaller JSON reply payload.

The last three bars are Apache Thrift server cases. The first is as close to an apples-to-apples comparison with the REST example as can be had with Apache Thrift. An Apache Thrift server was created with the same one method service, packaged as a servlet, deployed on Tomcat, and configured to use the JSON protocol over an HTTP transport. The result is a significant improvement in performance. This is attributable to the serialization benefits produced by the purpose-built Apache Thrift client/server stubs, among other efficiencies.

Figure 1.6 Apache Thrift balances performance with reach and flexibility.

The real performance gains arrive when Tomcat and HTTP are left behind. The final two bars show the performance of compiled Apache Thrift servers running over TCP with JSON and Compact protocols respectively. Both are an order of magnitude faster and an order of magnitude smaller in memory.

While your mileage will vary with different languages, different levels of concurrency, different server shells, different services, and different frameworks, the previous example case provides a frame of reference and explains why many firms have moved large-scale backend services away from REST/SOAP and/or JSON serialization when under pressure for performance. Migrating to Apache Thrift from REST or SOAP could enable the same hardware to support 10 times the traffic.

Certain developers contemplate REST with payloads serialized using Protocol Buffers or Apache Thrift, however this doubles the toolkit burden and complexity, misses out on the significant benefits to be had by eliminating HTTP, and gives up the endearing “human readable payload” property typically associated with REST. It’s an altogether unsatisfying combination.

When it comes to performance, Apache Thrift offers a complete package with near REST-class interoperability, significantly improved performance, and the widest range of protocol and transport choices. See Figure 1.6.

REACH

Apache Thrift offers support in tree for a comprehensive set of programming languages but also an impressive range of platforms. Apache Thrift can be a good fit for embedded systems, offering support for Java’s Compact Profile and small footprint servers for C++ and other languages.

Apache Thrift is a natural fit for typical enterprise development environments, with support for Java/JVM and C#/CLR/.Net Core on Windows, Linux, and OSX. Apache Thrift is also a perfect fit for cloud-native systems, offering small footprint servers in many languages perfect for container packaging. See Figure 1.7.

Figure 1.7 Apache Thrift is an effective solution in embedded, enterprise, and web technology environments.

Apache Thrift integrates well with the world of the web also, including native support for languages such as JavaScript and Dart. Apache Thrift also offers HTTP, TLS, WebSocket, and JSON support in backend systems written in Node.js, C++, Java, C#, and more. Mobile solutions on IOS and Android are also easy to build and have support for Objective-C and Java.

1.4.6 Take away

You have many viable communications schemes to choose from today and they all have their place. As a default API option and particularly if you want broad accessibility over the public internet, REST may be your best choice. If speed is your priority, you can write your own native binary protocol or use something edgy like Cap'n Proto (https://capnproto.org). If you are principally serializing to disk, look at Apache Avro. If you want a solid, name-brand, high-speed serialization system, consider FlatBuffers (https://google.github.io/flatbuffers), or if you need RPC services as well, perhaps Protocol Buffers and GRPC will fit the bill.

However, if you want...

- *Servers and Serialization*—A complete serialization and service solution in tree
- *Modularity*—Pluggable serialization protocols and transports with a range of provided implementations
- *Performance*—Lightweight, scalable servers with fast and efficient serialization
- *Reach*—Support for an impressive range of languages, protocols, and platforms
- *Rich IDL*—Language-independent support for expressive type and service abstractions
- *Flexibility*—Integrated type and service evolution features
- *Community Driven Open Source*—Apache Software Foundation hosted and community managed

. . . in one package, then Apache Thrift belongs at the top of your consideration list. In the next chapter we'll look at the architecture of Apache Thrift and examine transports, protocols, and servers in more detail.

1.5 Summary

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

- Apache Thrift is a cross-language serialization and service implementation framework.
- Apache Thrift supports a wide array of languages and platforms.
- Apache Thrift makes it easy to build high performance services.
- Apache Thrift is a good fit for service-oriented and microservice architectures.
- Apache Thrift is an Interface Definition Language (IDL)-based framework.
- IDLs allow you to describe interfaces and generate code to support the interfaces automatically.
- IDLs allow you to describe types used in messaging, long-term storage, and service calls.
- 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 and disk and network transports, yet makes it easy to add additional transports.
- Apache Thrift supports interface evolution empowering CI/CD environments and Agile teams.