Learning Apache Thrift

This book will help you move beyond the basics of service-oriented systems through your first Apache Thrift-powered app. Then, progressing to more complex examples, it will provide you with tips for running large-scale applications in production environments.

You will learn how to assess when Apache Thrift is the best tool to use. To start with, you will run a simple example application, learning the framework’s structure along the way; you will quickly advance to more complex systems that will help you solve various real-life problems. Moreover, you will be able to add a communication layer to every application written in one of the popular programming languages, with support for various data types and error handling. Furthermore, you will learn how pre-eminent companies use Apache Thrift in their popular applications.

This book is a great starting point if you want to use one of the best tools available to develop cross-language applications in service-oriented architectures.

Who this book is written for

If you have some experience of developing applications in one or more languages supported by Apache Thrift (C++, Java, PHP, Python, Ruby, and others) and want to broaden your knowledge and skills in building cross-platform, scalable applications, then this book is for you.

What you will learn from this book

- Understand the need for cross-language services and the basics of Apache Thrift
- Learn how Apache Thrift works and what problems it solves
- Determine when to use Apache Thrift instead of other methods (REST API), and when not to use it
- Create and run an example application using Apache Thrift
- Use Apache Thrift in your applications that are written in different languages supported by Apache Thrift (PHP, Python, Ruby, Java, and C++)
- Handle exceptions and deal with errors
- Modify code in different languages
- Use Apache Thrift in the production environments of big applications

In this package, you will find:

- The author biography
- A preview chapter from the book, Chapter 1 'Introducing Apache Thrift'
- A synopsis of the book’s content
- More information on Learning Apache Thrift
About the Author

**Krzysztof Rakowski** has 13 years of professional experience in IT as a team leader, software developer and architect, and agile project manager. During the course of his career, he has helped major global brands establish their online presence using scalable, fault-tolerant, and high-performance systems. His broad experience comes from various industries, including interactive advertising, banking, retail, and e-commerce. He is a recognized expert, Zend Certified Engineer, and a Professional Scrum Master.

Currently, Krzysztof works for the largest online shop in central and eastern Europe—where he is responsible for supervising teams of software engineers and project managers who pair the smartest IT solutions with the best customer experience.

He enjoys sharing his knowledge through articles and presentations. He occasionally writes about his side projects on his website at www.rakowski.pro.

In his free time, Krzysztof likes to travel around the world with his wife, go snowboarding, or read a good book.
In 2007, Facebook's engineers needed to integrate the various applications powering their website. As their engineering culture encouraged selecting the best tools for a task without imposing strict rules regarding the choice of technology, their applications were written in a wide spectrum of different programming languages, which were considered the best for the given task.

Looking for the best solution to fulfill their needs, the engineers reviewed lots of different frameworks that were already available on the market. None of them was deemed sufficient in terms of performance or flexibility. They made a decision to develop their own solution, which became a standard to integrate all the services on Facebook.

As they considered their solution to be exceeding the current standards of the market, they released their code to the open source community, passing the task of maintaining their work on the project to the Apache Software Foundation. Since then, Apache Thrift has been developed by a large group of volunteers.

Now you can use Apache Thrift as a tool to expose your own services that are written in different languages and make your applications communicate with each other. Regardless of whether you intend to work on a small-scale application or huge enterprise, Apache Thrift may be one of the best tools for you.

In *Learning Apache Thrift*, you will find an introduction to various concepts of the services around you and some service-oriented architecture (SOA). Then you will learn how to use Apache Thrift in various projects. We will discuss advanced concepts too to see how the giants of the industry use this framework, and you will get some solid advice and much needed inspiration.
What this book covers

Chapter 1, Introducing Apache Thrift, gives you basic information about the environment where services are needed. You will learn about the history of Apache Thrift and its position in the market. This chapter provides some solid understanding of the context in which Apache Thrift exists.

Chapter 2, Installing and Running Apache Thrift, provides you with a quick tutorial that will allow you to have Apache Thrift up and running on your machine in no time. Instructions for Linux (Debian and CentOS), Windows, and Mac OS X are included.

Chapter 3, Running Your First Apache Thrift Service and Client, gives you the ability to see Apache Thrift in action. Simple instructions will get you through the process of setting up a server and client that run in two different programming languages (PHP and Python) and communicate with each other.

Chapter 4, Understanding How Apache Thrift Works, provides you with real knowledge of the framework's internals. You will learn about its components, network stacks, data types, interface description language (IDL), and the programming languages that are supported. You will also find out about its limitations and how to deal with them. This chapter is essential to understand the concept of "under the hood", and how to design your own Apache Thrift-supported services.

Chapter 5, Generating and Running Code in Different Languages, provides you with a toolbox of essential information about different popular programming languages and how you can use them with Apache Thrift. You may read it from the beginning to the end or just focus on those languages that interest you. The same example is used for every language, so you can easily compare the server's and client's implementation for each of them.

Chapter 6, Handling Errors in Apache Thrift, gives you information on how to deal with undesirable situations that may occur when you run your service or client. Handling errors is an important part of any programming project, and is especially essential when dealing with cross-platform applications where errors occur frequently due to the nature of the distributed architecture.

Chapter 7, An Example Client-Server Application, gathers knowledge from the whole book into one example client-server application. You will build the code step by step. The example touches every aspect of Apache Thrift and is a bit more complicated than what you have done until now. Three different languages will be used (PHP, Python, and Ruby).
Chapter 8, Advanced Usage of Apache Thrift, inspires you to further expand your Apache Thrift skills. You will learn how big companies use this framework, how to run your applications in production, and how to address security, performance, and scalability issues. You will also be given access to other interesting Apache Thrift-related projects.
Introducing Apache Thrift

There is a milestone in the life of every sufficiently large application that marks the point when it is too big to be maintained as a monolith. For some systems, it is in their blueprints from the very beginning, while for others, it comes as a growth induced necessity and brings along the need for massive rebuild.

Apache Thrift is one of the tools that assist in building scalable, distributed systems, spanning across different platforms and languages. Originally developed for internal use by Facebook, now it is an open source software project backed by the Apache Foundation. It is characterized by a wide range of supported languages, flexibility, and performance.

In this chapter, you will learn about the scenarios where using Apache Thrift may be necessary. You will also get familiar with its basic properties and how it is compared to other similar frameworks. It is essential to know the big picture to be able to select the best tool for your job.

Let’s see how you can put Apache Thrift to good use!
Distributed systems and their services

Imagine typical web applications that you use every day, such as search engines, messaging platforms, or social networks. Under one web address, they deliver different services. For example, a social network delivers people search, messaging, and users’ profile pages. While you access them by one user interface—a web page written in HTML and JavaScript—what you see in your browser is only a gateway. Your request to message a friend is being relayed by the underlying application to the messaging service—an application which is specifically designed to deal with exchange of messages between the social network’s users.

Service-oriented architecture

Messaging service, which we use as an example here, may be written in a completely different programming language than web application. It is a design decision. The system architect may decide that interface of your social network; the web pages that you see every time you log in will be easier to manage and maintain when they are written in, let's say, PHP or Ruby on Rails. However, messaging systems may be written in Python as the architect may decide that this language offers better libraries for this task. On the other hand, search engines or other tools that need superb performance are often written in C++. There may be also some internal corporate applications in Java or C#.
Those applications, of course, need to communicate with each other. But how to do that? There is a concept in software design called **service-oriented architecture** (SOA). We just discussed the first part of this principle. It focuses on creating applications around distinct tasks. If every task is performed by a different application, there is a need for some means of communication between them.

To achieve this goal, applications expose services that are used by other applications. Typically, they are accessible over some medium, that is, an internal network or the Internet. They are self-contained and autonomous, which means they are independent of other services and are able to deliver complete response when queried. They should also be well documented so that any developer can use them.

**Distributed systems**

When—as in our example of social network—we have a system that consists of many autonomous services, we call such systems **distributed systems**. Depending on the scale, business needs, or technical constraints, the systems may be spread over lots of computers in a local network, the Internet, or just on a single machine. Benefitting from the SOA principles, you may run and test on your desktop computer distributed system of the same logical architecture, which will be then used on hundreds of servers in the production environment.

There are many advantages of SOA in distributed systems over monolithic applications. Let's discuss some of them.

**Maintainability**

The greatest advantage of distributed systems in SOA is their maintainability, which means ease of performing all the tasks related to the caretaking of the software. If the system consists of many applications, each dedicated to one task or type of tasks instead of one big monolith, some of the actions can be performed a lot easier:

- You can select tools (that is, programming languages, libraries, and services) that are best for a given task. You can use different toolsets for search engine, message queues, or data-intensive calculations.
- Instead of having all the developers working on one application (that means one code base), you can split the team to work on many applications separately. You can even outsource some of the work to external teams or companies. This way, they won't get in each other's way. Smaller teams are more agile and yield better results.
- Communication between the different components of the system is narrowed to only one specified interface, which is easier to comprehend, monitor, and debug than lots of convoluted classes and methods.
Introducing Apache Thrift

- It is easier to respond to failures and fix bugs. Let's say there's some bug introduced that causes whole application to crash. In distributed systems, only one service may be down, while the whole system is operational. System operators or developers are able to replace the service with the stable version and do some tests to identify the bug or perform other actions without affecting the rest of the system.

- Introducing changes is a lot easier too. In the common workflow, if a new version of a service is to be deployed, it can be run as a separate instance with the old version simultaneously. System operators can switch the client application from the old to the new service and see whether everything performs correctly. If it does, the old service is turned off; otherwise, it is easy to switch back to the old service and fix the new one. It is even easier in the cloud environments.

Scalability

Many systems are required to perform well under a high load. It is not only the domain of web applications, but it is best pictured here: popular websites receive hundreds of millions of page views per day, which constitutes a high traffic load. To withstand such increasing stress, systems need to scale. The most obvious way, known by every computer user, is to add RAM or switch to a better CPU if applications don't run smoothly. But there is a limit to such scaling (called vertical scaling). You don't expect Google to run on a single powerful computer, do you?

The other type of scalability is horizontal scaling, which means adding more computers (called nodes) to the system. For example, our imaginary social network system may consist of several web application nodes, a few database nodes, and also some user search nodes. In properly designed systems, operators can add or remove nodes depending on the expected load and other circumstances. More sophisticated systems can even scale themselves, starting or stopping nodes in the cloud automatically, based on the traffic analysis.

SOA allows multiple nodes of the same function to be accessible to the clients. As services are self-contained, independent of the state of other services, and documented, developers can prepare their software without much care if they will be dealing with one or hundred nodes. In most scenarios, traffic to the services is managed by software or hardware load balancers, making it completely invisible for the client.
Testability
Another advantage of distributed systems is the easiness of testing them and finding and fixing bugs. Independence of services means that they can be tested in isolation from the whole system. Only a particular service's operation is being tested without any influence from other components. Because services should be well documented, it is easy to predict the desired output for a given input. If bugs are found, they can be evaluated and fixed without the need to consider them in the scope of whole system.

An introduction to Apache Thrift
You probably know Facebook, the popular social network. A small website started in 2004 as a funny side project by a Harvard student, Mark Zuckerberg, gained huge popularity, having more and more users. In its early years it faced rapid growth in terms of traffic, system, and network structure. Their engineering culture allowed choosing any solution that was deemed optimal for a given task without any constraints or standards. This led to a situation when they had lots of different services, but no reliable way to connect them together. Describing Apache Thrift, Facebook's engineers stated in the white paper (you can download it from https://thrift.apache.org/static/files/thrift-20070401.pdf):

"(...) we were presented with the challenge of building a transparent, high-performance bridge across many programming languages."

They tested solutions available in the market and came to the conclusion that none of them fulfilled the requirements of high performance, flexibility, and simplicity. The result of their work was Thrift—a piece of software that was later open sourced and handed over to the Apache Foundation.

Apache Thrift's simplicity comes from the fact that the code for different programming languages is generated automatically from a single file written in the interface definition language (IDL). In other similar solutions, data has to be prepared before it is transferred to meet the limitations of the method of transport—not all structures are easily transferred. In most cases, simple data types such as strings are integers and transferrable. Due to this, a developer has to translate every structure more complex than that to the text form in a process called serialization. This has to be done on both ends (deserialization being the reverse process), which needs extra work, testing, and debugging. In the case of Apache Thrift, the developer can use data types native to their programming language of choice using the methods dedicated to this language. All serialization and deserialization is made by the Apache Thrift itself and is not visible to the developer. This architecture of the solution allows programmers to focus on working on the actual services, and not having to care about how the data is going to be transferred from one application to another.
Introducing Apache Thrift

Let's have a quick glance at the pillars of Apache Thrift. Some of the topics will be covered in much more detail in Chapter 4, Understanding How Apache Thrift Works, so here are just the basics that you will need to understand our first code examples.

Supported programming languages
Before starting any work with Apache Thrift, you should probably check whether it supports the programming language that you use. Of course, there is a great chance that it does – most of the popular languages are supported. The complete list for version 0.9.3 is as follows:

- ActionScript 3
- C++
- C#
- D
- Delphi
- Erlang
- Haskell
- Java
- JavaScript
- Node.js
- Objective-C/Cocoa
- OCaml
- Perl
- PHP
- Python
- Ruby
- Smalltalk

Note that Apache Thrift is still in the pre-1.0 version, so some of the languages may be not fully supported. It is best to check on the Apache Thrift website (https://thrift.apache.org/docs/features), in the source code, or try to learn the current status of support for your favorite programming language yourself.
If your language of choice is on the list (especially if it is a popular one), you can be sure that you will be able to generate all the code necessary to work with Apache Thrift.

**Data types**

One of the basic features of every programming language is their data types. Although the basic ones may be very similar, that is, integer or string, it may not be that easy for the rest of them. Some of the languages (for example, C++) are statically typed. This means that the type of the variable has to be known at the compile time. Thus, it has to be defined in the source code when the program is written. After that, the variable can be of only this type. For example, consider the following line from C++:

```cpp
int x = 42;
```

It initializes the variable `x`, which is an integer. This variable has to stay an integer through the execution of the program. If later on you would like to assign a value of some other type, it will produce an error as soon as you compile your program.

Let's take a look at the following example:

```cpp
int main()
{
    int x = 42;
    // this line will produce compilation error
    x = "forty two";
    return 0;
}
```

If you try to compile this simple code, you will end up with the following compile error:

```bash
$ g++ -o example example.cpp
example.cpp: In function 'int main()':
example.cpp:4:6: error: invalid conversion from 'const char*' to 'int'
[-fpermissive]
    x = "forty two";
^
```

Other languages are dynamically typed, that is, the type of the variable is checked in the runtime, but in the source code it might be anything, any time. Consider this example from PHP:

```php
if (rand(0,1) == 1) {
    $x = 42;
} else {
```

```bash
```
Introducing Apache Thrift

```php
$x = "forty two";
}
var_dump($x); // var_dump() function prints type of specified
// variable and its value
```

Depending on the random outcome of the condition, the value of the variable may be either integer or string. Let's take a look at the following example:

```
$ php -f example.php
```

The result of running this program would be either `string(9) "forty-two"` or `int(42)`.

As you can see, both values are permitted as PHP interpreter changes the type of the variable during the runtime.

Programming language allows that and, moreover, later on, you can assign values of different types to the same variable.

Without Apache Thrift, developer would have to serialize the variables. It means that before the variables are transferred, they should be mapped to the most basic data types that are understood by every programming language (most probably, integers and strings of characters). After the transmission, those serialized variables have to be translated back to the structures available in the programming language at the receiving end.

Apache Thrift does all that dirty work for the developer. It provides its own data types that are then mapped to the ones native to the given programming language, thereby allowing the developer to focus on creating the application, not the communication interface.

## Transports

Transports are a part of Apache Thrift's network stack. They allow you to transmit data over different channels, that is, HTTP protocol, sockets, or files. Decoupling the transport layer lets you to easily choose the transport that best fits your solution without many changes in the code.

The choice of transport should be dictated by the architecture of your solution.
Protocols

Protocols prepare data to be transmitted over transports. The name of the process is called serialization (when sending data) and deserialization (when receiving data). There are different protocols that can be used: JSON, binary, plain text, and so on. It means that depending on what data you want to transfer, you can use different methods of serialization. For example, if you expect to transmit images or other binary data, choosing the binary protocol is the best option as there would be almost zero overhead. If you chose JSON for this purpose, binary data would be converted to text, thereby increasing the payload by a third or more.

The choice of protocol should be dictated by the data you wish to transfer using Apache Thrift.

Versioning

Versioning is an approach for managing changes in the service's API (and in the software in general). As software is being developed, it changes. Sometimes the changes are miniscule, and sometimes great. They are often manifested by modification of the methods or parameters exposed by the API.

When developing client and server software, you shouldn't assume that clients will be updated to the newest version instantly. It is not possible, even if you have total control of the environment. It is also wise to allow the older versions of the client to work with the newer versions of the server.

Changes in the APIs, libraries, and other externally available components pose a big challenge for the developers, leading to problems often referred to as dependency hell – when different applications are compatible with different versions of the same library or API, leading to difficulties with managing those dependencies.

To alleviate this inconvenience, most of the software developers adopt a convention of marking the version of the application with decimal numbers, according to the template, MAJOR.MINOR.PATCH, where PATCH means miniscule changes (that is, fixing some bugs), MINOR is a larger change but backward-compatible with the previous versions, and MAJOR means a major release that might break the compatibility with the previous versions of the software.
Introducing Apache Thrift

Apache Thrift's feature is soft versioning. It means that there are no formal requirements as to how the changes between the subsequent versions should be handled or announced. However, it delivers a set of tools that allows users to easily keep backward compatibility with the new versions of the service. It is achieved by the following properties:

- The method's arguments are numbered. You can add or remove them. As long as the same number is not reused, the new versions of methods may function without removed arguments. Those numbers shouldn't be changed for any existing argument.
- You can set default values for the arguments, so if the older version of the client has a method without a new variable, the service doesn't receive any value for such an argument and the default value is set. This is useful when you want to add some fields.
- While manipulation with fields is relatively easy, you shouldn't rename methods or services. This makes them unavailable for the older clients.

Security

Security is essential to every service. Although you definitely need to take extra care when exposing services to the Internet, it is also important when they are available in private networks.

Apache Thrift allows you to use TSSLTransportFactory to utilize RSA key pairs, providing security for the connection.

Another way of securing your Apache Thrift connection (although a little bit more complicated) is tunneling it over SSH.

We will discuss this in the detail in Chapter 8, Advanced Usage of Apache Thrift.

Interface description language

Apache Thrift's core feature is its own IDL, one that shapes its simplicity and usability. It will be familiar at first sight to anyone who has programmed in contemporary programming languages. Using IDL, you are able to define the service and all the variables that it uses in one file. It is an unambiguous description of what the service will look similar to without going into the implementation details.

Let's consider a very simple service, which allows you to add two integers:

```python
namespace py thrift.example1
namespace php thrift.example1
```
service AddService {
    i32 add(1: i32 a, 2: i32 b),
}

This example code defines AddService service, which contains the add method. This method takes two 32-bit signed integers (i32) as parameters and also returns such an integer as a result. We will want to have the code generated for Python and PHP languages, but of course Apache Thrift is able to do it for a far greater spectrum of languages.

Now the Thrift's magic begins; if you save this code to the file (let's say, example1.thrift) and run the following commands, you will get the code of client and server for this service in desired languages (Python and PHP in this example) in the newly-created folders, gen-py and gen-php:

```
$ thrift --gen py example1.thrift
$ thrift --gen php example1.thrift
```

In the simplest solution, it is enough to fill the code of the add method, and voilà, you have a fully-functional client and server.

This example is, of course, oversimplified, but shows the major advantage of Apache Thrift—the ability to define in one place and then instantly generate services and the corresponding client code without the need of writing code in every language from scratch. It is a great tool not only for final solutions, but also for rapid prototyping for different programming languages.

To see how much work Apache Thrift just spared you, examine the generated files that are saved in the gen-py and gen-php folders.

IDL is a very powerful tool. It has a lot of options and gives you a great deal of flexibility. We will discuss it in greater detail in Chapter 4, Understanding How Apache Thrift Works.

**Apache Thrift and others**

Until now, you may have come to the conclusion that Apache Thrift is the best solution for all your needs when dealing with distributed systems. Surprisingly, it is not always true. In this section, we will review similar tools so that you are able to understand how Apache Thrift compares to them and when to use which tool.
Introducing Apache Thrift

Custom protocols

Frequently, inventing your own custom protocol is the first idea that comes to a developer's mind when he/she needs to transfer data between two applications. Very often, it works surprisingly well in small solutions, which are not expected to scale or be modified frequently.

Examples of such solutions are popular in web applications. Creating your own custom protocol is as simple as generating output with some text: just plain or formatted according to JSON or XML specification, and serving it through HTTP. On the client side, we need to connect to this service, get the content, and parse it.

To imagine such a solution better, consider a very simple example of a service adding two numbers. The request may be the following GET call:

```plaintext
GET /add?number1=30&number2=12
```

The response in the JSON format may be the following:

```json
{"result":42}
```

Unfortunately, the only advantage of such solutions is that they are quick and easy to implement, both on the server- and client-side, on a small scale. Besides that, there are some disadvantages:

- Text-based protocols have significant overhead. This is especially true for XML, which encapsulates everything with lots of tags.

- They transfer binary data (that is, images), adding additional overhead to the payload. As those protocols are text-based, binary data has to be converted to text. One of the popular techniques is Base64, which encodes the message byte by byte into a printable text character. The outcome of such an operation is that the string that is ready to be transferred is around 37% larger than the original binary data. There is also extra processing required on both client's and server's end.

- There are really no standards for such protocols; everything has to be invented by the developer. It poses not only difficulty when designing such a service, but also is a complication when the client's applications have to be maintained; for every service, there need to be custom tools prepared. And no standards means that debugging is a lot more difficult.

- Maintenance is another problem with such protocols. When there is a change needed, both server and client code needs to be modified separately and deployed at the same time. There is no way to modify the code once and have it working on both client and server.
Of course, the spectrum of possibilities when designing custom protocols is much wider than those examples that are typical for web applications. One can design their own binary protocols working on sockets, files, queues, or another medium. This gets rid of some of the disadvantages of text-based protocols, but still leaves lots of other problems to deal with.

**XML-RPC and JSON-RPC**

XML-RPC is one of the early remote procedure call (RPC) protocols, which uses XML-encoded messages transferred over HTTP. JSON-RPC is its much younger cousin, which is based on the same principle, but uses JSON instead of XML.

Both protocols allow you to call remote services with handful of data types in the relevant format. The exchanged messages are plain XML or JSON without any overhead.

Here is an example of a typical XML-RPC request:

```xml
<?xml version="1.0"?>
<methodCall>
  <methodName>add</methodName>
  <params>
    <param>
      <value>
        <int>30</int>
      </value>
    </param>
    <param>
      <value>
        <int>12</int>
      </value>
    </param>
  </params>
</methodCall>
```

And, the corresponding response is:

```xml
<?xml version="1.0"?>
<methodResponse>
  <params>
    <param>
      <value>
        <int>42</int>
      </value>
    </param>
  </params>
</methodResponse>
```
Introducing Apache Thrift

JSON-RPC request is much more verbose:

```
{"method": "add", "params": [30, 12], "id": 1}
```

The service will return the following response:

```
{"result": 42, "error": null, "id": 1}
```

The simplicity of both of these protocols comes at a price. While they may be easily implemented, they share disadvantages of custom protocols, such as lack of standards and need for maintenance of both server and client codes, and they may not be best suited for transferring binary data.

**SOAP and WSDL**

**Simple Object Access Protocol (SOAP)** is a solution for some problems with customarily designed protocols, which evolved from XML-RPC. It is used mainly for web services (over HTTP) to exchange structured information between them and clients.

SOAP is a protocol based on XML. It is rather complicated with several layers of specification. The messages are structured according to this specification.

Every SOAP message consists of the following elements:

- **Envelope**: This is the root element of the message that identifies the message as SOAP and defines its structure.
- **Header**: This is an optional field that may contain extra application-specific control information for identifying the message.
- **Body**: This contains the actual payload of the message (call or response).
- **Fault**: This is an optional element that is used to pass information about errors. It contains error code, description, and other application-specific information.

Web services over the Internet are commonly provided with SOAP as a method of calling operations described in the **Web Services Description Language (WSDL)** file. In this file, the available messages are described in the XML schema form.

Due to SOAP’s standardization it is easy to debug, and there are many tools that help to do that. It is enough to parse the WSDL file to be able to communicate with the given web service.

Unfortunately, SOAP still has disadvantages discussed previously: a large overhead connected to XML processing and the need to encode binary data into text form.
RESTful APIs

WSDL-based web services using SOAP were considered cumbersome and complex, so **Representational State Transfer (REST)** was introduced as a simpler alternative. Web services that are developed in accordance with REST’s architecture constraints are called **RESTful APIs**.

Features of REST can be perceived as a mix of two previously discussed topics: custom protocols and SOAP.

RESTful APIs are simpler and a lot lighter than SOAP. They make use of HTTP methods to manipulate the data (collections of elements):

- **GET**: This is used to retrieve information about some collection or its elements
- **PUT**: This is used to create or replace the collection or element
- **POST**: This is used to create a new element in the collection
- **DELETE**: This is used to delete entire collection or a specific element

Every collection or its element has its own, unique **Universal Resource Identifier (URI)**.

The advantages of RESTful APIs are their simplicity and efficiency. They are also scalable and cacheable.

On the side of disadvantages, there is a lack of standardization (each service's message and response format may be different), no built-in error handling, and no standardized authentication mechanisms.

**CORBA**

**Common Object Request Broker Architecture (CORBA)**, [http://www.corba.org/](http://www.corba.org/), dates back to 1991, and is the oldest of the standards presented in this chapter. However, its concepts are quite similar to Apache Thrift (for example, it uses its own IDL).

It is considered a bit cumbersome; instead of using a language's native code, a developer needs to use a CORBA-specific one. It's hard to install and heavy to run. There are different implementations and they are inconsistent.

**Apache Avro**

Apache Avro ([https://avro.apache.org/](https://avro.apache.org/)) is another remote procedure call and data serialization framework developed with the support of the Apache Foundation. It was developed as a tool for the Apache Hadoop framework.
Introducing Apache Thrift

Lots of similarities to Apache Thrift include describing the interface with IDL, support for many programming languages (Java, C, C++, C#, Scala, Python, and Ruby), and a compact, fast binary format.

The main difference is that Avro's code doesn't have to be generated when the service is defined and later on, when it changes. It could be, for statically typed languages, but for dynamically typed languages, it is not necessary. It is possible because Avro uses the dynamic schema, which accompanies data when it is being transferred.

As a disadvantage in comparison with Apache Thrift, Apache Avro doesn't offer such a wide selection of serialization formats (protocols, in Thrift's terminology) and transports.

Protocol Buffers
Protocol Buffers are an older brother of Apache Thrift, and they share lots of similarities. They were developed as an internal proprietary software in Google and are used in most of the inter-machine communication. Since their release to open source in 2008, they have gained support not only for officially implemented languages (C++, Java, and Python), but also a lot more (JavaScript, Go, PHP, Ruby, Perl, and Scala).

Apart from IDL syntax and implementation details, Protocol Buffers differ from Apache Thrift in that they have less language support, different base types, a lack of constants and containers, and no built-in exception handling. In the open source version, there is also no RPC implementation for services (you need to implement it yourself).

On the other hand, Protocol Buffers are a little bit faster than Apache Thrift and their objects are smaller. Also the documentation and availability of tutorial is considered better and more complete.

When to choose Apache Thrift
When designing and developing applications that have to communicate with each other, one may go through the whole evolution process involving the solutions presented in the previous section. Many services start as a very limited tool, which works quite well with some simple custom protocol. But the data that needs to be transferred may become more and more complicated than the need for some format, such as JSON or XML appears JSON-RPC or XML-RPC may be then used.
As the service is growing and is exposed to more external applications, the need to standardize the architecture and proper documentation arises. In such cases, using web services based on SOAP and WSDL seems to be a proper idea. If your application's goal is to operate on collections of elements, RESTful API may be a good solution.

But there are situations where one needs to transfer binary data and provide flexibility for changing the definition of the services along with support for different platforms and languages; all this in an environment where high performance is crucial. In these cases, serialization and remote procedure call for frameworks such as Apache Avro, Protocol Buffers, and Apache Thrift. From these three, the last one offers the widest selection of serialization formats, and transports along with remote procedure call implementation.

**Summary**

Distributed systems, SOA, SOAP, WSDL, XML, and JSON, are some of the popular buzzwords that are frequently encountered by developers interested in creating applications that talk to each other. It is often hard to comprehend how these theoretical concepts can be used to accomplish the goal.

In this chapter, we learned what these distributed systems are and Apache Thrift's role in achieving communication between different services. We also discussed its position among similar solutions and their advantages and disadvantages.

In the upcoming chapters, we will install Apache Thrift, generate and run our simple services, and discuss the features in great detail. Having this knowledge, we will advance to prepare our own client-server application using Apache Thrift.
Where to buy this book

You can buy Learning Apache Thrift from the Packt Publishing website.

Alternatively, you can buy the book from Amazon, BN.com, Computer Manuals and most internet book retailers.

Click here for ordering and shipping details.