**Apache Thrift's architecture**

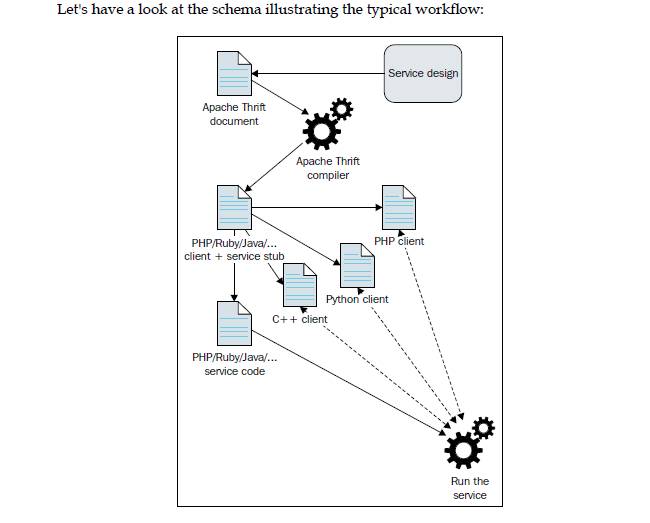
Apache Thrift's architecture can be viewed from two angles. The first approach shows the tool that is used by the developer—the Thrift compiler—along with the set of instructions that determine what to do in order to run the services.

The second way is to get to know the internal architecture and the purpose of protocols, transports, services, and processors that the developer will use, in order to know how the tools are built and what are the ways of their operation.

As in every craft, you can build things knowing the basic operation of the tool that you will work with; not every developer needs to have a deep knowledge of Apache Thrift's internals to do his job. However, to fully understand how it works and design the scalable and performant services and system architectures, this knowledge is essential.

**Going about using the tool**

One of Apache Thrift's advantages is that it is rather simple to use and implement in the programs written in many different programming languages. Developers are able to apply a simple instruction set in order to have the services and clients up and running in no time.



In this section, we will discuss the following steps:

1. Designing the services
2. Preparing interface description
3. Generating the service and client libraries
4. Implementing services and clients
5. Running the server and clients

**Designing the services**

First, the most important step is independent of the software; it is to design your system and services. In a typical scenario, you would need to assess the applications that are already in place or be developed. It has to be taken into consideration that what kind of interaction between them is needed. Most commonly the applications would need to do one of the two things, that is, either receive or deliver some data or perform some operations remotely.

It is possible that there will be one or many such services exposed in your system.

Frequently in the typical production setup, some applications may be accessible to other applications, others not. Sometimes, the communication may be permitted only in one direction. This is often due to security restrictions, company policy, or network topology. Take this into consideration when designing services and making the decision about which application should expose them and which should act as a client.

**Preparing the interface description**

After you reviewed your needs and designed your services, you need to prepare the Apache Thrift document, which is written in IDL and has the .thrift extension. You will describe each of the services. To do this, first you need to define variables and constants that you will use. Service description offers the information about the functions, their parameters, the kind of value they return, and if they can throw an exception. In one file, there can be multiple services described.

You learned about how the exemplary Apache Thrift document looks like in the previous chapter. More on the syntax of this document is in the upcoming sections.

**Generating service and client libraries**

When your interface description document is ready, you can run the thrift command with the required parameters on your IDL document. This command takes the file, processes it and generates—in accordance with different specifications for each of the programming languages—the files containing description of the services and related variables' types that will be used by your service and clients. These files extend the classes delivered by the Apache Thrift library.

If you will go through the generated files, you will see that every element from your IDL document was translated to the programming language of your choice. We will examine these files in detail in *Chapter 5*, *Generating and Running Code in Different Languages*.

**Implementing services and clients**

This step is sometimes perceived as the hardest part, as the developer is required to prepare the server's and client's code from scratch, and the documentation is sometimes sparse on this subject.

The first step is to implement the services and wrap them around in the server's code. This code will be based heavily on the Apache Thrift library and the generated classes. You need to choose the desired processor, transport, and protocol (more on this in later sections) and add the actual functions' code. In real life, this code will rely hugely on the different parts of your application. However, for the sake of brevity, the client and server code is self-contained (with the exception of any language's standard libraries) throughout this book.

The second step is to implement the clients. As with the server, you are required to do it by yourself from scratch. This time, however, it is easier: you need to pick the same type of processor, transport, and protocol as you choose for the server and instantiate the service class generated by the compiler.

In most cases, you will implement the services only in one programming language as a part of the application delivering those services. The client's code, on the other hand, may be implemented in many languages depending on the different applications that need to access your service.

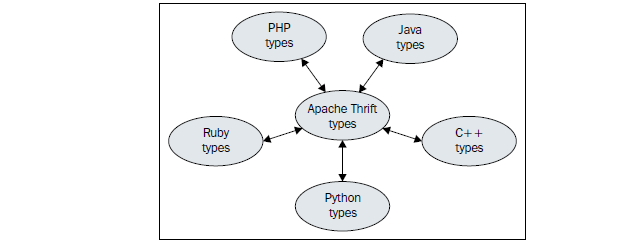
**Running server and clients**

You run your server's and clients' code in a way typical for the programming language of your choice. Remember to start the server first and then run the clients.

As you may see, Apache Thrift removes the burden of preparing the communication from scratch; you just need to fill in the gaps that are prepared for you to have a fully functional, cross-language communication.

**Apache Thrift's type system and IDL**

Apache Thrift offers its own type system, which is designed to allow the developers to use the variable types native to their language of choice. Then, Thrift's libraries take care of translating the types between different languages.



Apache Thrift creators divided its types into a few categories:

**Basic**: These are the simplest types present in virtually every programming language.

**Special**: These are those which don't fit into the basic category (currently, it is only one type).

**Structs**: These are the equivalent of structs or classes from popular programming languages (with some limitations).

**Container**: These are equivalent to commonly used container types in most of the programming languages.

**Exceptions**: These are used to handle the errors.

**Services**: These are core concept in Apache Thrift. They gather all the other types mentioned earlier to describe the procedures that can be called remotely. Exposing services is the main purpose of Apache Thrift.

IDL is a way of defining Apache Thrift types to create the services.

**Basic types**

Basic variable types reflect those present in every programming language. Those are basic numeric, string and binary types:

1. bool: Boolean value—true or false
2. byte: Byte or 8-bit signed integer
3. i16: 16-bit signed integer
4. i32: 32-bit signed integer
5. i64: 64-bit signed integer
6. double: 64-bit double precision signed floating point number
7. string: UTF-8 encoded text

Variable declarations don't appear by themselves in the interface description file; they are present in the service declaration, as a function attribute or in typedef statements.

In a function declaration, the type of the variable precedes its name, for example:

i32 add(1: i32number1, 2: i32number2)

The integer type names seem a little bit less readable in comparison to other languages, so the developers often tend to re-declare their name to create some more familiar name using the typedef statement:

typedefi32int

**Structs**

A struct is an object with a set of strongly typed fields, which is used to encapsulate similar variables. Some readers may immediately notice the similarity to C's structs.

Let's have a look at a simple struct in a very popular example:

struct Person {

1: required string first\_name,

2: optional string middle\_name,

3: required string last\_name,

4: i32 age = 0

}

As with many other syntax elements in Apache Thrift's IDL, the attributes are numbered. It is needed to preserve the compatibility between different versions of the services, you should keep the number with the corresponding attribute once and for all. More on the subject of versioning will be covered in *Chapter 8*, *Advanced Usage of Apache Thrift*. This identifier is positive, should be unique, but the numbers don't have to be consecutive.

You may mark the field as being required or optional. As the names state, the former will enforce you to a set given value, while the latter (which is default) will allow you to skip this field. Setting the required field may run you into trouble if you decide later on to remove this requirement (or the field altogether), but still have some instances of your application running the old version of the service.

You may also set the default value of the field, which will be used if you don't explicitly set this field's value.

One of the features of struct is that it can contain other struct:

struct Employee {

1: i32id,

2: Person person,

3: string position

}

This way, you can construct more complex structures and since struct doesn't support inheritance, this syntax provides some kind of substitute.

**Unions**

Unions in Apache Thrift are similar to those in C and C++. They provide a set of possible fields of different types, and only one of them can be used. The syntax is the same as that for struct with the difference of using the union keyword instead of struct. Union's fields can't be set as required as any of the types is equally valid.

Let's look at the example definition of union:

unionMyNumber {

1: i32number\_int,

2: string number\_word

}

We defined MyNumber union, which can have either a 32-bit unsigned integer or string as a value, not both. So this function may take either of the variable types as a parameter:

void save\_number(1: MyNumber number)

You can use any valid Apache Thrift data type in union.

Note that not all of the programming language support union. In such situations, there is a fallback to struct.

**Containers**

Apache Thrift's containers are mapped to the commonly available container types in the popular programming languages. They are strongly typed, which means that the types of keys and values are predefined. This may come as an inconvenience for programmers used to weakly typed languages, but is necessary to offer interoperability with strongly typed ones.

There are three types of containers with the syntax borrowed from the Java generics style.

**list**

list contains elements of a specified type. It is ordered and may contain duplicates. The syntax for list is:

list<type>

In the place of the type keyword, there should be an identifier of the type of the elements of list. For example, list<i32> is a list of 32-bit signed integers.

list is mapped to native lists or arrays in most of the programming languages, STL vector in C++, and ArrayList in Java.

**set**

set also contains elements of a specified type, but they are in contrast to list, unordered and unique (that means there are no duplicates allowed). The syntax for set is:

set<type>

As with list, in the place of the type keyword, there should be an identifier of the type of the elements of set. For example, set<string> is a set of strings.

set is mapped to the relevant set types. Set in Python, STL set in C++, HashSet in Java. In PHP, there are no sets, so Apache Thrift's sets are treated in the same way as lists.

**map**

map contains mapping of keys to values. The keys are strictly unique (that is, there couldn't be two values with the same key). The syntax for map is:

map<type1,type2>

As with other container types in the place of type1 and type2 keywords, there should be identifiers of the types of keys (former) and values (latter) of the map, for example, map<i32,string> is a map of unique 32-bit signed integers to strings.

Map is translated to an associative array in PHP, dictionary in Ruby and Python, STL map in C++, and HashMap in Java.

While any valid Apache Thrift type may be a key for the map, due to the restrictions of some programming languages that don't support more complex types as map keys, it is recommended that only basic types be used as keys. Moreover, when using JSON protocol, it is required that the keys are of basic type.

**Usage of containers**

Elements of sets can be of any valid Apache Thrift type.

Containers are used as any other type of variable. For example, struct that among other fields contains a list of strings—list<string>—is defined as:

structCompany {

1: i32 id,

2: string name,

3: list<string> offices

}

A function that takes a list of integers list<i32> as an argument and returns a set of integers set<i32>, is declared in a service as:

set<i32> flatten(1: list<i32>mylist);

To make the code more concise, you may of course use containers with the typedef statement:

typedef map<i32,string>MyMap

**Enums**

Enum (enumerated type) is a data type that consists of a set of named values. The set cannot be modified. Its elements have values; they are assigned sequentially starting with 0, but you can set your own values. Let's have a look at the example:

enum Position {

CEO,

MANAGER,

SPECIALIST,

TRAINEE = 9

}

In this example, the enumerated type Position consists of four values. They are written in uppercase, but it is a popular convention, not a syntax requirement. Everywhere you need, the values have to be referred by their full name, that is, Position.MANAGER.

Enums can be used as any other valid Apache Thrift data type, that is, as a parameter or a return value of functions:

Position getEmployeePosition(1: i32employee\_id)

They also can be used in containers, for example, in list:

list<Position>getFreePositions()

It is also possible to add them to structs:

struct Employee {

1: i32 id,

2: Person person,

3: Position position

}

**Exceptions**

Exceptions are similar to structs, with one difference that they are declared using the exception keyword. In Apache Thrift's implementation, in every programming language, they inherit from the native exception class thus integrating with the native exception handling.

In the interface definition file, exceptions are declared as follows:

exceptionMyError {

1: i32 code,

2: string message

}

exception WrongIdsError {

1: string message,

2: set<i32> ids // list of unaccepted IDs

}

In this case, the MyError exception has a field for numeric error code and a message. WrongIdsError is more complex as it has a field that is a set of signed 32-bit integers. Any valid Apache Thrift type can be used in an exception.

As in Java, Apache Thrift needs you to declare the exceptions that we expect to be thrown by the function:

set<i32>findRecords(1: set<i32> ids) throws (1: MyErrorerror1, 2: WrongIdsErrorerror2)

In this case, the findRecords function may throw MyError or WrongIdsError exceptions depending on the function's logic.

**Services**

This is where all Apache Thrift's type system components are combined to serve its main purpose: define the services that will be accessible from other applications. The thrift command (Apache Thrift's compiler) will parse the service definition and generate client and server stubs, which need to be implemented by the developer, like you did in *Chapter 3*, *Running Your First Apache Thrift Service and Client*.

The service's syntax is familiar to those who program in modern object-oriented programming languages; it looks almost exactly like the interface. The service consists of a set of function declarations, each with its parameters, return types, and optional information about thrown exceptions. It is also possible to declare the function as oneway, which means that the code will not wait for the response (thus, the function has to have void return type).

Services can extend other services. This simple inheritance model means that the child service will include its parent's function definitions.

Let's have a look at the example:

service PeopleDirectory {

oneway void log(1: string message),

void reloadDatabase()

}

service EmployeeDirectory extends PeopleDirectory {

Employee findEmployee(1: i32employee\_id) throws (1: MyError error),

bool createEmployee(1: Employee new\_employee)

}

At the beginning, we have a service named PeopleDirectory, which may be some base service for the people directory. It has a function log:

oneway void log(1: string message)

It takes one argument (message) that when the function is implemented, we may presume will be saved to some logging backend, such as a file or a database. Note that this function is one way, which means that the program won't wait for the result; it is a quite common scenario in a case where a lot of information is being logged to the file. If we will not wait for the result, we can't expect the function to provide any; thus its return value type is void.

Such a service may be working standalone, but it also may be used in some other services, which we want to have something similar. In our example, the EmployeeDirectory service extends PeopleDirectory service, which means it exposes the log and reloadDatabase functions of PeopleDirectory along with its own set of functions. It's a typical inheritance model in object-oriented programming languages, which is implemented using the extends keyword:

serviceEmployeeDirectory extends PeopleDirectory {

Then, we have a declaration of different functions exposed by this service:

Employee findEmployee(1: i32employee\_id) throws (1: MyError error),

bool createEmployee(1: Employee new\_employee)

The functions, similar to those in C++ or Java can take as an argument and return a value of any valid Apache Thrift type (including void). They can also throw an exception (or more than one).

Note that in Apache Thrift, you can extend only one service at a time; there is no multiple inheritance.

**IDL syntax**

Now that you have the knowledge of the Apache Thrift's variable types, it is time to put those together in a single file describing our interface.

Apache Thrift's IDL syntax will be familiar to the developers programming in C++, Java, or even PHP. Let's have a look at the most important components.

**Comments**

Apache Thrift supports three types of comments. The first is a bash-style comment—a line beginning with #:

# This is a comment

The second syntax is C++/Java/PHP-style—a line with // at the beginning:

// This is a comment

The last type of supported comment syntax is C-style multiline with /\* at the beginning and \*/ at the end (using space and \* at the beginning of every line is a common convention in many programming languages, although it is not required):

/\*

\* This is C-style

\* multiline

\* comment.

\*/

**Document**

The formal name for your Apache Thrift IDL file (that is, myfirst.thrift) is **document**. Anything that is inside this file is either a header or a definition. Any document can contain 0 or more headers and definitions. It may sound weird, but after reading this section, you will be able to easily identify every part of your document.

**Headers**

Headers are statements that don't define any objects that can be used in the document, or in the services. They contain special instructions regarding the processing of the file or generating the service stubs.

There are three types of headers that we will review:

1. Thrift include
2. C++ include
3. Namespace

Let's start with the first one.

**Thrift include**

This is Apache Thrift's include; it looks like this:

include "shared.thrift"

This syntax is very common in other programming languages. Apache Thrift compiler will read those files and include any definitions present in this file (structs, services, and so on). Provided files are searched in the current path or by searching in any paths provided in the -I flag to the compiler, for example, if you have some special Thrift files that you want to use through different projects in the ~/includes directory, you should use following command to be sure that they are found and included:

**$ thrift --gen php -I ~/includes myfirst.thrift**

Objects included in such a way are accessed using the name of the file as a prefix to their name, so for example, if you have included the file shared.thrift and in this file there was structmystruct, you can access it with shared.mystruct.

**C++ include**

This is a special keyword used when you specifically want to have some extra include in your C++ code generated from this document.

So, for example, if you want to have the following include statement in your C++ code generated by the Thrift compiler:

#include <vector>

You have to use the following header in your Apache Thrift document:

cpp\_include "<vector>"

**Namespace**

Various programming languages have different methods of categorizing and separating the files related to different logic; such separated units are often called packages, modules, or namespaces. They are called **uniformly namespaces** in Apache Thrift documents. By setting a proper namespace, you order thrift compiler to place the code generated for a given language, in a desired place.

Namespaces are defined per programming language and can be different for each of them:

namespace java myfirst

namespace phpmyfirstthrift

namespace rb mf

Of course, for the sake of simplicity and to avoid mistakes, it is better to have the same namespace for every language using \* in place of the language identifier:

namespace \* myfirst

Every Apache Thrift document is compiled to the desired namespace; it applies to the included documents too.

**Definitions**

Definition is the other type of element present in the Apache Thrift document. In short, everything that is neither include (Thrift or C++) or namespace command is a definition. In the document, various building blocks are defined to reach the ultimate purpose, which is defining the service.

In the Apache Thrift document, you can define:

1. const
2. typedef
3. enum
4. struct
5. union
6. exception
7. service

Those of the definitions that are Apache Thrift's data types were described in the previous section, so here we are going to concentrate only on const and typedef, which were not mentioned earlier.

**const**

The const keyword is used to define the constant value. This value can then be used in other definitions in the document. The purpose of this element is to have one place when some constant value is defined, instead of having it set in different places of the file. The value may change during the development (that is, for testing purposes) but will not change after the code is generated. Common usage for such language construct is to define some physical or mathematical value, for example:

const double PI = 3.1415926

In this example, const is followed by the type name (double), const name (PI), and the value (3.1415926).

The values of the complex types and structs are defined using JSON notation, for example, constant, which is a map, is declared in the following way:

const map<i32,string> CITIES = {0: "New York", 1: "London", 2: "Madrid"}

List constant:

const list<string> LANGUAGES = ["PHP", "Java", "C++"]

It is also possible to set const of previously defined struct, for example:

struct city {

1: string name,

2: i32 population

}

const city NEW\_YORK = {"name": "New York", "population": 8500000}

As you see any valid Apache Thrift type can be used to declare const and there are a lot of possible combinations.

**typedef**

The typedef keyword is used to give other names to the Apache Thrift type. The commonly used case is renaming the integer type names i16, i32, and i64, for example:

typedef i32 integer

typedef i32 myinteger

typedef i64 userid

In this case, integer or myinteger can be used instead of i32 and userid instead of i64. The same Apache Thrift type can be mapped to many different names.

It is also common to give pretty names to the complex types so that they are easier to handle, for example:

typedef map<i32,string> EmployeeList

In this case, map mapping unique 32-bit unsigned integers to strings becomes the EmployeeList type.