Apache THRIFT: A Much Needed Tutorial

This is the first article on Apache Thrift, as there is neither any official documentation nor any tutorial available for the fabulous tool Apache Thrift. This article tries to bridge that gap and introduce you to Apache Thrift and how, when and why you should use it.

What you will learn…

- What is Apache Thrift
- Why we should use Apache Thrift
- How to use Apache Thrift
- Comparison with similar tools
- How it can be scaled

What you should know…

- basic shell commands
- basic knowledge of programming

This article is about Apache Thrift and how it can be used. It also tries to address the challenge of scalability by discussing how Thrift can be used to meet enterprise expectations.

What is Apache Thrift

From the Thrift Website (*http://thrift.apache.org/*):

Thrift is a software framework for scalable cross-language services development. It combines a software stack with a code generation engine to build services that works efficiently and seamlessly between C++, Java, Python, PHP, Ruby, Erlang, Perl, Haskell, C#, Cocoa, JavaScript, Node.js, Smalltalk, and OCaml.

Thrift is an *Interface Definition Language* (IDL) which is used to define and create services between numerous languages as a *Remote Procedure Call* (RPC). Its lightweight framework and support for cross-language communication makes it more robust and efficient than other RPC frameworks like SOA (REST/SOAP) for most of the operations. Through a simple and straightforward IDL, it allows you to create services that are usable by numerous languages. Using code generation, Thrift creates a set of files that can be used to create clients and/or servers. In addition to interoperability, Thrift can be very efficient because of a serialization mechanism which can save both space and time.

In other words, Apache Thrift lets you create a service to send/receive data between two or more softwares that are written in completely different languages/platforms.

History

Thrift was originally developed by the folks at Facebook. It's also one of the "core parts of their infrastructure". After a while, they decided to make it Open Source and finally contributed it to Apache Software Foundation (ASF) in April 2007 in order to increase usage and development. Thrift was then released under the Apache 2.0 license.

The choice of programming language at Facebook is based on what language is best suited for the task at hand. While pragmatic, this flexibility resulted in difficulties when these applications needed to call one another. After some analysis, Facebook engineers did not find anything currently existing which could meet their needs of interoperability, transport efficiency, and simplicity amongst others. Out of this need, they

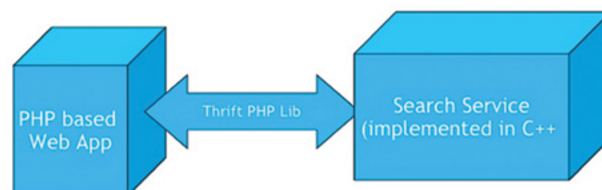

Figure 1. *Facebook Search Service Architecture*

developed efficient protocols and a service infrastructure that became Thrift. Facebook now uses Thrift for their back-end services, the reason for which it was initially designed.

Who uses Thrift?

- Facebook Thrift is one of the core components of Facebook infrastructure. Its search services are implemented in C++, but its Web Application is based on PhP. Thus, to bridge the gap between them, Facebook uses Thrift (Figure 1).
- Evernote Thrift is extensively used in a variety of the Evernote public API.
- Scribe is also built on top of Thrift.
- HBase leverages Thrift for cross-language API.
- The whole list can be found here: *http://wiki.apache.org/thrift/PoweredBy*

How to Download and Install

The stable release of Apache Thrift can be downloaded from: *http://thrift.apache.org/download/*.

Listing 1. *Installing Thrift on Debian/Ubuntu*

```
##Install the pre-requisites
sudo apt-get install libboost-dev libboost-test-
                    dev libboost-program-options-dev
                    libevent-dev automake libtool flex
                    bison pkg-config g++ libssl-dev
sudo apt-get install php5-dev php5-cli
cd /usr/tmp/
fetch "http://www.trieuvan.com/apache/thrift/0.9.0/
                    thrift-0.9.0.tar.gz"
tar xvzf thrift-0.9.0.tar.gz
cd thrift-0.9.0
./bootstrap.sh
./configure
sudo make
sudo make install
```

Listing 2. *Installing Thrift on FreeBSD*

```
##Install the pre-requisites and then,
##on freeBSD, Thrift can be installed from FreeBSD
                    ports

cd /usr/ports/devel/thrift
./bootstrap.sh
./configure
make
make install
```

You can talk the talk.
Can you walk the walk?

[IT'S IN YOUR DNA]

LEARN:
Advancing Computer Science
Artificial Life Programming
Digital Media
Digital Video
Enterprise Software Development
Game Art and Animation
Game Design
Game Programming
Human-Computer Interaction
Network Engineering
Network Security
Open Source Technologies
Robotics and Embedded Systems
Serious Game and Simulation
Strategic Technology Development
Technology Forensics
Technology Product Design
Technology Studies
Virtual Modeling and Design
Web and Social Media Technologies

www.uat.edu > 877.UAT.GEEK

Install on Ubuntu

Commands are given in Listing 1.

Install on FreeBSD

Commands are given in Listing 2.

Install on Windows

- First install some pre-requisites
  - MinGW
  - GNU Build Tools
  - g++ 4.0+
  - bison 2.3-1
  - boost 1.33.1-4
  - boost-devel 1.33.1-4
  - flex 2.5.33-1
  - pkgconfig
  - libtool
- Download thrift.exe
- Copy it to C:\Thrift\thrift.exe
- Now, add "C:\Thrift\" to your PATH environment variable
- Now the thrift compiler can directly be used from CMD prompt.

Install Thrift Editor for Eclipse

- Open Eclipse
- Eclipse –> help –> Install new Software

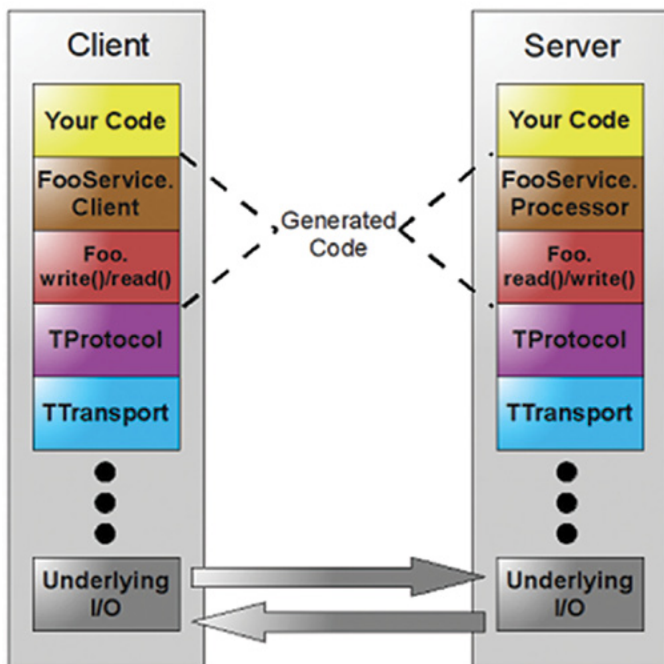

Figure 2. *Apache Thrift client-server architecture*

- Add the URL: *http://thrift4eclipse.sourceforge.net/updatesite/*
- Check the only available package
- Install the package

Architecture

Thrift includes a complete stack for creating clients and servers. Figure 2 depicts the Thrift Stack.

The top portion of the stack is generated code from your Thrift definition file. Thrift services result in generated client and processor code. These are represented by the brown boxes in the figure. The data structures that are sent (other than built-in types) also result in generated code. These result in the red boxes. The protocol and transport are part of the Thrift runtime library. Therefore with Thrift, you can define a service and have the freedom to change the protocol and transport without regenerating your code. Thrift also includes a server infrastructure to tie the protocols and transports together. There are blocking, non-blocking, single and multithreaded servers available. The "Underlying I/O" portion of the stack differs based on the language in question. For Java and Python network I/O, the built-in libraries are leveraged by the Thrift library, while the C++ implementation uses its own custom implementation.

Thrift allows you to choose independently between your protocol, transport and server. With Thrift being originally developed in C++, Thrift has the greatest variation among these in the C++ implementation.

Transport Layer

The transport layer provides simple abstraction for reading/writing to/from the network. The transport layer basically describes "how" data is transmitted. This layer decouples the underlying transport from the rest of the system, exposing only the following interface:

- open
- close
- read
- write
- flush

In addition to the above interface, Thrift also uses ServerTransport interface on the server side to accept or create transport objects. The interface includes:

- open
- listen
- accept
- close

There are a number of transports supported by Thrift:

- TSocket: Uses blocking socket I/O for transport.
- TFramedTransport: Sends data in frames, where each frame is preceded by a length. This transport is required by a non-blocking server.
- TFileTransport: This transport writes to a file. This transport is not included with the Java implementation.
- TMemoryTransport: Uses memory for I/O. The Java implementation uses a simple ByteArrayOutputStream internally.
- TZlibTransport: Performs compression using zlib. It should be used in conjunction with another transport. Not available in the Java implementation.

Protocol Layer

The protocol abstraction defines a mechanism to map in-memory data structures to a wire-format. It specifies how datatypes use the underlying Transport to encode/decode themselves. Separates Data Structure from Transport representation. Thus the protocol implementation governs the encoding scheme and is responsible for (de)serialization. Thrift protocols are stream oriented by nature thus there is no need for any explicit framing. In other words, Protocols describe "WHAT" is actually transmitted. Thrift supports both text and binary protocols. The binary protocols almost always outperforms text protocols, but sometimes text protocols may prove to be useful in cases of debugging. The Protocols available for the majority of the Thrift-supported languages are:

TBinaryProtocol
A straightforward binary format encoding numeric values as binary, rather than converting to text. They are not optimized for space efficiency.

TCompactProtocol
Very efficient and dense encoding of data. This protocol writes numeric tags for each piece of data. The recipient is expected to properly match these tags with the data (If the data is not present, no tag/data pair is present). For Integers, this protocol performs compression using Variable-Length Quantity (VLQ) encoding from the MIDI file format.

TDenseProtocol
It's similar to TCompactProtocol but strips off the meta information from what is transmitted and adds it back at the receiver side. It is still experimental and not yet implemented in Java.

TJSONProtocol
Uses JSON for data encoding.

TSimpleJSONProtocol
A write-only protocol using JSON. Suitable for parsing by scripting languages.

TDebugProtocol
Sends data in the form of human-readable text format. It can be well used in debugging applications involving Thrift.

Processor Layer

A processor encapsulates the ability to read data from input streams and write to output streams. The input and output streams are represented by protocol objects. The processor interface is extremely simple. Service-specific processor implementations are generated by the Thrift compiler.

Thus, the generated code makes the Process Layer of the architecture stack. The processor essentially reads data from the wire (using the input protocol), delegates processing to the handler (implemented by the user), and writes the response over the wire (using the output protocol).

Server Layer

A server pulls together all of the various functionalities described above to complete the Thrift server. First, it creates a transport and specifies input/output protocols for

Listing 3. *file: hello.thrift*

```
namespace java helloworld
service HelloService {
   string sayHello()
}
```

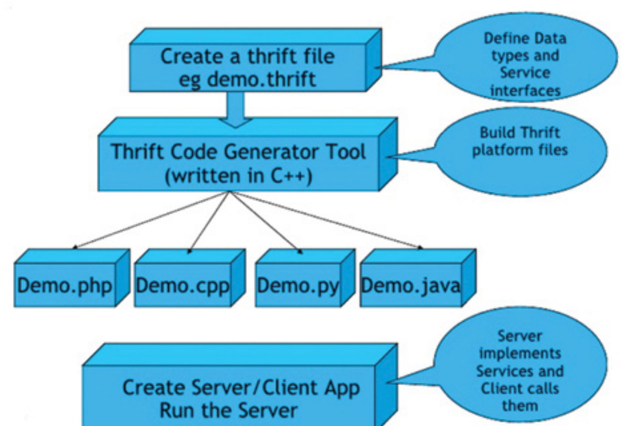

Figure 3. *Thrift Application Development flow-chart*

Listing 4. *Generating Processors in Thrift*

```
##Generating the processor for Server Side
thrift --gen java hello.thrift
##Generating the processor for Client Side
thrift --gen php hello.thrift
```

Listing 5. *file: HelloServiceImpl.java implementing the hello world service*

```
package helloworld;
import org.apache.thrift.TException;

public class HelloServiceImpl implements HelloService.Iface {

    public String sayHello() throws TException {
        return "Hello World!!";
    }
}
```

Listing 6. *file: HelloServer.java which is code for Java server.*

```
package helloworld;

import org.apache.thrift.server.TServer;
import org.apache.thrift.server.TThreadPoolServer;
import org.apache.thrift.transport.TServerSocket;
import org.apache.thrift.transport.TTransportException;

public class HelloServer implements Runnable {

    private static final int PORT =  9090;

    public void run() {
        try {
            TServerSocket serverTransport = new
                      TServerSocket(PORT);
            HelloService.Processor processor = new
                      HelloService.Processor(new
                      HelloServiceImpl());
            TServer server = new TThreadPoolServer(new
                      TThreadPoolServer.Args(serverTransport).
                      processor(processor));
            System.out.println("Starting server on port: "+PORT);
            server.serve();

        } catch(TTransportException e) {
            System.out.println("Message: "+e.getMessage());
            System.out.println("StackTrace: ");
            e.printStackTrace();
        }
    }
    public static void main(String[] args) {
        new Thread(new HelloServer()).run();
    }
}
```

Listing 7. *file: client.php, Client side code.*

```
<?php
    // defining the port and server to listen
    define("PORT", '9090');
    define("SERVER", 'localhost');

    //Global variable where the php library files are stored
    $GLOBALS['THRIFT_ROOT'] = 'thrift';

    //including the library files
    require_once $GLOBALS['THRIFT_ROOT'].'/Thrift.php';
    require_once $GLOBALS['THRIFT_ROOT'].'/protocol/
                 TBinaryProtocol.php';
    require_once $GLOBALS['THRIFT_ROOT'].'/transport/
                 TSocket.php';
    require_once $GLOBALS['THRIFT_ROOT'].'/transport/
                 TBufferedTransport.php';

    //loading the auto-generated package
    require_once $GLOBALS['THRIFT_ROOT'].'/packages/
                 hello/HelloService.php';
?>
<?php
    try {
        //create a thrift connection
        $socket = new TSocket(SERVER, PORT);
        $transport = new TBufferedTransport($socket);
        $protocol = new TBinaryProtocol($transport);

        //create a new hello service client
        $client = new HelloServiceClient($protocol);

        //open the connection
        $transport->open();

        $result = $client->sayHello();
        echo "Result: ".$result;

        $transport->close();
    } catch(TException $tx) {
        echo "Thrift Exception: ".$tx->getMessage()."\r\n";
    }
?>
```

the transport. Then, it creates a processor based on the I/O protocols. It finally waits for incoming connections. When a connection is made, it hands them off to the processor to handle the processing of the request.

Thrift provides a number of servers:

TSimpleServer
A single-threaded server using standard blocking I/O socket. Mainly used for testing purposes.

TThreadPoolServer
A multi-threaded server with N worker threads using standard blocking I/O. It generally creates *N=5* minimum threads in the pool if not specified otherwise.

TNonBlockingServer
A multi-threaded server using Non-Blocking IO (Java implementation uses NIO channels). TFramedTransport must be used with this server.

THttpServer
HTTP server (for JS clients) optionally with REST like URLs.

TForkingServer
Forks a process for each request to server.

TProcessPoolServer
Available in Python. Pre-forks workers to avoid Global Interpreter Lock.

Listing 8. *Non-Blocking server in Java*

```
//Works with Asynchronous client too

public class NonblockingServer {

private static final int PORT = 7911;
    private void start() {
        try {
            TNonblockingServerTransport serverTransport
                    = new TNonblockingServerSocket(PORT);
            HelloService.Processor processor =
                    new HelloService.Processor(new
                    HelloServiceImpl());

            TServer server = new TNonblockingServer(new
                    TNonblockingServer.
                    Args(serverTransport).
                     processor(processor));
            System.out.println("Starting server on port
                    " + PORT.toString() + " ...");
            server.serve();
        } catch (TTransportException e) {
            e.printStackTrace();
        }
    }

    public static void main(String[] args) {
        NonblockingServer srv = new NonblockingServer();
        srv.start();
    }
}
```

Listing 9. *Non-Blocking Client in Java*

```
//Note the usage of TFramedTransport for Non-Blocking
                    Server which would frame the data
//sent over the wire

public class NonblockingClient {
    private void invoke() {
        TTransport transport;
        try {
            transport = new TFramedTransport(new
                    TSocket("localhost", 7911));
          TProtocol protocol = new TBinaryProtocol(transport);

            HelloService.Client client = new
                    HelloService.Client(protocol);
            transport.open();

            System.out.println(client.sayHello());

            transport.close();
        } catch (TTransportException e) {
            e.printStackTrace();
        } catch (TException e) {
            e.printStackTrace();
        }
    }

    public static void main(String[] args) {
        NonblockingClient c = new NonblockingClient();
        c.invoke();
    }
}
```

Listing 10. *Asynchronous Client written in Java*

```
//Uses the Non-Blocking Server given in Listing 8.
//Note that for each operation of the service, a callback will be defined. Also, a new client will
//have to be used with every different operation else an exception will be thrown

public class AsyncClient {

    private void invoke() {
        try {
            HelloService.AsyncClient client = new HelloService.
                    AsyncClient(new TBinaryProtocol.Factory(), new TAsyncClientManager(),
                                new TNonblockingSocket("localhost", 7911));

            client.sayHello(new sayHelloMethodCallback());
        } catch (TTransportException e) {
            e.printStackTrace();
        } catch (TException e) {
            e.printStackTrace();
        } catch (IOException e) {
            e.printStackTrace();
        }
    }

    public static void main(String[] args) {
        AsyncClient c = new AsyncClient();
        c.invoke();
    }

    class sayHelloMethodCallback
            implements AsyncMethodCallback<HelloService.AsyncClient.sayHello_call> {

        public void onComplete( HelloService.AsyncClient.sayHello_call sayHello_call) {
            try {
                string result = sayHello_call.getResult();
                System.out.println("Result from server: " + result);
            } catch (TException e) {
                e.printStackTrace();
            }
        }

        public void onError(Exception e) {
            System.out.println("Error : ");
            e.printStackTrace();
        }

    }
```

Thrift allows only one service per server. This is certainly a limitation and will be addressed later in this article under multiplexing.

How to use?

Figure 3 describes how the flow of a Thrift Application is made and Figure 4 shows the anatomy of a Thrift service.

First we'll make a small "Hello World" application which aims to make RPC's between Java (Server) and PHP (Client).

We'll start by making a Thrift IDL file named "hello.thrift" as shown in Listing 3. The full tutorial for making a Thrift IDL file can be found here: *http://diwakergupta.github.io/thrift-missing-guide/thrift.pdf*.

Then we'll generate the files for processors layer using the Thrift compiler for both Client and Server using the commands given in Listing 4.

Make a new project in Eclipse with type, "Dynamic Web Project". Put the "thrift file" in the `<project-name>/Java/Resources/src/` directory. Copy the lib files (`libthrift-<version>.jar, build/lib/*`) to `<project-name>/WebContent/WEB-INF/lib/` folder. Now, we have to implement the services mentioned in the thrift file. Write a new class named "HelloServiceImpl" implementing the "HelloService.Iface" interface under the package "helloworld". The code for the same is given in Listing 5.

Now, we have to write the server which actually caters to the client's requests in Java. We will call this "HelloServer" implementing the "Runnable" interface. This class will also come under the package "helloworld". The code for the same is given in Listing 6. As you can see, here we are.

Now finally, we have to make the Client, which is written in PHP. We already generated the client side processor from the Thrift IDL. So first, we include all the Thrift run-time libraries by creating a new directory named "thrift" and copy all the php library files available in the directory `/path/to/thrift-version-folder/lib/php/src/` to the newly created directory. Also, create a new sub-directory named "packages" in "thrift" directory and copy the auto-generated PHP package here. Create a new file <client-file>.php adjacent to the "thrift" directory. The contents of the PHP file are given in Listing 7.

We now know how to make a basic Server-Client Application in Thrift which uses different language/platform for both the server and client. We will now try and explore the different available server and client combinations available to us for improving performance. The previous example we gave was of *Blocking* Server and Client. Now we'll make another server and client which is *Non-Blocking* in nature. The codes for Non-Blocking server and client are given in Listing 8 and Listing 9 respectively.

Last but not least, we will try and implement an *Asynchronous* Server and Client. We can write asynchronous clients to call a Thrift service. A callback needs to be registered which will get invoked at successful completion of the request. Blocking mode server didn't work (method invocation returned with an empty response) with the asynchronous client (Maybe it's because we are using TNonblockingSocket at the client side. See construction of HelloService.AsyncClient in Listing 10. So this may be the proper behaviour). Non-blocking mode server given in Listing 8 seems to work without an issue. So you can use the non blocking server from earlier with the client shown in Listing 10 to implement Asynchronous mode.

So this is how we can use different combinations of transports, protocols, and servers to suit our needs without changing much of the code.

It is now worth summarizing the pros and cons of the Thrift approach:

Benefits

- Cross-Language serialization with lower overhead than alternatives such as SOAP due to use of binary format.
- A lean and clean library. Neither any framework to code nor any XML configuration files.
- The language bindings are very natural. For example, Java uses `ArrayList<int>` and similarly C++ uses `std::vector<int>`.
- The application level wire format and the serialization level wire format are clearly separated. They can be modified independently.
- Soft versioning of the protocol. Thrift does not require a centralized and explicit mechanism like major-version/minor-version. Loosely coupled teams can freely evolve RPC calls.
- No build dependencies or non-standard software. No mix of incompatible software licenses.
- Changing the current interface is very easy. The old Client or Server remain compatible with the new Server or Client respectively, silently ignoring any deprecated or newly added fields if necessary.
- Thrift supports a wide variety of languages and environments.
- Service inheritance, subservices implement all functions of their base services and can have additional functions.

Limitations

- Only one service per server. This can be addressed using multiplexing discussed in a short while but still adds to the complexity.
- There can be no cyclic structs. Structs can only contain structs that have been declared before it. A struct also cannot contain itself.
- Important OOP concepts like inheritance and polymorphism are not supported.
- Lacks full documentation.
- Null cannot be returned by a server. Instead a wrapper struct or value is expected.
- No out-of-the-box authentication service available between server and client.
- No Bi-Directional messaging is available.

Multiplexing of services

Despite being a powerful and efficient cross-language communication tool, Thrift's services are challenged by high administrative and maintenance overheads. The fact remains that *every Thrift server is only capable of exposing only one service at a time*. In order to host multiple functions, Thrift provides organizations with the following options:

- Write a monolithic, unwieldy implementation and host it as a single service.
- Host multiple small services across a series of ports.

If we follow the first option, writing monolithic services elevates the cost of development and maintenance as time passes since the complexity increases with the addition of any new service. Even the second option can prove deadly in the long run as the number of ports consumed to host the multiple services keeps on increasing. Ports are a limited resource and need to be judiciously used. Clients will have to maintain too many connections for each service they want to use. Also, as many ports are opened, security needs to be properly scrutinized, which introduces more overhead.

Thus, we introduce multiplexing by extending the Thrift framework to create and host multiple services on each server. The baseline approach is to assign a symbolic name to each service which is referred to as "service context". This will help us host multiple services on each server where each service can be recognized by its respective service context. The basic architecture can be seen in Figure 4.

Let me explain the different components involved:

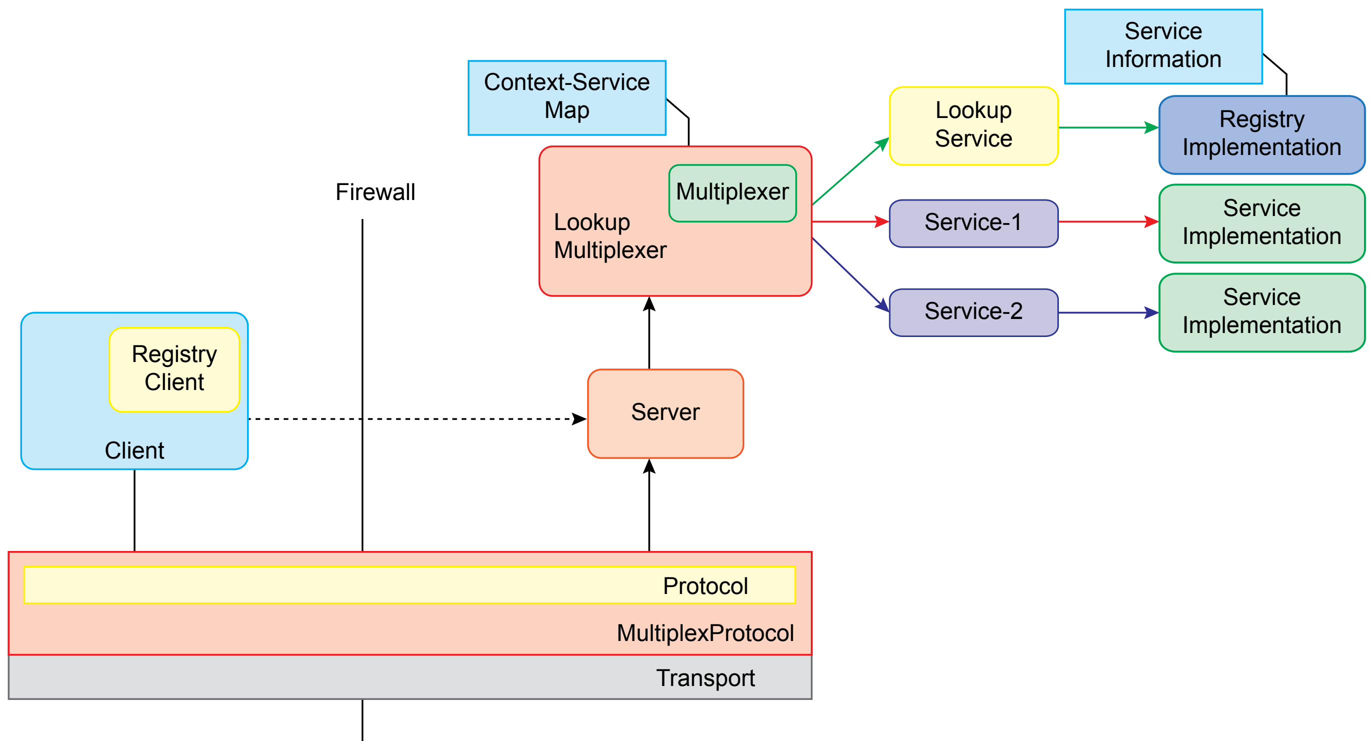

Figure 4. *Thrift Multiplexing architecture with registry lookup*

Multiplexer
This acts as the server side request breaker and identifying the service that the client has requested for, based on the service context provided by the client. This component maintains a mapping between the service context and the available services in that context.

MultiplexProtocol
It is a wrapper class around the underlying protocol that is capable of embedding service context to the message on the client side and fetching it on the server side.

RegistryLookup
In order to reduce the overhead associated with managing the service context manually, we have created a registry system that is responsible for managing meta information about the service context and their services.

ServiceInformation
This class captures/represents the information regarding services on a particular server. This object should be capable of transmitting across the network and hence be used by the client to get service information. Service information consists of service context, service name and description.

MultiplexServer
It is a new abstract server (wrapper to the old Server) which is capable of hosting any server implementation on any transport and protocol using TMultiplexLookup. It basically provides an additional degree of freedom when it comes to hosting a new service on different transport and protocols with no additional coding effort.

Table 1. *Protocol efficiency study.*

Technology	Message Size (The smaller the better)	% larger than TCompact Protocol
Thrift – TCompactProtocol	260	--NA--
Thrift – TBinaryProtocol	445	71.15%
Protocol Buffers	255	-1.9%
Remote Method Invocation	880	238.46%
REST – JSON	536	106.15%
REST – XML	809	211.15%

Comparison With Similar Technologies

As there was no data available on the internet about how different protocols perform against each other, I tried to compare some of the available technologies for RPC between different languages, namely, Apache Thrift (TCompactProtocol, TBinaryProtocol), Google Protocol Buffers, Java Remote Method Invocation, REST using JSON and XML. These tests were conducted on a simple machine available at any home. To compare the protocols, I send out the same amount of data from the different technologies. I used WireShark to capture the sizes that were transferred between Client and Server. You can also perform the same exercise to see the performances for yourself. The results are tabulated and shown in Table 1.

Thrift has a clear advantage in the size of its payload particularly compared to RMI and XML-based REST. Protocol Buffers from Google are effectively the same given that the Protocol Buffers number excludes messaging overhead.

To compare the runtime performances of the different technologies, I used the same computer to run both client and server whose specifications are as follows:

- Operating System: Ubuntu Linux 12.04
- System CPU: Intel Core i5-430M processor
- Memory: 4GB
- Java Version: Java(TM) SE Runtime Environment (build 1.7.0_04-b20)

I made around 100,000 queries from the Client monitoring the CPU usage and the total time taken. The results are tabulated and shown in Table 2.

Table 2. *Runtime Performance comparison*

Technology	CPU usage (SERVER)	CPU Usage (CLIENT)	Average Wall Time (mm:ss)
Thrift – TCompactProtocol	22%	30%	01:12
Thrift – TBinaryProtocol	20%	33%	01:35
Protocol Buffers	41%	28%	01:20
Remote Method Invocation	46%	16%	02:18
REST – JSON	73%	20%	05:42
REST – XML	81%	12%	06:38

The tests yielded some interesting observations. In terms of wall time, Thrift clearly outperformed REST and RMI. In fact, TCompactProtocol took less than 20% of the time it took REST-XML to transmit the same data. The clear dominance of the binary protocols should not be too surprising, as binary data transmission is well-known to have higher performance than text-based protocols. RMI in fact significantly out-performed JSON-based REST in wall time, despite its significantly larger payload size.

References

1. Apache Thrift official: *http://thrift.apache.org/*
2. Wikipedia Page: *http://en.wikipedia.org/wiki/Apache_Thrift*
3. Thrift missing guide: *http://diwakergupta.github.io/thrift-missing-guide/thrift.pdf*
4. Slide on Thrift: *http://www.slideshare.net/talentica/building-scalable-and-language-independent-java-services-using-apache-thrift-6286467*
5. slide on facebook architecture: *http://www.slideshare.net/dleyanlin/facebook-architecture-13994448*
6. Silde on thrift: *http://www.slideshare.net/dvirsky/introduction-to-thrift*
7. java & web develop tips: *http://www.javawebtips.com/117109/*
8. Apache Thrift: *http://jnb.ociweb.com/jnb/jnbJun2009.html*
9. Thrift Home: *http://incubator.apache.org/thrift*
10. Thrift Wiki: *http://wiki.apache.org/thrift*
11. Thrift Installation: *http://wiki.apache.org/thrift/ThriftInstallation*
12. Thrift Installation (Win32): *http://wiki.apache.org/thrift/ThriftInstallationWin32*
13. Wikipedia Page on Variable-Length Quantity(VLQ): *http://en.wikipedia.org/wiki/Variable_length_unsigned_integer*

The CPU percentages yielded some interesting numbers. While the Thrift and Protocol Buffers servers had the highest server CPU percentages, the REST clients had the highest CPU percentages of the clients. For whatever reason, Thrift and REST disproportionately place their CPU loads on their clients and servers. Protocol Buffers balanced its load most evenly between client and server, but then again this was a simple quick hand-rolled server that I wrote for this article. While I did not have time to analyze the cause of the CPU load, the Thrift and Protocol Buffers examples needed to do manual conversion of objects between what is transmitted and what is used. The RMI and REST implementations required no such object conversion. This extra bit of work may account for the additional CPU utilization on the Thrift and Protocol Buffers servers.

Given the poor performance of REST, there may certainly be higher performing servlet containers than Jetty that could be used as part of this test. Jetty was merely chosen because of its relative ease in implementation and ease in bundling the sample code used in this article for download. Doing some quick searches, I found one performance comparison that showed Apache Tomcat to be faster than Jetty and another that showed them at parity. Neither study showed anywhere near a performance difference to make up for the wall time performance of the binary protocols.

All of these technologies are roughly equivalent in the amount of coding complexity required to make them work. This excludes Protocol Buffers of course as it contains no services infrastructure. It should also be noted that Thrift generates all the code you need for a client or server for each language it supports. Java was the server of choice in this article, but other languages could be used if they are better suited – one of the main reasons Thrift was developed in the first place. That being said, I found many of the implementations incomplete. As mentioned previously, the Python implementation, for instance, only had the TBinaryProtocol implemented.

Conclusion

This article introduced you to the tool Apache Thrift and the different ways you can use it and shows how Apache Thrift can fulfill any needs with the flexibility it provides in choosing the different layers of the architecture separately. It outmatches any other similar technology available, even Google protocol buffers, by providing more language support and flexibility. A sneak peek was given on how multiplexity can be used to tackle one of the limitations Apache Thrift has; a full implementation of it can be a whole article in itself and may be discussed in another article.

CHIRAG MAHESHWARI

Chirag is a pre-final year undergraduate (B.Tech) student at the Indian Institute of Technology Guwahati, studying Mathematics and Computing. He is currently working as a Software Developer Intern at S&P Capital IQ, Hyderabad. He's a technology enthusiast and likes to stay updated with all the latest in the field. He can be contacted online at http://digital-madness.in/.