**Thrift Lab 2020**

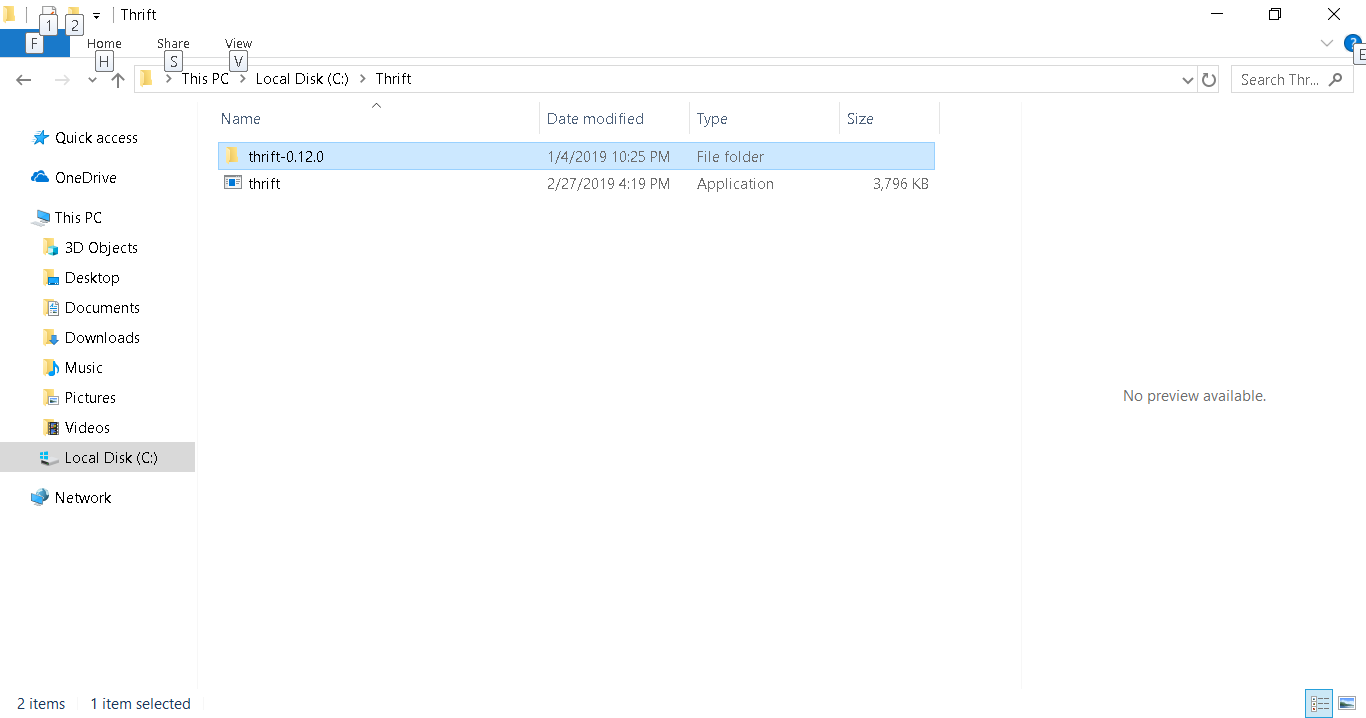
**Exercise 1 – Downloading and testing Thrift**

Download Thrift for Windows and the Thrift tarball from <https://thrift.apache.org/download>

Create a folder called Thrift and place the two files there

No installation is needed; you may work directly with the downloaded file. For consistency, I suggest renaming the file that you downloaded (that is, thrift-0.12.0.exe) to just thrift.exe.

Unzip the tarball (twice) in the same place – this contains the Thrift libraries.



In order to be able to run this command from any directory on your computer, add this file's location to the PATH environment variable. (It varies from version to version, but in most cases, it may be found if you right-click on **My Computer** and choose **Properties**. Then, look for the **Advanced** panel and the **Environment Variables…** tab. Or type Path in the search button. If in doubt, the solution can be found easily on Google).

To test whether Apache Thrift is installed and works properly, we will generate some code for a service, which does nothing - just to check whether everything is in place.

Using notepad++, write the following IDL definition and put it in a file called **test.thrift** in a new folder called ThriftTest

# this is just a Test service, which contains two methods

service Test {

# this method probably does nothing

void donothing(),

# this method probably multiplies two numbers

i32 multiply(1:i32 number1, 2:i32 number2),

}

As you may see, we defined a simple service called Test. It contains two methods: one called donothing, which doesn't return anything (void), and the other, multiply, which takes two arguments of type i32 (which is 32-bit signed integer)—number1 and number2—and also returns a 32-bit signed integer.

At this stage, it doesn't matter what the methods do; they even don't have to be implemented. To see if you did everything properly, run the following command in the directory where you saved your test.thrift file:

**thrift --gen php --gen py --gen java test.thrift**

The command that you just ran takes the test.thrift file, parses it through the Apache Thrift compiler, and generates service files in PHP, Python and Java languages that you may use to develop your services.

The default output directory is gen-<language\_name>; so in this case, the directories are gen-php, gen-py and gen-java. Check whether these directories were created, and in fact, whether there are some PHP or Python files. If yes, congratulations; your Apache Thrift was installed successfully!

As an extra task, browse through the Java files and see the vast amount of work that Apache Thrift does for you while you don't need to write a single line of code yet.

**Exercise 2 –Write a Thrift service to add two numbers. Write the client and server in Java.**

Based on article at: https://dzone.com/articles/apache-thrift-java-quickstart

Create a new folder called dzone on your C: drive

For simplicity, copy and paste the thrift library (thrift-0.13.0) to dzone

You need to download and unzip a logging module called Simple Logging For Java (SL4J) (<https://www.slf4j.org/download.html>)

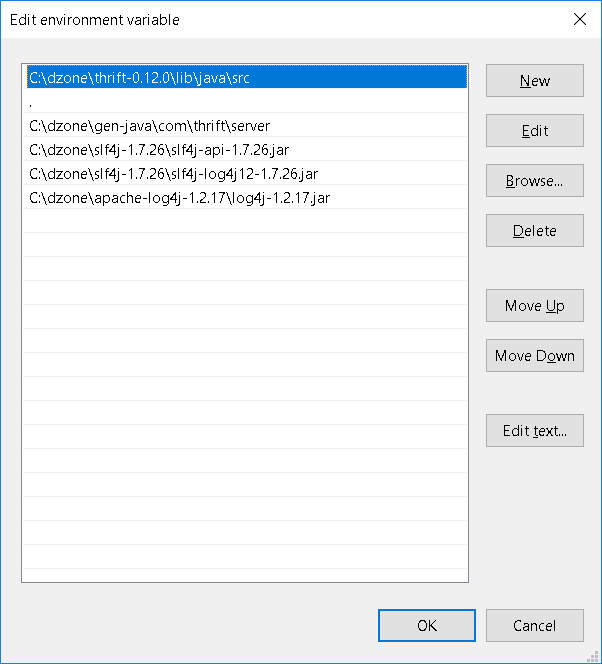
Also download and unzip apache’s log4j: <http://www.apache.org/dyn/closer.cgi/logging/log4j/1.2.17/log4j-1.2.17.zip>

Put the slf4j folder in dzone folder

Also put the apache folder in dzone

“To use SLF4J, it is required to include 3 jars SLF4J API (slf4j-api-x.x.x.jar), SLF4J bindings jar (eg. slf4j-log4j12-1.6.5.jar) and the actual logging framework (eg. Apache’s log4j)”

I added the following to classpath:



Note one path is the current directory (.)

Now we will write an interface file to define the addition service

Add the interface file as add.thrift to the dzone folder,

namespace java com.thrift.server // defines the namespace

typedef i32 int //typedefs to get convenient names for your types

service AdditionService { // defines the service to add two numbers

int add(1:int n1, 2:int n2), //defines a method

}

Now open a command line and compile the interface file using thrift

thrift --gen java add.thrift

This will generate a gen-java folder with com/thrift as sub folders

It also create a client and server folder i.e. you will have com/thrift/server and com/thrift/client folders

It also generates an AdditionService.java file in the server folder.

You must write an AdditionServiceHandler.java and place it in the server folder

package com.thrift.server;

import org.apache.thrift.TException;

public class AdditionServiceHandler implements AdditionService.Iface {

@Override

public int add(int n1, int n2) throws TException {

return n1 + n2;

}

}

Now write the server file called MyServer.java and put it in the server folder

package com.thrift.server;

import org.apache.thrift.transport.TServerSocket;

import org.apache.thrift.transport.TServerTransport;

import org.apache.thrift.server.TServer;

import org.apache.thrift.server.TServer.Args;

import org.apache.thrift.server.TSimpleServer;

public class MyServer {

public static void StartsimpleServer(AdditionService.Processor<AdditionServiceHandler> processor) {

try {

TServerTransport serverTransport = new TServerSocket(9090);

TServer server = new TSimpleServer(

new Args(serverTransport).processor(processor));

// Use this for a multithreaded server

// TServer server = new TThreadPoolServer(new

// TThreadPoolServer.Args(serverTransport).processor(processor));

System.out.println("Starting the simple server...");

server.serve();

} catch (Exception e) {

e.printStackTrace();

}

}

public static void main(String[] args) {

StartsimpleServer(new AdditionService.Processor<AdditionServiceHandler>(new AdditionServiceHandler()));

}

}

Finally write the Addition.client.java file and save it in the client folder

package com.thrift.client;

import org.apache.thrift.TException;

import org.apache.thrift.protocol.TBinaryProtocol;

import org.apache.thrift.protocol.TProtocol;

import org.apache.thrift.transport.TSocket;

import org.apache.thrift.transport.TTransport;

import org.apache.thrift.transport.TTransportException;

import com.thrift.server.AdditionService;

public class AdditionClient {

public static void main(String[] args) {

try {

TTransport transport;

transport = new TSocket("localhost", 9090);

transport.open();

TProtocol protocol = new TBinaryProtocol(transport);

AdditionService.Client client = new AdditionService.Client(protocol);

System.out.println(client.add(100, 200));

transport.close();

} catch (TTransportException e) {

e.printStackTrace();

} catch (TException x) {

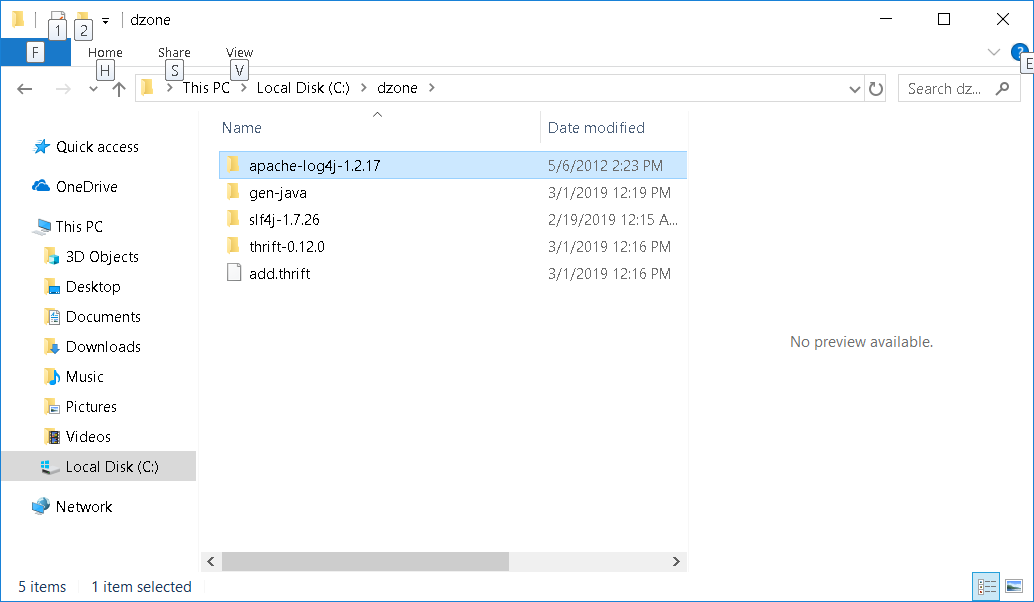
x.printStackTrace();

}

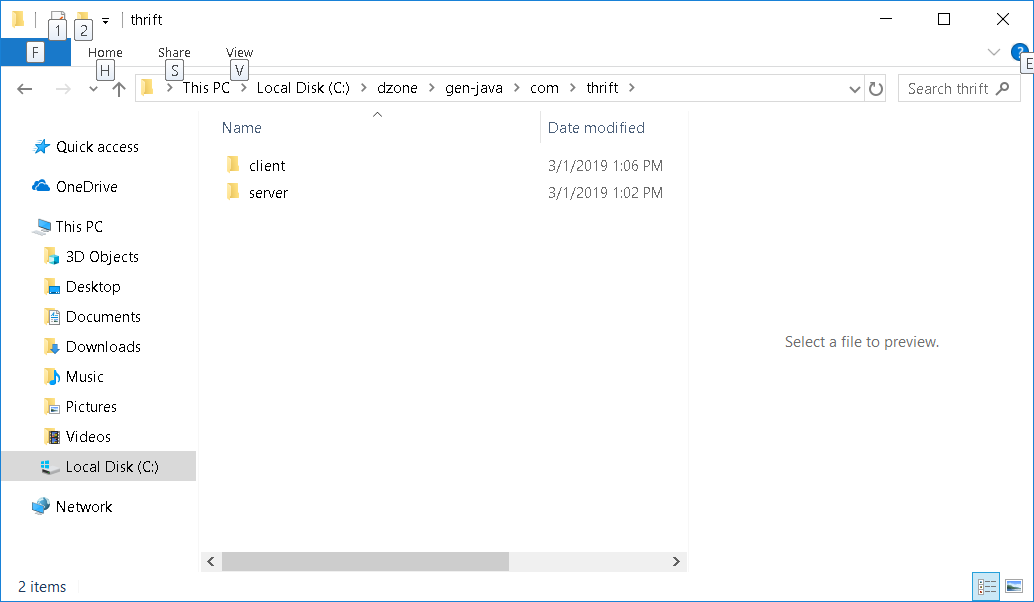
}

}

You should now have a folder system like this:



And



Open a command prompt and cd to the gen-java folder

I compiled from gen-java folder as follows

javac --add-modules java.se.ee ./com/thrift/server/AdditionService.java

javac --add-modules java.se.ee ./com/thrift/server/AdditionServiceHandler.java

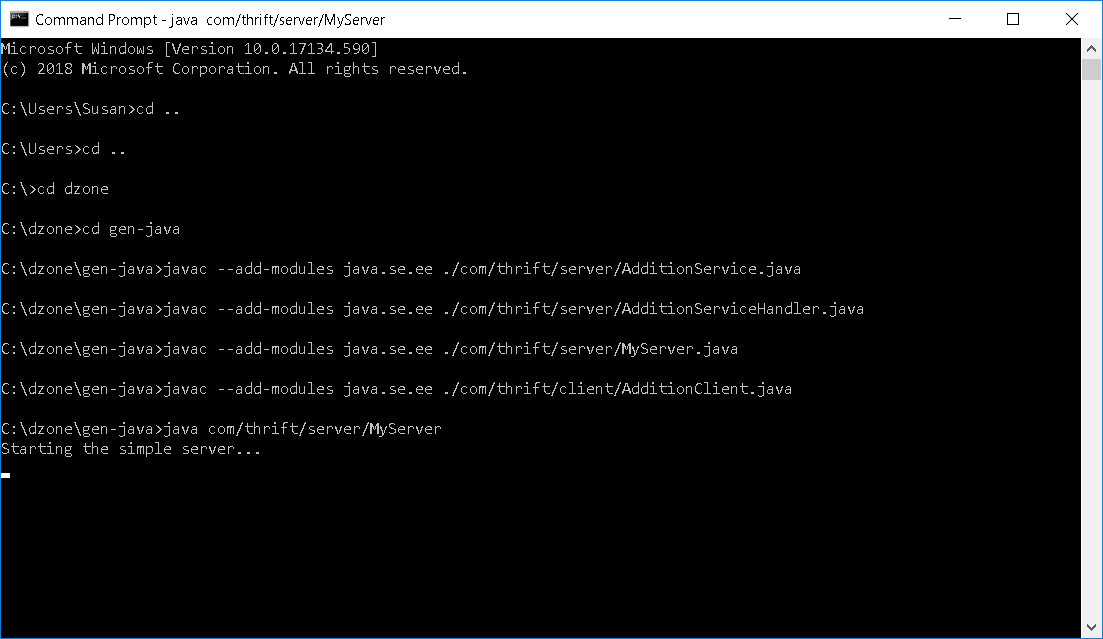
javac --add-modules java.se.ee ./com/thrift/server/MyServer.java

javac --add-modules java.se.ee ./com/thrift/client/AdditionClient.java

(The add-modules were necessary to get all required java modules from the jdk as some are in the ee version.)

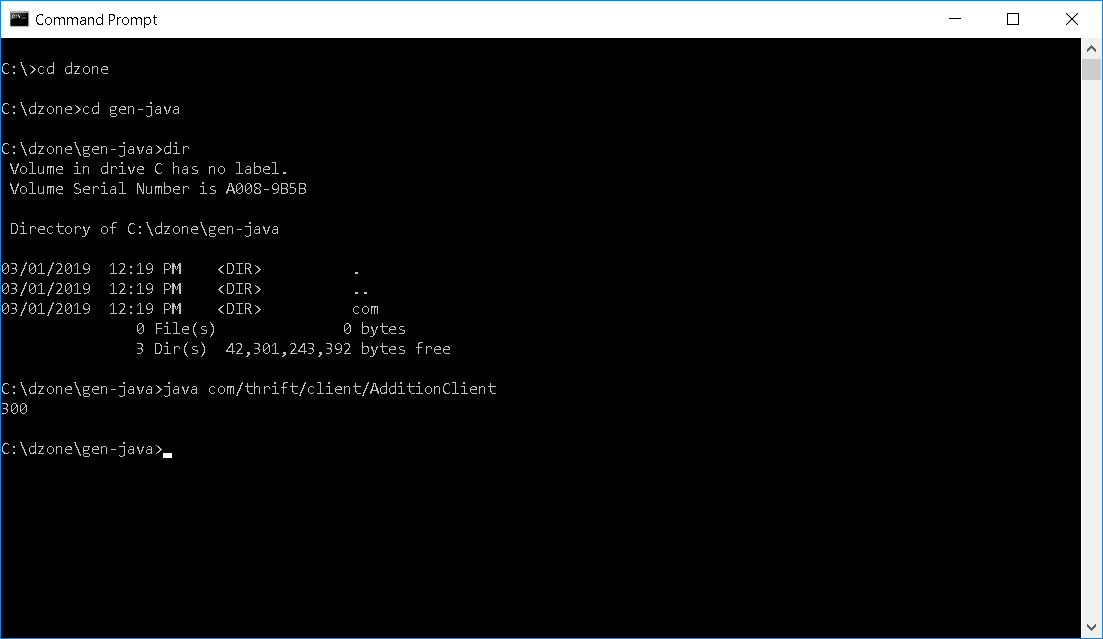
Now run the server

java com/thrift/server/MyServer



And the Client

java com/thrift/client/AdditionClient



**Exercise 3**

Write a Thrift program to reverse a word – both client and server in Java

Steps:

* Designing the services
* Preparing interface description
* Generating the service and client libraries
* Implementing services and clients
* Running the server and clients

**Exercise 4**

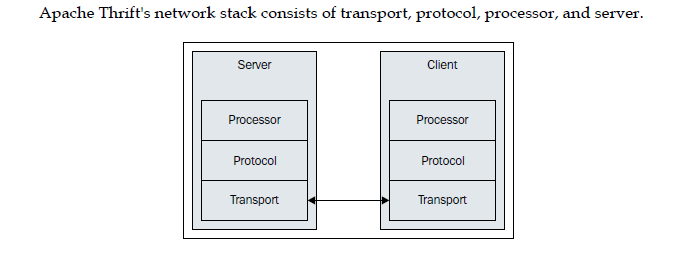
Write the client for exercise 3 in a different language you are familiar with.

**Exercise 5**

Compare and contrast Thrift with message passing using sockets

**How Thrift Works**

**The network stack**



**Transport**

The transport layer provides a way to read and write data from and to the network or any other medium that you want to use.

It is independent from the protocol, so you are able to separately choose how you will serialize the data to be transferred.

Apache Thrift offers a wide spectrum of transports that could be used depending on the architecture of your solution.

Their availability is not consistent; some are widely available while others only in selected languages.

The documentation is also sparse, sometimes even none; one has to look through the implementation source code to work out the details.

All of the transports implement Apache Thrift's TTransport interface in respective programming languages.

Some of the transports are end point transports (it means they write and read directly to or from the device) others are layered transports (it means they are chained with other transports).

The following transports are widely supported:

**TSocket**: This uses a blocking socket, which means that only one connection can be active at a time. Thus, it is not a very good solution for production.

**TPhpStream** (only in PHP): This is useful in situations when we have HTTP server running (that is, Apache or nginx) and want to output the result of the PHP script through this server without actually running our own server.

**TBufferedTransport**: Other transports are often wrapped around in this one, as it provides buffering of input and output data.

**TFramedTransport**: This is also a wrap-around layer to provide framing of the payload.

**TFileTransport** (or TFileObjectTransport in Python): This is used to read and write to the file.

**Protocol**

Protocol is responsible for mapping the in-memory structures (simple and complex data types) to a format that can be transmitted over the selected transport.

This process is called serialization (or **encoding**) when transmitting the message and deserialization (or **decoding**) when receiving it.

Protocol is independent of the transport used.

Protocols in Apache Thrift face the same issues as the transports; there is a wide offer that can be matched to your architecture.

However, protocols' availability is inconsistent and very often there is lack of documentation; the developer needs to look through the source code to figure out the details.

This is a list of the most popular protocols:

**TBinaryProtocol**: This is a simple protocol that converts all data to binary values. You should use it if you don't have any specific needs, as this one is the most universal.

**TCompactProtocol**: This Apache Thrift's own protocol uses a lot of optimizations to make the payload as small as possible.

**TJSONProtocol**: The payload is encoded as a JSON string.

All of the transports implement Apache Thrift's TProtocol interface in respective programming languages.

**Processor**

Processor is generated by the Apache Thrift compiler from your interface description document, so you don't have much choice here.

It reads data from the input protocol, passes it to the handler, and sends the result to the output protocol. (Normally, both input and output protocols are the same.)

**Server and client**

The server combines all of the previously mentioned layers; it creates the transport, input, and output protocols (most of the time it is the same) processor based on the generated code, and protocols.

Then it runs and waits for the incoming connections on the specified port.

You can use any of the servers implemented with Apache Thrift.

As with transports and protocols, there is a wide variety, and every programming language has a different choice.

You can select any server you like the best for your solution. The most popular are:

**TNonblockingServer**: This is a multithreaded, non-blocking I/O server that is optimized for handling concurrent connections.

**TThreadPoolServer**: This is a multithreaded, blocking I/O server that uses much more resources than the previous one, but offers a better throughput.

**TSimpleServer**: This is mainly used for testing purposes. It is single threaded with blocking I/O, which means it can process only one connection at a time.

**Example**

You may see the example of how the network stacks works in the excerpt from the Java code above:

StartsimpleServer(AdditionService.Processor<AdditionServiceHandler> processor) {

try {

TServerTransport serverTransport = new TServerSocket(9090);

TServer server = new TSimpleServer(

new Args(serverTransport).processor(processor));

…

}

public static void main(String[] args) {

StartsimpleServer(new AdditionService.Processor<AdditionServiceHandler>(new AdditionServiceHandler()));

}

Here ***main*** creates an instance of **AdditionServiceHandler** implemented by the developer and as the name states, it is used to handle the service.

This is passed to the **Processor** generated by Thrift from the interface definition.

The processor is passes onto the ***StartsimpleServer*** method of the class.

The Transport is chosen (in this case an instance of **TServerSocket.**

Finally a server is chosen, in this case a **TSimpleServerand** and this is passed both the processor object and the transport object.

The processor reads the data from the protocol (see the client – it chooses a TBinaryProtocol) passes it to handler, and writes the result to the same protocol and back to the client.