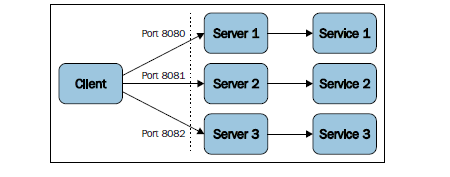
Thrift Service Multiplexing

**Service multiplexing**

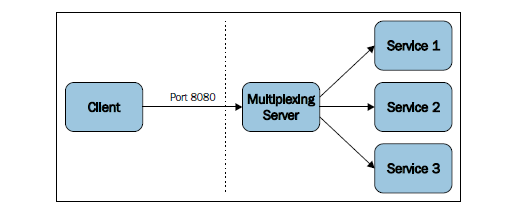
Apache Thrift servers are run on a specified port on which they listen and wait for incoming connections. This is convenient when you want to expose one service. However, what about a situation where you would like to provide access to 2, 3, 10, or 50 services? Imagine the trouble of running theses services as 50 separate servers on 50 separate ports. How much administration work would be needed, and how hard it would be to ensure the security of all the services.

This problem is not an imaginary one—it is not uncommon for an enterprise to have the need to expose more than a one or two services. Fortunately, there is a solution for this, which I briefly mentioned earlier: multiplexing.

Multiplexing is the ability to expose multiple services through one server and on one port. Not much work is really required here; you just need to use a proper multiplexed protocol. This concept is illustrated in the following figure:



This is the basic way to expose the services. Every server runs separately and listens on a different port (8080, 8081, and 8082 are used in this example). A client needs to be configured and connected to multiple different endpoints. When these endpoints change (for example, when the hostname or port number is changed), all of the configurations need to be updated. This leads to maintainability issues.



When we multiplex the services, only one server is exposed on a single port. It is the server's job to route the requests to proper services. Clients need to have only one endpoint in their configuration (the hostname and port number) so that maintenance is easier. The same goes for the server's administrators who have only one instance to deal with.

Let's look at the examples. In Java, you need to use TMultiplexedProcessor in the server and register the services' processors:

import org.apache.thrift.server.TServer;

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

import org.apache.thrift.server.TThreadPoolServer;

import org.apache.thrift.transport.TServerSocket;

import org.apache.thrift.transport.TServerTransport;

import org.apache.thrift.TMultiplexedProcessor;

// import code generated by Apache Thrift compiler

import myservices.\*;

public class MyMultiplexedServer {

// define separate handlers and processors for both services

public static Service1Handler handler1;

public static Service1.Processor processor1;

public static Service2Handler handler2;

public static Service2.Processor processor2;

public static void main(String [] args) {

handler1 = new Service1Handler();

processor1 = new Service1.Processor(handler1);

handler2 = new Service2Handler();

processor2 = new Service2.Processor(handler2);

TMultiplexedProcessor mprocessor = new TMultiplexedProcessor();

// register processors for both services with multiplexed processor

// note the labels for both services

mprocessor.registerProcessor("Service1", processor1);

mprocessor.registerProcessor("Service2", processor2);

// the rest is similar as for the single service

Runnable server = new Runnable() {

public void run() {

myserver(mprocessor);

}

};

new Thread(server).start();

}

public static void myserver(TMultiplexedProcessor processor) {

TServerTransport serverTransport = new TServerSocket(8080);

TServer server = new TThreadPoolServer(new TThreadPoolServer. Args(serverTransport).processor(processor));

System.out.println("Starting multiplexed server on port 8080...");

server.serve();

}

}

As you must have noticed, here, we used the code from *Chapter 5*, *Generating and Running Code in Different Languages*, so you can compare it with the multiplexed version. We use Service1 and Service2 for instructional purposes. We need to create separate handlers and processors for them. Let's look at excerpts from the preceding code:

public static Service1Handler handler1;

public static Service1.Processor processor1;

public static Service2Handler handler2;

public static Service2.Processor processor2;

and:

handler1 = new Service1Handler();

processor1 = new Service1.Processor(handler1);

handler2 = new Service2Handler();

processor2 = new Service2.Processor(handler2);

Then, these processors need to be registered with TMultiplexedProcessor:

TMultiplexedProcessor mprocessor = new TMultiplexedProcessor();

// register processors for both services with multiplexed processor

// note the labels for both services

mprocessor.registerProcessor("Service1", processor1);

mprocessor.registerProcessor("Service2", processor2);

Note that we using the services' names as labels that will distinguish them when the clients send their requests.

The rest is similar to the regular server code for the single instance: the server is started and it occupies only one designated port. The processor will be able to route the requests to the called services.

Now let's take a look at how multiplexing is reflected on the client's side:

// Import code generated by Apache Thrift compiler

import myservices.\*;

import org.apache.thrift.transport.TTransport;

import org.apache.thrift.transport.TSocket;

import org.apache.thrift.protocol.TBinaryProtocol;

import org.apache.thrift.protocol.TProtocol;

import org.apache.thrift.protocol.TMultiplexedProtocol;

public class MyMultiplexedClient {

public static void main(String [] args) {

TTransport transport = new TSocket("localhost", 8080);

transport.open();

TProtocol protocol = new TBinaryProtocol(transport);

// create protocol and clients for both services

// remember about the labels - the same as in the server

MultiplexedProtocol protocol1 = new TMultiplexedProtocol(protocol, "Service1");

Service1.Client client1 = new Service1.Client(protocol1);

MultiplexedProtocol protocol2 = new TMultiplexedProtocol(protocol, "Service2");

Service2.Client client2 = new Service2.Client(protocol2);

// call remote functions using the client objects

client1.somefunction(42);

client2.otherfunction("abc");

transport.close();

}

}

We need to make the client aware of multiplexing. We do this by wrapping the regular protocol with TMultiplexedProtocol. That's it; we can call functions from both the services. Note that we need to use the same labels as the ones used in the server.

As you can see, multiplexing in Apache Thrift is relatively simple, but it is a powerful tool when you need to serve many services at once. There are more complex solutions when you need them, for example, to include not only Thrift, but also other services in your multiplexing.