Microservices

Lab 4 - Modern RPC

Apache Thrift is a modern, microservice friendly RPC solution. Apache Thrift uses a C like interface description language (IDL) to represent interfaces for cross language implementation. Also at the time that Thrift was being developed, all languages in mainstream use included collections such as map, set and list. Their ubiquity is a testament to their usefulness. Apache Thrift makes it possible to pass collections, structures, unions, and exceptions to identify various application level error cases encountered in the service handler.

Perhaps the most powerful feature of Apache Thrift is its interface evolution capability. By encoding each element of a struct or parameter list with an ordinal value, Thrift gains the ability to add, and occasionally remove elements from an interface without damaging compatibility with existing clients.

In this lab we will build a simple Python microservice for capturing and reporting on the trash can fill levels as well as a test client using Apache Thrift.

1. Defining and Building an Interface

Most RPC systems require you to define your interface in IDL. This is usually done before or while you code your service. This is actually beneficial because it allows you to focus on the interface and its business requirements, not how it might be implemented. This interface will become the access layer for some bounding context within your microservice based application.

For our service we need to be able to return all of the trash cans above a certain threshold and set trash can levels. Our IDL might look something like this:

```
typedef i64 CanId

struct can_levels {
    1: i32 count
    2: map<CanId, double> can_levels
}

service CanLevels {
    can_levels get_cans_above_threshold(1: double percent_full)
    oneway void update_can_level(1: CanId can_id, 2: double percent_full)
}
```

The IDL begins with a typedef for CanIds. This will ensure that developers always use the same IDL type for the CanId semantic. Next we define a structure for a list of cans and fill levels, we can return this struct from functions. This is important because RPC functions can typically only return one thing per call, by creating a struct we can include as much data within that thing as we like. Also, structs such as this can be serialized and passed through messaging systems like NATS, RabbitMQ and SQS.

The last element defined in the IDL is the service, CanLevels. This service has two functions, the first returns the can levels for all cans over a given percent full threshold. The second function is a oneway function and can be called but never returns, similar to an async message. This function updates a given can's level.

Create a new working directory for the levels service under the trash-can application directory:

```
user@ubuntu:~$ cd ~/trash-can/

user@ubuntu:~/trash-can$ mkdir levels

user@ubuntu:~/trash-can$ cd levels
```

Now create the thrift IDL file from the above listing:

```
user@ubuntu:~/trash-can/levels$ vim levels.thrift

user@ubuntu:~/trash-can/levels$ cat levels.thrift

typedef i64 CanId

struct can_levels {
    1: i32 count
    2: map<CanId, double> can_levels
}

service CanLevels {
    can_levels get_cans_above_threshold(1: double percent_full)
    oneway void update_can_level(1: CanId can_id, 2: double percent_full)
}
```

We can now compile our Apache Thrift IDL for any number of languages. However we need to have access to the Apache Thrift IDL compiler to build our IDL into code. The `/usr/bin/thrift` compiler installed by system package managers like `yum` and `apt` is often old. Fortunately there is a Docker image on Docker Hub for Apache Thrift.

Use the command below to build your IDL with the thrift Docker image:

```
user@ubuntu:~/trash-can/levels$ docker container run -v "$PWD:/data" thrift thrift -o /data --gen py
/data/levels.thrift

Unable to find image 'thrift:latest' locally
latest: Pulling from library/thrift
762ae076e9a3: Pull complete
e4ab3623ebb0: Pull complete
Digest: sha256:000dc3e0bf67d7dfb97ff7a084bc4896a4b5a4353c33c5012633db746deb4165
Status: Downloaded newer image for thrift:latest
user@ubuntu:~/trash-can/levels$
```

```
user@ubuntu:~/trash-can/levels$ ls -la

total 16
drwxrwxr-x 3 user user 4096 Jan  7 05:49 .
drwxrwxr-x 4 user user 4096 Jan  7 05:47 ..
drwxr-xr-x 3 root root 4096 Jan  7 05:49 gen-py
-rw-rw-r-- 1 user user  256 Jan  7 05:47 levels.thrift
user@ubuntu:~/trash-can/levels$
```

This Docker command is a little complex so lets analyze it in parts:

- -v: maps a volume (directory) from the host, in our case the current working directory ($PWD), to a path in the container, `/data` in this case
- thrift: in the first case is the name of the image repository on Docker hub
- thrift: in the second case is the program inside the container we want to run
- -o: sets the thrift output directory
- --gen: tells thrift which language to emit, *py* for Python in our case
- `/data/levels.thrift` : is the path within the container to our IDL file

Because we have never run this image before it takes a moment to download from Docker Hub. Now that is has been pulled it will execute instantly next time we run it. The listing shows that thrift has created a `gen-py` directory with our service stubs for use by clients and servers.

Notice that the `gen-py` directory is owned by *root*. This will be a problem when we try to use the files in `gen-py` with our normal user. Container processes run as *root* by default. We need to change the owner of all of the files generated back to our user id before we proceed:

```
user@ubuntu:~/trash-can/levels$ sudo chown -R user:user gen-py
```

2. Create a Thrift Microservice

Before we can progress with Apache Thrift for Python we will need to install the Thrift support libraries for our chosen language. Use the `pip` command to install the Apache Thrift Python libraries at the system level:

```
user@ubuntu:~/trash-can/levels$ sudo -H pip install thrift

Collecting thrift
  Downloading thrift-0.10.0.zip (87kB)
    100% |████████████████████████████████| 92kB 605kB/s
Requirement already satisfied (use --upgrade to upgrade): six>=1.7.2 in /usr/local/lib/python2.7/dist-packages (from
thrift)
Building wheels for collected packages: thrift
  Running setup.py bdist_wheel for thrift ... done
  Stored in directory: /root/.cache/pip/wheels/e7/f1/d3/b472914d95caa1781fb29b1257b85808324b0bfd1838961752
Successfully built thrift
Installing collected packages: thrift
Successfully installed thrift-0.10.0
You are using pip version 8.1.1, however version 9.0.1 is available.
You should consider upgrading via the 'pip install --upgrade pip' command.
user@ubuntu:~/trash-can/levels$
```

This installs Thrift version 0.10.0. Now enter the following code for the levels microservice:

```
user@ubuntu:~/trash-can/levels$ vim levels-server.py

user@ubuntu:~/trash-can/levels$ cat levels-server.py
```

```
import sys
sys.path.append("gen-py")

from thrift.transport import TTransport
from thrift.transport import TSocket
from thrift.protocol import TBinaryProtocol
from thrift.server import TServer

from levels import CanLevels
from levels import ttypes

wcans = {}

class CanLevelsHandler:
    def get_cans_above_threshold(self, percent_full):
        return ttypes.can_levels(count=2, can_levels=wcans)

    def update_can_level(self, can_id, percent_full):
        wcans[can_id] = percent_full


handler = CanLevelsHandler()
proc = CanLevels.Processor(handler)
trans_ep = TSocket.TServerSocket(port=9095)
trans_fac = TTransport.TBufferedTransportFactory()
proto_fac = TBinaryProtocol.TBinaryProtocolFactory()
server = TServer.TThreadedServer(proc, trans_ep, trans_fac, proto_fac)

print("[Server] Started")
server.serve()
```

```
user@ubuntu:~/trash-can/levels$
```

The above version of our level server implements the CanLevels service interface with some test code. The updated levels are saved in a dictionary called `wcans`, however all requests for can levels return all cans at present. We can update the server and integrate it with the rest of our application later, for now we'll focus on the RPC bits.

The code creates what is called an I/O Stack in Apache Thrift. An Apache Thrift I/O Stack defines the way RPC is managed between clients and servers. In our example we use the following Apache Thrift library features:

- transport: this defines the way bits are transferred, we use the TSocket transport which send bits over TCP but we could easily change this to other transports for named-pipes, files, Unix sockets or event memory.
- protocol: this defines the serialization scheme used, we used binary but Apache Thrift also supports JSON and Compact

The trans_ep object is our listening port, the trans_fac object is our transport factory used to create a new transport for each client that connects and our proto_fac object is a protocol factory that creates a new serializer for each client that connects.

Our code uses the predefined TThreadedServer (Thrift types generally begin with a `T`) which allows us to serve multiple clients, creating a new thread for each connected client.

Run your new Thrift microservice:

```
user@ubuntu:~/trash-can/levels$ python levels-server.py

[Server] Started
```

Perfect!

3. Create a Thrift Microservice Client

To test our RPC microservice we will create a simple client which submits two levels and then retrieves them. Open a new terminal, change to the levels project directory and enter the following code into the `levels-client.py` file:

```
user@ubuntu:~$ cd ~/trash-can/levels/

user@ubuntu:~/trash-can/levels$ vim levels-client.py

user@ubuntu:~/trash-can/levels$ cat levels-client.py
```

```
import sys
sys.path.append("gen-py")
```

```
from levels import CanLevels

from thrift.transport import TSocket
from thrift.transport import TTransport
from thrift.protocol import TBinaryProtocol

trans = TSocket.TSocket("localhost", 9095)
trans = TTransport.TBufferedTransport(trans)
proto = TBinaryProtocol.TBinaryProtocol(trans)
client = CanLevels.Client(proto)

trans.open()
client.update_can_level(42, 0.85)
client.update_can_level(57, 0.89)
res = client.get_cans_above_threshold(0.8)
print("[Client] received: %d results" % (res.count))
print(res)
trans.close()
```

```
user@ubuntu:~/trash-can/levels$
```

This Python client simply connects to the Thrift server on its listening port (9095), using the server's serialization protocol and then begins to make normal function calls. The client sets two trash levels first then retrieves all of the levels above 0.8 (80%.)

Try running your client while the server is running in another terminal:

```
user@ubuntu:~/trash-can/levels$ python levels-client.py

[Client] received: 2 results
can_levels(count=2, can_levels={57: 0.89, 42: 0.85})
user@ubuntu:~/trash-can/levels$
```

While this may seem like a bit more work than our REST service it is probably about an order of magnitude faster. It is also complete with IDL to document the interface contract. Notice we did not have to do any JSON to int/float conversions, Thrift knows what the correct types are and ensures that only those types are accepted in the interface. Also consider the fact that you would probably need to document your REST API using OAI or RAML, mirroring the IDL effort made here with Thrift.

REST is king for request/response in public interfaces but many large internet firms use Thrift (Facebook, Twitter, Evernote among them) or gRPC/ProtoBufs (Google, Docker, Square, among them) for internal services that are under heavy load or are latency sensitive.

Now create a Git repo for your level code:

```
user@ubuntu:~/trash-can/levels$ git init

Initialized empty Git repository in /home/user/trash-can/levels/.git/
```

Setup a `.gitignore` for all of the thrift generated code directories:

```
user@ubuntu:~/trash-can/levels$ vim .gitignore

user@ubuntu:~/trash-can/levels$ cat .gitignore

gen-*/
user@ubuntu:~/trash-can/levels$
```

Now commit your code:

```
user@ubuntu:~/trash-can/levels$ git status

On branch master

Initial commit

Untracked files:
  (use "git add <file>..." to include in what will be committed)

        .gitignore
        levels-client.py
        levels-server.py
        levels.thrift

nothing added to commit but untracked files present (use "git add" to track)
```

```
user@ubuntu:~/trash-can/levels$  git add .
```

If you did not globally set your Git configuration information, then set again.

```
user@ubuntu:~/trash-can/levels$ git config user.name "Bat Man"

user@ubuntu:~/trash-can/levels$ git config user.email "me@example.com"
```

Now commit.

```
user@ubuntu:~/trash-can/levels$ git commit -m "initial IDL commit"

[master (root-commit) ca987ab] initial IDL commit
 4 files changed, 62 insertions(+)
 create mode 100644 .gitignore
 create mode 100644 levels-client.py
 create mode 100644 levels-server.py
 create mode 100644 levels.thrift
user@ubuntu:~/trash-can/levels$
```

4. Polyglot Programming

One of the most important features of Apache Thrift is its ability to produce interfaces that can be used across a wide range of programming languages. To test this let's rebuild our Python client in Ruby!

First we need to generate the interface stubs for our IDL in Ruby:

```
user@ubuntu:~/trash-can/levels$ docker container run -v "$PWD:/data" thrift thrift -o /data --gen rb
/data/levels.thrift

user@ubuntu:~/trash-can/levels$
```

This produces a `gen-rb` directory with all of the needed Ruby code to call our server.

Now code up a quick Ruby client:

```
user@ubuntu:~/trash-can/levels$ vim levels-client.rb

user@ubuntu:~/trash-can/levels$ cat levels-client.rb
```

```
require 'thrift'
$:.push('gen-rb')
require 'can_levels'

begin
  trans_ep = Thrift::Socket.new(ARGV[0], ARGV[1])
  trans_buf = Thrift::BufferedTransport.new(trans_ep)
  proto = Thrift::BinaryProtocol.new(trans_buf)
  client = CanLevels::Client.new(proto)

  trans_ep.open()
  client.update_can_level(42, 0.85)
  client.update_can_level(57, 0.89)
  res = client.get_cans_above_threshold(0.8)
  puts '[Client] received: ' + res.count.to_s
  puts res.inspect
  trans_ep.close()
rescue Thrift::Exception => tx
  print 'Thrift::Exception: ', tx.message, "\n"
end
```

```
user@ubuntu:~/trash-can/levels$
```

This program is functionally identical to the Python client. It uses the Ruby require statement to pull in the thrift client library and the `can_levels.rb` source generated for our interface. We need Ruby and the Thrift Ruby Gem (library) to run our new client. Rather than installing Ruby on the host lab system we can just run our program from within a Ruby container. We can run a Ruby container, install thrift and then execute our program in a few commands, try it:

```
user@ubuntu:~/trash-can/levels$ docker container run -v "$PWD:/app" -w "/app" ruby bash -c "gem install thrift &&
ruby levels-client.rb 172.16.151.239 9095"

Unable to find image 'ruby:latest' locally
latest: Pulling from library/ruby
723254a2c089: Pull complete
abe15a44e12f: Pull complete
409a28e3cc3d: Pull complete
503166935590: Pull complete
0f46f97746e4: Pull complete
9d641d922d2f: Pull complete
891f591c9621: Pull complete
825fab4554b7: Pull complete
Digest: sha256:090577ac73868da6c74a2c5d716e67189e8f6c597b1d4bfbe61e8a68c829c02e
Status: Downloaded newer image for ruby:latest
Building native extensions. This could take a while...
Successfully installed thrift-0.10.0.0
1 gem installed
[Client] received: 2
<Can_levels count:2, can_levels:{57: 0.89, 42: 0.85}>
user@ubuntu:~/trash-can/levels$
```

The `docker container run` command used, maps our current directory into the container so that we can run our program. The `-it` switch creates a tty (t) in the container and connects our input (i) to it. If we just run a ruby container it will execute the IRB Ruby interpreter. We needed to install thrift before running our program so we specified 'bash' as the program to run.

We tell the ruby client to target our host IP where we have the Python server running. You can discover your host IP address using the following command on the host (not in the container!):

```
user@ubuntu:~/trash-can/levels$ ip a s

1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state UNKNOWN group default qlen 1
...

2: ens33: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc pfifo_fast state UP group default qlen 1000
    link/ether 00:0c:29:2d:33:92 brd ff:ff:ff:ff:ff:ff
    inet 172.16.151.239/24 brd 172.16.151.255 scope global ens33
       valid_lft forever preferred_lft forever
    inet6 fe80::20c:29ff:fe2d:3392/64 scope link
       valid_lft forever preferred_lft forever
3: docker0: <NO-CARRIER,BROADCAST,MULTICAST,UP> mtu 1500 qdisc noqueue state DOWN group default
...

user@ubuntu:~/trash-can/levels$
```

Ignore loopback (lo) and the Docker bridge (docker0), or you can try:

```
user@ubuntu:~/trash-can/levels$ ip route get 1.1.1.1

1.1.1.1 via 192.168.131.2 dev ens33  src 192.168.131.129
    cache

user@ubuntu:~/trash-can/levels$
```

That's it, in seconds we have a high performance polyglot microservice solution up and running.

5. Interface evolution

Many early RPC systems did not support changes to a given interface. Once an interface was defined it could not be modified without breaking all existing clients and servers. Modern RPC systems like Apache Thrift and Protocol Buffers allow many modifications to be made to a given interface without breaking backwards compatibility. This is a very important trait in a dynamic high turn over microservices world.

Recall our original IDL:

```
typedef i64 CanId

struct can_levels {
    1: i32 count
    2: map<CanId, double> can_levels
}

service CanLevels {
    can_levels get_cans_above_threshold(1: double percent_full)
    oneway void update_can_level(1: CanId can_id, 2: double percent_full)
```

```
}
```

Now imagine we realize that some of our clients need to be able to determine the date that a given can_levels struct was created on, yet other clients are happy with can_levels as is. What to do?

With Apache Thrift we can extend the struct without breaking backwards compatibility. Modify the IDL to include a new Julien date field:

```
user@ubuntu:~/trash-can/levels$ vim levels.thrift

user@ubuntu:~/trash-can/levels$ cat levels.thrift
```

```
typedef i64 CanId

struct can_levels {
    1: i32 count
    2: map<CanId, double> can_levels
    3: i64 date
}

service CanLevels {
    can_levels get_cans_above_threshold(1: double percent_full)
    oneway void update_can_level(1: CanId can_id, 2: double percent_full)
}
```

```
user@ubuntu:~/trash-can/levels$
```

If your python server is running stop it, and update the server to return the new date field:

```
[Server] Started

^C

Traceback (most recent call last):
  File "levels-server.py", line 30, in <module>
    server.serve()
  File "/usr/local/lib/python2.7/dist-packages/thrift/server/TServer.py", line 106, in serve
    client = self.serverTransport.accept()
  File "/usr/local/lib/python2.7/dist-packages/thrift/transport/TSocket.py", line 189, in accept
    client, addr = self.handle.accept()
  File "/usr/lib/python2.7/socket.py", line 206, in accept
    sock, addr = self._sock.accept()
KeyboardInterrupt

user@ubuntu:~/trash-can/levels$ vim levels-server.py

user@ubuntu:~/trash-can/levels$ cat levels-server.py
```

```
import sys
sys.path.append("gen-py")

from thrift.transport import TTransport
from thrift.transport import TSocket
from thrift.protocol import TBinaryProtocol
from thrift.server import TServer

from levels import CanLevels
from levels import ttypes

wcans = {}

class CanLevelsHandler:
    def get_cans_above_threshold(self, percent_full):
        return ttypes.can_levels(count=2, can_levels=wcans, date=37465)

    def update_can_level(self, can_id, percent_full):
        wcans[can_id] = percent_full


handler = CanLevelsHandler()
proc = CanLevels.Processor(handler)
trans_ep = TSocket.TServerSocket(port=9095)
```

```
trans_fac = TTransport.TBufferedTransportFactory()
proto_fac = TBinaryProtocol.TBinaryProtocolFactory()
server = TServer.TThreadedServer(proc, trans_ep, trans_fac, proto_fac)

print("[Server] Started")
server.serve()
```

```
user@ubuntu:~/trash-can/levels$
```

The only change we have made here is the addition of the date field in our can_levels constructor, returned from the get_cans_above_threshold() method.

Now generate the new stubs for the updated IDL and rerun your server:

```
user@ubuntu:~/trash-can/levels$ docker container run -v "$PWD:/data" --user $(id -u) thrift thrift -o /data --gen py
/data/levels.thrift

user@ubuntu:~/trash-can/levels$ python levels-server.py

[Server] Started
```

To test the new functionality lets run the Python client in another shell:

```
user@ubuntu:~/trash-can/levels$ python levels-client.py

[Client] received: 2 results
can_levels(count=2, date=37465, can_levels={57: 0.89, 42: 0.85})

user@ubuntu:~/trash-can/levels$
```

As you can see the date field is now included in the results displayed. This is as expected, however the particularly important feature of this interface is that we can still call the server with the only interface stubs successfully. For example, rerun your Ruby client without rebuilding the Ruby stubs for the new IDL:

```
user@ubuntu:~/trash-can/levels$ docker container run -v "$PWD:/app" -w "/app" ruby bash -c "gem install thrift &&
ruby levels-client.rb 172.16.151.239 9095"

Building native extensions. This could take a while...
Successfully installed thrift-0.10.0.0
1 gem installed
[Client] received: 2
<Can_levels count:2, can_levels:{57: 0.89, 42: 0.85}>
user@ubuntu:~/trash-can/levels$
```

As you can see, the old Ruby code does not know anything about the data field in the response structure and does not display it. However, we know from the server code that it is returned every time. This means that the Ruby client is capable of ignore fields it does not recognize. This is one of the many evolution features supported by Apache Thrift that allows you to evolve interfaces overtime while supporting a range of clients using various iterations of that interface.

6. Cleanup

Stop your Thrift server and commit the new code you have created.

Control + C will stop the server:

```
user@ubuntu:~/trash-can/levels$ python levels-server.py

[Server] Started
^C

Traceback (most recent call last):
  File "levels-server.py", line 29, in <module>
    server.serve()
  File "/usr/local/lib/python2.7/dist-packages/thrift/server/TServer.py", line 109, in serve
    client = self.serverTransport.accept()
  File "/usr/local/lib/python2.7/dist-packages/thrift/transport/TSocket.py", line 177, in accept
    client, addr = self.handle.accept()
  File "/usr/lib/python2.7/socket.py", line 206, in accept
    sock, addr = self._sock.accept()
KeyboardInterrupt
```

Now add and commit your new work.

```
user@ubuntu:~/trash-can/levels$ git status

On branch master
Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git checkout -- <file>..." to discard changes in working directory)

        modified:   levels-server.py
        modified:   levels.thrift

Untracked files:
  (use "git add <file>..." to include in what will be committed)

        levels-client.rb

no changes added to commit (use "git add" and/or "git commit -a")

user@ubuntu:~/trash-can/levels$ git add levels-server.py levels.thrift levels-client.rb

user@ubuntu:~/trash-can/levels$ git commit -m "evolved interface"

[master f9b2a0e] evolved interface
 3 files changed, 22 insertions(+), 1 deletion(-)
 create mode 100644 levels-client.rb
user@ubuntu:~/trash-can/levels$
```

Note that per the **12 factor app** model we are creating a single git repo for each services we create (including its test clients).

- https://12factor.net/

Food for Thought

Given that our service is stateful, tracking trash can levels, we should probably ask ourselves:

- Are either of the methods exposed by our interface safe?
- Are either of the methods exposed by our interface idempotent?
- Do you think that it would be important to document the presence or absence of such attributes function by function?

Apache Thrift IDL (much like Java Doc, PyDoc and others) allows you to embed interface semantic documentation in comments which can then be used to generate an html based interface documentation website. One might argue that our interface contract is not complete until such semantics are codified and documented.

[OPTIONAL] Challenge Step

Create a REST version of the service in this lab then create a client that makes 1 million REST "update can level" calls in a loop. Run this under the `time` command to see how long it takes. Then try the same thing with Apache Thrift. Use Python in both cases, you can use Flask (per lab 1) to implement the REST api.

Congratulation you have successfully completed the lab!