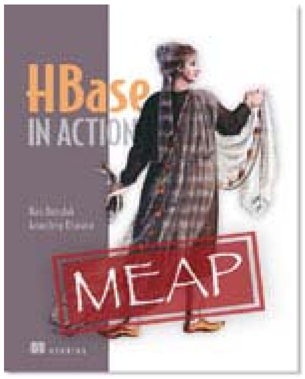
MANNING PUBLICATIONS



Hadoop is written in Java; HBase is written in Java; the stock HBase client is written in Java. There's only one problem: you don't use Java. You don't even like the JVM. You still want to use HBase. Now what? HBase provides you with alternate clients (both JVM-based as well as those that don't require the JVM) that you can use when Java is not an option. When it comes to doing anything more than exploring a cluster, you'll want to use the Thrift gateway. That's what this article based on chapter 6 from HBase in Action is all about.

Using the HBase Thrift Gateway from Python

When you live in the world beyond Java, the most common way to interact with HBase is via Apache Thrift[1]. Thrift is a language and set of tools for generating language-agnostic services. Thrift has an Interface Definition Language (IDL) for describing services and objects. It provides a networking protocol for communication between processes using those object and service definitions. Thrift uses the IDL you describe to generate code for you in your favorite languages. Using that code, you can write applications that communicate with each other using the lingua franca provided by Thrift.

HBase ships a Thrift IDL describing a service layer and set of objects. It also provides a service implementing that interface. In this section, you'll generate the Thrift client library for interacting with HBase. You'll use that client library to interact with HBase from Python, completely outside of Java and the JVM. We chose Python because its syntax is approachable to both novice and seasoned programmers. The same approach applies for interacting with HBase from your favorite language. At the time of this writing, Thrift supports 14 different languages.

This API is… different.

In part because of Thrift's ambitions to support so many languages, its IDL is relatively simple. It lacks features common in many languages, such as object inheritance. As a result, the HBase interface via Thrift is slightly different from the Java client API we've explored thus far.

There is an effort[2] under way to bring the Thrift API closer to Java, but it remains a work in progress. An early version is available with HBase 0.94, but it lacks some key features like Filters and access to Coprocessor Endpoints.[3] The API we're exploring here will be deprecated upon completion of this effort.

The beauty of using the Thrift API is that it's the same for all languages. Whether you're using PHP, Perl, or C#, the interface is always the same. Additional HBase feature support added to the Thrift API is additional feature support available everywhere.

[1] Originally a project out of Facebook, Thrift is now an Apache project. Learn more at http://thrift.apache.org/
[2] For more details, see the JIRA ticket "Thrift server to match the new Java API": https://issues.apache.org/jira/browse/HBASE-1744
[3] Well, you can, but you have to modify the Hbase.thrift file for each Endpoint you want to expose. For details, see https://issues.apache.org/jira/browse/HBASE-5600.

The Thrift gateway is not without limitations. Notably, it suffers the same throughout challenges as the REST gateway. All client connections are funneled through a single machine that communicates with the cluster on their behalf. Because the Thrift client opens a connection to a single instance for the duration of its session, clustering Thrift gateways is easier than REST. Still, portions of the API are stateful, so a disconnected client will lose access to allocated resources when it opens a new connection. Figure 1 illustrates the network topology of a Thrift gateway deployment.

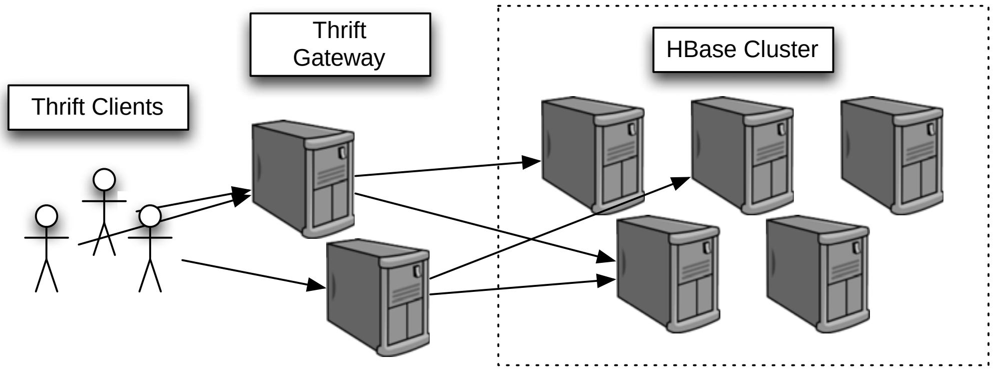


Figure 1 A Thrift gateway deployment. All clients are funneled through the gateway, greatly reducing client throughput. Clustering is easier because the Thrift protocol is session-based.

Python is our language for this exercise, so let's begin by creating a Python project, complete with an HBase client library. The final code for this project is available in our Github repository, https://github.com/hbaseinaction/twitbase.py.

Generate the HBase Thrift client library for Python

To build the Thrift client library, you're going to need Thrift. Thrift isn't packaged yet, so you'll need to build it from source. On a Mac, that's easy because Thrift is available through homebrew[4]:

```
$ brew install thrift
...
==> Summary
/usr/local/Cellar/thrift/0.8.0: 75 files, 5.4M, built in 2.4 minutes
```

For those running other platforms, you'll need to build Thrift manually. See the Thrift Requirements[5] doc for details specific to your platform.

Once that's done, verify your build is alive and well:

```
$ thrift -version
Thrift version 0.8.0
```

If you want a Thrift client, you'll need to grab the source:

```
$ wget http://www.apache.org/dist/hbase/hbase-0.92.1/hbase-0.92.1.tar.gz
...
Saving to: `hbase-0.92.1.tar.gz'
$ tar xzf hbase-0.92.1.tar.gz
```

With the HBase source in hand and Thrift installed, you're interested in one file: src/main/resources/org/apache/hadoop/hbase/thrift/Hbase.thrift. That's the IDL file that describes the HBase

[4] Homebrew is "The missing package manager for OS X". Find out more at http://mxcl.github.com/homebrew/
[5] Apache Thrift Requirements: http://thrift.apache.org/docs/install/

service API and related objects. Go ahead and skim through it—the Thrift IDL is quite legible. Now you have everything you need to generate the Python client.

Start by creating a project directory for yourself and generating the HBase client bindings:

```
$ mkdir TwitBase.py
$ cd TwitBase.py
$ thrift -gen py ../hbase-0.92.1/.../Hbase.thrift
$ mv gen-py/* .
$ rm -r gen-py/
```

You've created a project called TwitBase.py and generated the HBase python library. Thrift generated its code into a subdirectory called gen-py. By moving all that up into your project, you'll be able to easily import that code into your application. Have a look at what was generated:

```
$ find .
./__init__.py
./hbase
./hbase/__init__.py
./hbase/constants.py
./hbase/Hbase-remote
./hbase/Hbase.py
./hbase/ttypes.py
```

You also need to install the Thrift Python library. These are the core components common across all Thrift services used from Python, so you can install them globally:

```
$ sudo easy_install thrift==0.8.0
Searching for thrift==0.8.0
Best match: thrift 0.8.0
...
Finished processing dependencies for thrift
```

Alternately, this library is also part of the source you compiled. You could copy these files into your project like you did with the HBase client. From within the TwitBase.py directory:

```
$ mkdir thrift
$ cp -r ../thrift-0.8.0/lib/py/src/* ./thrift/
```

Verify that everything worked as expected. Launch Python and import both Thrift an HBase libraries. No output means everything is as it should be:

```
$ python
Python 2.7.1 (r271:86832, Jul 31 2011, 19:30:53)
...
>>> import thrift
>>> import hbase
```

Be sure to run these commands from within the TwitBase.py directory, or the `import` statements will fail. With our client library ready to go, let's start the server component.

Launch the HBase Thrift service

The server component ships with HBase, so it doesn't involve all of the setup required by the client library. Launch the Thrift service the same way you launch the shell, using the `hbase` command:

```
$ hbase thrift
...
usage: Thrift [-b <arg>] [-c] [-f] [-h] [-hsha | -nonblocking |
       -threadpool]  [-p <arg>]
 -b,--bind <arg>   Address to bind the Thrift server to. Not supported by
                   the Nonblocking and HsHa server [default: 0.0.0.0]
 -c,--compact      Use the compact protocol
 -f,--framed       Use framed transport
 -h,--help         Print help information
 -hsha             Use the THsHaServer. This implies the framed transport.
 -nonblocking      Use the TNonblockingServer. This implies the framed
                   transport.
 -p,--port <arg>   Port to bind to [default: 9090]
 -threadpool       Use the TThreadPoolServer. This is the default.
```

Make sure HBase is up and running, and then launch the Thrift service. The default settings should be fine:

```
$ hbase thrift start
...
ThriftServer: starting HBase ThreadPool Thrift server on /0.0.0.0:9090
```

With both the client and server ready, it's time to test them out. Open a terminal window in your TwitBase.py project directory and once again launch Python:

```
$ python
Python 2.7.1 (r271:86832, Jul 31 2011, 19:30:53)
...
>>> from thrift.transport import TSocket
>>> from thrift.protocol import TBinaryProtocol
>>> from hbase import Hbase
>>> transport = TSocket.TSocket('localhost', 9090)
>>> protocol = TBinaryProtocol.TBinaryProtocol(transport)
>>> client = Hbase.Client(protocol)
>>> transport.open()
>>> client.getTableNames()
['followers', 'twits', 'users']
```

It took us a little while to get here, but it all works! Now we can get down to business.

Scan the TwitBase users table

Before you start writing code, let's explore at the interpreter a little more to get a feel for the API. You're interested in scanning the `users` table, so let's start with a scanner. Examining the `Hbase.IFace` class in Hbase.py, it looks like `scannerOpen()` is the simplest method. It returns a scanner ID we can call on the Thrift server. Let's try it out:

```
>>> columns = ['info:user','info:name','info:email']
>>> scanner = client.scannerOpen('users', '', columns)
>>> scanner
14
```

Here we've asked for an unbounded scanner over the `'users'` table, returning only three qualifiers from the `'info'` column. It happens to have the ID 14. Let's take the first row and see what we get:

```
>>> row = client.scannerGet(scanner)
>>> row
[TRowResult(
  columns={'info:email': TCell(timestamp=1338530917411,
                             value='samuel@clemens.org'),
           'info:name': TCell(timestamp=1338530917411,
                            value='Mark Twain'),
           'info:user': TCell(timestamp=1338530917411,
                            value='TheRealMT')},
  row='TheRealMT')]
```

`scannerGet()` returns a list with a single `TRowResult`. That row has a `columns` field that is a dictionary from column qualifier to a `TCell` instance.

Now that we know what we're working with, let's build out a class to wrap up all these details. Call that helper class `TwitBaseConn` and give it a constructor to hide all these Thrift connection details. Also, be sure to `close()` anything you `open()`:

```
class TwitBaseConn(object):
    def __init__(self, host="localhost", port=9090):
        transport = TSocket.TSocket(host, port)
        self.transport = TTransport.TBufferedTransport(transport)
        self.protocol = TBinaryProtocol.TBinaryProtocol(self.transport)
        self.client = Hbase.Client(self.protocol)
        self.transport.open()

    def close(self):
        self.transport.close()
```

This defines a default constructor that will connect to the Thrift service running locally. It also adds an extra layer to the networking stack, wrapping the socket in a buffer. Now add a method to handle scanning rows from the `users` table:

```
def scan_users(self):
    columns = ['info:user','info:name','info:email']
    scanner = self.client.scannerOpen('users', '', columns)
    row = self.client.scannerGet(scanner)
    while row:
        yield row[0]
        row = self.client.scannerGet(scanner)
    self.client.scannerClose(scanner)
```

That takes care of reading rows and cleaning up after the scanner. Those rows are full of Thrift library details, though, so let's add another method to pull out the data we want:

```
def _user_from_row(self, row):
    user = {}
    for col,cell in row.columns.items():
        user[col[5:]] = cell.value
    return "<User: {user}, {name}, {email}>".format(**user)
```

This method loops through the `TCells` and creates a string from their contents. Update `scan_users()` to call this method instead of returning the raw rows:

```
def scan_users(self):
    columns = ['info:user','info:name','info:email']
    scanner = self.client.scannerOpen('users', '', columns)
    row = self.client.scannerGet(scanner)
    while row:
        yield self._user_from_row(row[0])
        row = self.client.scannerGet(scanner)
    self.client.scannerClose(scanner)
```

Great! All that's left is to wrap it up in a `main()` and we can give it a spin. The final TwitBase.py script looks like listing 1.

Listing 1 TwitBase.py: connecting to TwitBase from Python via Thift

```
#! /usr/bin/env python

import sys

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

from hbase import Hbase
from hbase.ttypes import *
```

```
usage = """TwitBase.py action ...
  help - print this messsage and exit
  list - list all installed users."""

class TwitBaseConn(object):
    def __init__(self, host="localhost", port=9090):
        transport = TSocket.TSocket(host, port)
        self.transport = TTransport.TBufferedTransport(transport)
        self.protocol = TBinaryProtocol.TBinaryProtocol(self.transport)
        self.client = Hbase.Client(self.protocol)
        self.transport.open()

    def close(self):
        self.transport.close()

    def _user_from_row(self, row):
        user = {}
        for col,cell in row.columns.items():
            user[col[5:]] = cell.value
        return "<User: {user}, {name}, {email}>".format(**user)

    def scan_users(self):
        columns = ['info:user','info:name','info:email']
        scanner = self.client.scannerOpen('users', '', columns)
        row = self.client.scannerGet(scanner)
        while row:
            yield self._user_from_row(row[0])
            row = self.client.scannerGet(scanner)
        self.client.scannerClose(scanner)

def main(args=None):
    if args is None:
        args = sys.argv[1:]

    if len(args) == 0 or 'help' == args[0]:
        print usage
        raise SystemExit()

    twitBase = TwitBaseConn()

    if args[0] == 'list':
        for user in twitBase.scan_users():
            print user

    twitBase.close()

if __name__ == '__main__':
    main()
```

The `main()` is super simple. It opens the connection, calls the scan, prints the results, and closes the connection again. With Python there's nothing to compile. You do need to make the file executable though, which is a one-liner:

```
$ chmod a+x TwitBase.py
```

Now you're ready to try it out:

```
$ ./TwitBase.py list
<User: TheRealMT, Mark Twain, samuel@clemens.org>
<User: GrandpaD, Fyodor Dostoyevsky, fyodor@brothers.net>
<User: SirDoyle, Sir Arthur Conan Doyle, art@TheQueensMen.co.uk>
<User: HMS_Surprise, Patrick O'Brian, aubrey@sea.com>
```

Nicely done! Now you have everything you need to start building HBase applications in Python.

Summary

The decision to deploy HBase ties to you to the JVM, at least on the server side. That decision, however, does not restrict your client application options. One of the options outside the JVM is Thrift. Thrift provides some measure of language-agnostic API definition. The way it does that is by using Interface Definition Language (IDL) for describing services and objects. Thrift uses the IDL you describe to generate code for you in your favorite languages. Using that code, you can write applications that communicate with each other using the lingua franca provided by Thrift.

Last updated: August 17, 2012