Implementing a Cross-Language Interface to Multiple Distributed File Systems Using Thrift *v. 0.5.3*

Collin Bennett[1], Robert Grossman[1,2] and Jonathan Seidman[1]
[1]Open Data Group
[2]University of Illinois at Chicago

Open Cloud Consortium Technical Report TR-09-03

www.opencloudconsortium.org

23 April 2009
Revised 1 June 2009

Copyright Open Cloud Consortium 2009

Abstract

Multiple open-source cloud storage systems are currently in use, such as Hadoop, CloudStore, and Sector. Although all of these systems implement a distributed file system providing reliable storage of large data sets, each uses its own client interface to access this data. Additionally, using the client interfaces requires coding in the implementation language of each system. For example, Sector and CloudStore only support C++ for implementing clients, while Hadoop only supports Java and C++. This means developers are restricted from using their language of choice when implementing clients. In this paper, we look at one approach to solving these problems using Thrift to implement a cross-language interface to multiple distributed file systems. With Thrift we're able to provide a common interface to multiple cloud storage systems which can be accessed from multiple languages.

After discussing the implementation, we present the results of performance testing of some basic file operations using the Thrift service. These tests compare the performance of reading data from the file systems using native clients vs. Thrift clients.

The following table summarizes the results of this performance testing:

	1 Thrift Client vs. 1 Native Client			**20 Thrift Clients vs. 20 Native Clients**		
	10 Million Total Records			200 Million Total Records		
	Native	**Thrift**	**Overhead**	**Native**	**Thrift**	**Overhead**
Hadoop	3m 55s	5m 36s	1.43 x	4m 10s	11m 11s	2.68 x
Sector	6m 51s	9m 29s	1.38 x	17m 6s	19m 7s	1.12 x

Copyright Open Cloud Consortium 2009

Introduction

We use Thrift to implement a common interface to multiple distributed file systems. Distributed file systems are storage systems designed to reliably store very large files on a multiple-node cluster. This allows the file system to be distributed across a local or wide area network but appear as a single file system to clients. Distributed file systems provide reliability by replicating files to multiple nodes of the cluster, providing redundancy if one of the nodes becomes unavailable.

Thrift is a framework for implementing cross-language interfaces to services. Thrift uses an Interface Definition Language to define interfaces, and uses that file to generate stub code to be used in implementing RPC clients and servers that can be used across languages. Using Thrift allows us to implement a single interface that can be used with different languages to access different underlying systems. The specific distributed file systems targeted are the Hadoop distributed file system (HDFS) and Sector. For each of these systems we implemented a custom Thrift server which utilizes the underlying systems client API. The Hadoop implementation uses the Hadoop FileSystem Java API to implement the server side code, and the Sector implementation uses the Sector client API, which is implemented in C++. Clients can be written in any Thrift supported language to access the Hadoop Java server or the Sector C++ server without requiring any code changes. The interface defines common filesystem commands such as `mkdir`, `move`, `remove`, `read`, `write`, etc.

Implementation

As a basis for our implementation we used the interface defined by the HDFS-APIs project, which is a Thrift interface to Hadoop to allow accessing the Hadoop filesystem from non-Java clients. The HDFS-APIs code is a contributed project included with the Hadoop distribution. We modified this interface to be more generally applicable to distributed file systems other than Hadoop.

As mentioned previously, Thrift uses an Interface Definition Language to define services. A Thrift IDL file can contain definitions for functions, structs, exceptions, etc. An example of a function definition in our Thrift IDL looks like:

```
/**
 * Read data from file.
 *
 * offset is file offset to start reading from.
 * len is length of data to read.
 */
string read( 1:DfsHandle dfsHandle, 3:i64 offset, 4:i64 len )
    throws ( 1:DfsServiceIOException ex ),
```

This defines a function to read from the distributed file system connected to the Thrift Server. There is nothing, at this layer, which is specific to an application or programming language. The Thrift compiler uses this function definition to generate language specific interfaces.

In Java the resulting method signature would look like:

```
/**
 * Read data from file.
 *
 * offset is file offset to start reading from.
 * len is length of data to read.
 */
public String read(DfsHandle dfsHandle, long offset, long len)
    throws DfsServiceIOException, TException;
```

And in C++:

```
void read(std::string& _return, const DfsHandle& dfsHandle,
    const int64_t offset, const int64_t len)
```

Thrift generates code providing full support for RPC, so we simply had to write server code to implement the generated language specific interfaces and then write clients to call the server code. All messages passing between clients and servers are handled transparently by the Thrift framework. The following excerpt from a C++ client provides an example of calling the `copyFromLocalFile` method on the server:

```
// log into the application
DfsServiceClient client(protocol);
ClientHandle cl;
client.init( cl, "sector://localhost:6000" );
client.login( cl, "test", "xxx" );
// load a local file into the DFS
client.copyFromLocalFile( "/tmp/data/test.dat", "/test.dat" );
client.logout( cl );
client.closeDfs( cl );
```

And the following is the equivalent code from a Java client:

```
// log into the application
DfsService.Client client = new DfsService.Client(protocol);
ClientHandle chandle = null;
chandle = client.init(null);
client.login(chandle, "test", "xxx");
// load a local file into the DFS
client.copyFromLocalFile( "/tmp/data/test.dat", "/test.dat" );
client.logout(chandle);
client.clseDfs( chandle );
```

As noted above, using Thrift allows clients written in any Thrift supported language to access any of the implemented servers. For example, the same Java client can access either the C++ Sector server or the Java Hadoop server and execute any of the functions supported by the interface without requiring any changes. This applies to clients written in any of the Thrift supported languages, which includes:

- C++
- C#
- Cocoa
- Erlang
- Haskell
- Java
- OCaml
- Perl
- PHP
- Python
- Ruby
- Smalltalk

Tests

The scenario we used for testing was intended to represent a real-world use case – specifically reading previously loaded data from a file system and writing it to local disk. These tests were intended to mimic multiple users running client applications pulling data from the cloud in order to use it for local processing. The initial tests were run using clients implemented using the native API's of each test systems. Following this tests were run using Thrift clients. The tests first timed a single client reading a file from the cloud and writing it to local disk. We then scaled this to 20 clients.

The data used was simulated data created by MalGen. Each file contained 10 million 100 byte records. The Hadoop and Sector masters both resided on the master node of the rack. The Thrift servers were run from an unused slave node on the rack. The Thrift clients were run on the slave nodes.

Test Configuration

Tests were performed on a multi-node cluster on a single rack with the following configuration:

- Master node:
  - 2 x dualcore Xeon processors
  - 16 GB RAM
- Slave nodes:
  - 2 x dualcore Opteron processors
  - 12 GB RAM

The following configuration was used for Hadoop and Sector:

- File replication was set to `1`.
- The Hadoop Heap Size was `8000` MB (Sector does not explicitly set memory).
- The block size for Hadoop was `128` MB (Sector does not split files into blocks).
- The Sector version used for these tests was `1.19`.
- The Hadoop version used for theses tests was `0.18.3`.
- Java 1.6 was used at to compile and run the Hadoop-related components.

Test Results

To establish a base line, we first ran a single read from the Distributed Files Systems and wrote it to local disk without using Thrift. In practice, there might have been more efficient ways to code these 'native' applications by taking advantages of internal code or hooks, but to allow the comparison to focus on the overhead incurred by using Thrift, we attempted to replicate the code used on the Server side of the Thrift applications. We then ran these native calls on `20` nodes.

	1 Client	20 Clients	1 Rack Native Scaling Overhead
Total # Records Read	10 M	200 M	
Hadoop	3m 55s	4m 10s	1.06 x
Sector	6m 51s	17m 6s	2.49 x

Table 1. Native calls without Thrift

Ideally, as we grow the number of Thrift clients, we would see a similar amount of overhead.

We then ran the tests using Thrift. We first ran a single Thrift client against all 20 Hadoop and Sector nodes. We then scaled this to the rack. The results are below:

	1 Client	20 Clients	1 Rack Thrift Scaling Overhead
Total # Records Read	10 M	200 M	
Hadoop	5m 36s	11m 11s	1.99 x
Sector	9m 29s	19m 7s	2.02 x

Table 2. Reading from the Cloud and writing the data to local disk.

Using Thrift actually improves the performance of Sector as it scales to a rack and while the performance is not as good with Hadoop, Thrift's overhead is consistent across the two applications. For Hadoop and Sector, the changes from using Thrift and from scaling to a rack are given below:

The columns show the overhead incurred by scaling from a single client to twenty across a rack both natively and when using Thrift. The rows show the overhead incurred by using Thrift.

Hadoop

	Native	Thrift	OVERHEAD
1 client	3m 55s	5m 36s	***1.43 x***
20 clients	4m 10s	11m 11s	***2.68 x***
OVERHEAD	***1.06 x***	***1.99 x***	

Sector

	Native	Thrift	OVERHEAD
1 client	6m 51s	9m 29s	***4.38 x***
20 clients	17m 6s	19m 7s	***1.12 x***
OVERHEAD	***2.49 x***	***2.02 x***	

Analysis of Test Results

The test results show that Thrift adds only a small amount of overhead. This is encouraging and the tests should b expanded to

larger files, more nodes, and multiple racks over a Wide Area Network. We did discover some areas that need fine tuning in the implementations we tested.

Server Side

For both the Hadoop and Sector tests there was overhead in reading multiple files through the Thrift service. This may be due to the fact that the uploads are being routed through a multi-threaded server, while the native reads are performed through independent clients running on separate hardware nodes.

Also, the Thrift Server communicating with Hadoop is coded in Java and the server talking to Sector is in C++. Multi-threading is very different in Java than C++ and there is no reason to assume that the Thrift server will perform comparably across the different languages in which it can be implemented.

Client Side

On the client side, there are two main factors in governing the performance:

1. The size of the buffers used
2. The blocking by the read (remote) and write (local) calls.

The files are too large to bring over in one piece. We use `1` GB files, which are on the small side for cloud data, but they are too large to transfer as a single stream. The data is read in bytes into a buffer and then transferred via Thrift from the Distributed File System to the client. The size of the buffer should be optimized for the amount of data and the network. In all tests, buffers of `4096` bytes were used.

The data written to disk does not have to proceed in step with the transfer. Depending on the write performance, the client may save the transferred data in memory and write to disk after every `x` calls to the server.

Neither Thrift client used in the testing is multi-threaded. Adding threading to C++ is more complicated than with Java, but we should take advantage of Java's Runnable Interface and Pthreads to make the reading from the Distributed File System via Thrift a separate thread from the writing to the local disk. As it is coded, both of these calls block until completed. This is inefficient.

References

Thrift has been moved to the Apache incubator. The project page is:

`http://incubator.apache.org/thrift/`

Hadoop and Sector are both freely available and Open Source. Links to the project web pages are below.

`http://hadoop.apache.org/core/`
`http://sector.sourceforge.net/`

MalGen and its documentation have been released as a Google Code project under GPL and version `0.9` is available for download at:

`http://code.google.com/p/malgen/`

Although it is mentioned, Thrift over CloudStore is not described in this document. CloudStore, formerly the Kosmos filesystem, has been released under the Apache License Version 2.0 and is available at:

`http://kosmosfs.sourceforge.net/`

Copyright Open Cloud Consortium 2009