An overview of Apache THRIFT and its architecture

KARTHIK ANBAZHAGAN[1]

[1]School of Informatics and Computing, Bloomington, IN 47408, U.S.A.
*Corresponding authors: kartanba@iu.edu

April 13, 2017

Thrift is a software framework developed at Facebook to accelerate the development and implementation of efficient and scalable cross-language development services. Its primary goal is to enable efficient and reliable communication across programming languages by abstracting the portions of each language that tend to require the most customization into a common library that is implemented in each language. This paper summarizes the how Thrift provides flexibility in use by choosing different layers of the architecture separately.

Keywords: Cloud, Apache Thrift, cross-language, I524

https://github.com/cloudmesh/sp17-i524/tree/master/paper2/S17-IR-2008/report.pdf

INTRODUCTION

Apache Thrift is an Interface Definition Language [1] (IDL) used to define and create services between numerous languages as a Remote Procedure Call (RPC). Thrift's lightweight framework and its support for cross-language communication makes it more robust and efficient compared to other RPC frameworks like SOA [2] (REST/SOAP). It allows you to create services that are usable by numerous languages through a simple and straightforward IDL. Thrift combines a software stack with a code generation engine to build services that works efficiently and seamlessly between $C++$, Java, Python, PHP, Ruby, Erlang, Perl, Haskell, C, Cocoa, JavaScript, Node.js, Smalltalk, and OCaml. In addition to interoperability, Thrift can be very efficient because of a serialization mechanism [3] which can save both space and time. In other words, Apache Thrift lets you create a service to send/receive data between two or more softwares that are written in completely different languages/platforms.

Thrift was originally developed at Facebook and is one of the core parts of their infrastructure. The choice of programming language at Facebook [4] was based on what language was best suited for the task at hand. This flexibility resulted in difficulties when these applications needed to call one another and Facebook needed an application that could meet their needs of interoperability, transport efficiency, and simplicity. Out of this need, they developed efficient protocols and a service infrastructure which became Thrift. Facebook decided to make Thrift an Open Source and finally contributed it to Apache Software Foundation (ASF) in April 2007 in order to increase usage and development. Thrift was later released under the Apache 2.0 license.

ARCHITECTURE

Figure. 1 Architecture of Apache Thrift shows the architecture of a model for using the Thrift Stack. It is essential to understand every component of the architecture to understand how Apache Thrift works. It includes a complete stack for creating clients and servers. The top portion of the stack is the user generated code from the Thrift Client-Server definition file. The next layer of the framework are the Thrift generate client and processor codes which also comprises of data structures. The next two important layers are the protocol and transport layers which are part of the Thrift run-time libraries. This provides Thrift the freedom to define a service and change the protocol and transport without regenerating any code. Thrift includes a server infrastructure to tie the protocols and transports together. There are blocking, non-blocking, single and multi-threaded servers. The 'Physical' portion of the stack varies from stack to stack based on the language. For example, for Java and Python network I/O, the built-in libraries are leveraged by the Thrift library, while the C++ implementation uses its own custom implementation. Thrift allows users to choose independently between protocol, transport and server. With Thrift being originally developed in C++, Thrift has the greatest variation among these in the C++ implementation [5].

Transport Layer

The transport layer provides simple freedom for read/write to/from the network. Each language must have a common interface to transport bidirectional raw data. The transport layer describes how the data is transmitted. This layer seperates the underlying transport from the rest of the system, exposing only the following interface: open, close, isOpen, read, write, and flush

There are multiple transports supported by Thrift:

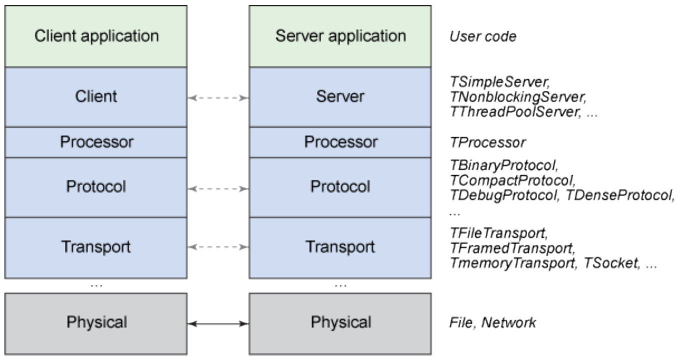
Client application
Server application
User code
Client
Server
TSimpleServer, TNonblockingServer, TThreadPoolServer, ...
Processor
Processor
TProcessor
Protocol
Protocol
TBinaryProtocol, TCompactProtocol, TDebugProtocol, TDenseProtocol, ...
Transport
Transport
TFileTransport, TFramedTransport, TmemoryTransport, TSocket, ...
Physical
Physical
File, Network

Fig. 1. Architecture of Apache Thrift [6]

1. **TSocket**: The TSocket class is implemented across all target languages. It provides a common, simple interface to a TCP/IP stream socket and uses blocking socket I/O for transport.
2. **TFramedTransport**: The TFramedTransport class transmits data with frame size headers for chunking optimization or non-blocking operation
3. **TFileTransport**: The TFileTransport is an abstraction of an on-disk file to a data stream. It can be used to write out a set of incoming Thrift requests to a file on disk
4. **TMemoryTransport**: Uses memory for I/O operations. For example, The Java implementation uses a simple ByteArrayOutput stream
5. **TZlibTransport**: Performs compression using zlib. It should be used in conjunction with another transport

Protocol Layer

The second major abstraction in Thrift is the separation of data structure from transport representation. While transporting the data, Thrift enforces a certain messaging structure. That is, it does not matter what method the data encoding is in, as long as the data supports a fixed set of operations that allows it to be read and written by generated code. The Thrift Protocol interface is very straightforward, it supports two things: bidirectional sequenced messaging, and encoding of base types, containers, and structs.

Thrift supports both text and binary protocols. The binary protocols almost always outperforms text protocols, but sometimes text protocols may prove to be useful in cases of debugging. The Protocols available for the majority of the Thrift-supported languages are:

1. **TBinaryProtocol**: A straightforward binary format encoding takes numeric values as binary, rather than converting to text
2. **TCompactProtocol**: Very efficient and dense encoding of data. This protocol writes numeric tags for each piece of data. The recipient is expected to properly match these tags with the data
3. **TDenseProtocol**: It's similar to TCompactProtocol but strips off the meta information from what is transmitted and adds it back at the receiver side
4. **TJSONProtocol**: Uses JSON for data encoding
5. **TSimpleJSONProtocol**: A write-only protocol using JSON. Suitable for parsing by scripting languages.
6. **TDebugProtocol**: Sends data in the form of human-readable text format. It can be well used in debugging applications involving Thrift.

Processor Layer

A processor encapsulates the ability to read data from input streams and write to output streams. The processor layer is the simplest layer. The input and output streams are represented by protocol objects. Service-specific processor implementations are generated by the Thrift compiler and these generated codes make the Process Layer of the architecture stack. The processor essentially reads data from the wire (using the input protocol), delegates processing to the handler (implemented by the user), and writes the response over the wire (using the output protocol).

Server Layer

A server pulls together all the various functionalities to complete the Thrift server layer. First, it creates a transport, then specifies input/output protocols for the transport. It then creates a processor based on the I/O protocols and waits for incoming connections. When a connection is made, it hands them off to the processor to handle the processing. Thrift provides a number of servers:

1. **TSimpleServer**: A single-threaded server using standard blocking I/O socket. Mainly used for testing purposes
2. **TThreadPoolServer**: A multi-threaded server with N worker threads using standard blocking I/O. It generally creates five minimum threads in the pool if not specified otherwise
3. **TNonBlockingServer**: A multi-threaded server using non-blocking I/O
4. **THttpServer**: A HTTP server (for JS clients)
5. **TForkingServer**: Forks a process for each request to server

ADVANTAGES AND LIMITATIONS OF THRIFT

A few reasons where Thrift is robust and efficient compared to other RPC frameworks are that Thrift leverages the cross-language serialization with lower overhead than alternatives such as SOAP due to use of binary format. Since Thrift generates the client and server code completely [7], it leaves the user with the only task of writing the handlers and invoking the client. Everything including the parameters and returns are automatically validated and analysed. Thrift is more compact than HTTP and can easily be extended to support things like encryption, compression, non blocking IO, etc. Since Protocol Buffers [8] are implemented in a variety of languages, they make interoperability between multiple programming languages simpler.

While there numerous advantages of Thrift over other RPC frameworks, there are a few limitations. Thrift [9] is limited to only one service per server. There can be no cyclic structs. Structs can only contain structs that have been declared before it. Also, a struct also cannot contain itself. Important OOP concepts like inheritance and polymorphism are not supported and neither can Null be returned by a server. Instead a wrapper struct or value is expected. No out-of-the-box authentication service available between server and client and no Bi-Directional messaging is available in Thrift.

CONCLUSION

Thrift provides flexibility in use by choosing different layers of the architecture separately. As mentioned in the advantage section, Thrift usage of the cross-language serialization with lower overheads makes it more efficient compared to other similar technologies. Thirft avoids duplicated work by writing buffering and I/O logic in one place. Thrift has enabled Facebook to build scalable back-end services efficiently. It has been employed in a wide variety of applications at Facebook, including search, logging, mobile, ads, and the developer platform. Application developers can focus on application code without worrying about the sockets layer.

REFERENCES

[1] Apache, “Thrift interface description language,” Web Page, 2016. [Online]. Available: https://thrift.apache.org/docs/idl

[2] K. Sandoval, “Microservice showdown – rest vs soap vs apache thrift,” May 2015, accessed: 2015-05-19. [Online]. Available: http://nordicapis.com/microservice-showdown-rest-vs-soap-vs-apache-thrift-and-why-it-matters/

[3] P. Kloe, “Benchmarks serializers,” Web Page, July 2016, accessed: 2016-07-09. [Online]. Available: https://github.com/eishay/jvm-serializers/wiki

[4] M. Slee, A. Agarwal, and M. Kwiatkowski, “Thrift: Scalable cross-language services implementation,” 2013. [Online]. Available: https://thrift.apache.org/static/files/thrift-20070401.pdf

[5] A. Prunicki, “Apache thrift,” Web Page, June 2009, accessed: 2009-06-01. [Online]. Available: https://objectcomputing.com/resources/publications/sett/june-2009-apache-thrift/

[6] C. Sun, “Understanding how thrift rpc works,” Web Page, March 2015, accessed: 2015-09-22. [Online]. Available: http://sunchao.github.io/posts/2015-09-22-understanding-how-thrift-works.html

[7] S. Dimopoulos, “Generating code with thrift,” Web Page, 2013, accessed: 2013-01-01. [Online]. Available: http://thrift-tutorial.readthedocs.io/en/latest/usage-example.html

[8] Google, “Protocol buffers - google’s data interchange format,” Web Page, 2015, accessed: 2017-03-01. [Online]. Available: https://developers.google.com/protocol-buffers/

[9] C. Maheshwari, “Apache thrift: A much needed tutorial,” Web Page, August 2013, accessed: 2013-08-01. [Online]. Available: digital-madness.in/blog/wp-content/uploads/2012/11/BSD_08_2013.8-18.pdf