Apache Thrift by Example

ECE 454 / 751: Distributed Computing

Instructor: Dr. Wojciech Golab

wgolab@uwaterloo.ca

Slides are derived from Apache Thrift source code
and documentation available at https://thrift.apache.org/.

Learning objectives

To develop a practical understanding of Apache Thrift:

- the Thrift IDL (Interface Definition Language)
- generation of client and server stubs
- implementing service handlers, clients, and servers
- multi-threading
- middleware layer protocols
- application layer protocol versioning

Note: this material is based on Thrift version 0.13.0 for Java.

Overview

- Thrift was developed at Facebook. Details published in http://thrift.apache.org/static/files/thrift-20070401.pdf.
- Language bindings now available for C++, Java, Python, PHP, Ruby, Erlang, Perl, Haskell, C#, Cocoa, JavaScript, Node.js, Smalltalk, OCaml, and Delphi.
- Scala support available via Scrooge, courtesy of Twitter (https://github.com/twitter/scrooge).
- Ample documentation available online (see last slide).

The Thrift software stack
(based on https://thrift.apache.org/docs/concepts)

```
+-------------------------------------------+
| Server - receives incoming connections    |
| (single-threaded, event-driven etc)       |
+-------------------------------------------+
| Processor - reads/writes I/O streams      |
| (compiler generated)                      |
+-------------------------------------------+
| Protocol - encodes/decodes data           |
| (JSON, compact etc)                       |
+-------------------------------------------+
| Transport - reads/writes network          |
| (raw TCP, HTTP etc)                       |
+-------------------------------------------+
```

Defining services using the IDL

```
namespace java example1

exception IllegalArgument {
  1: string message;
}

service MathService {
   double sqrt(1:double num) throws (1: IllegalArgument ia)
}
```

Notes on IDL syntax

- **base types:**
  bool, byte, i16, i32, i64, double, binary, string, void
- **containers:**
  list<type>, set<type>, map<type1,type2>
- **other types:** const**,** typdef, enum, struct, exception
- **field modifiers**: required, optional, default values
- **services and procedures:** service, extends, oneway
- **namespace:** determines Java package
- **file inclusion:** similar to C/C++ preprocessor

Compiling the IDL and stubs

1. Place the IDL example in a file called example1.thrift.
2. Execute the following command:
   ```
   thrift --gen java example1.thrift
   ```
3. Check for syntax errors. On success, look at the generated code in the "gen-java" subdirectory:
   ```
   gen-java/example1/IllegalArgument.java
   gen-java/example1/MathService.java
   ```
4. Compile the generated Java classes using javac. Remember to set the classpath correctly, for example by including "lib/*" where "lib" is the directory containing libthrift and other jars.

Implementing the service handler
(file name example1/MathServiceHandler.java)

```
package example1;
…
public class MathServiceHandler implements MathService.Iface {
    public MathServiceHandler() {
    }
    public double sqrt(double d) throws IllegalArgument {
        if (d < 0)
            throw new IllegalArgument(
                        "Can't take the square root of a negative number!");
        return Math.sqrt(d);
    }
}
```

Synchronous RPC client
(file name example1/JavaClient.java)

```
…
public static void main(String [] args) {
   try {
      TSocket sock = new TSocket(args[0], Integer.parseInt(args[1]));
      TTransport transport = new TFramedTransport(sock);
      TProtocol protocol = new TBinaryProtocol(transport);
      MathService.Client client = new MathService.Client(protocol);
      transport.open();
      double arg = Double.valueOf(args[2]);
      System.out.println("sqrt(" + arg + ")=" + client.sqrt(arg));
      transport.close();
   } catch (TException x) {
      …
```

Single-threaded server
(file name example1/JavaServer.java)

```
…
public static void main(String [] args) throws TException {
    MathService.Processor processor =
        new MathService.Processor(new MathServiceHandler());
    TServerSocket socket = new TServerSocket(Integer.parseInt(args[0]));
    TSimpleServer.Args sargs = new TSimpleServer.Args(socket);
    sargs.protocolFactory(new TBinaryProtocol.Factory());
    sargs.transportFactory(new TFramedTransport.Factory());
    sargs.processorFactory(new TProcessorFactory(processor));
    TServer server = new TSimpleServer(sargs);
    server.serve();
    …
```

Running the code

- Example command for launching the server:

```
java -cp … example1.JavaServer 10123
```

- Example command for launching the client:

```
java -cp … example1.JavaClient serverhost 10123 55
```

Distribution transparency

- Thrift clients must know the host name and port number for a given service, which violates the principle of **location transparency**. Alternatively, the services could be discovered using a directory server, as described in Section 4.2.4 of the textbook.
- Thrift objects may throw a variety of exceptions related to inter-process communication. For example a transport may throw TTransportException when a TCP connection cannot be opened. Similarly, a protocol object may throw TProtocolException if the client and server use mismatched Thrift protocol versions. This violates the principle of **access transparency**.

A few notes on programming

- **Note 1:** Some online Thrift examples hard-code the host name and port number for simplicity, but you should not. Specify the host and port as command line arguments instead, or use a property file.

- **Note 2:** TSocket and TServerSocket objects in Thrift, as well as their non-blocking counterparts, are usually implemented on top of TCP/IP. Reuse these objects when possible to mitigate overheads associated with connection creation and teardown.

- **Note 3:** The Thrift transports, protocols, and client stubs are not thread-safe. In a multi-threaded client either share them carefully or do not share them at all.

Asynchronous RPC client

- Must use a non-blocking socket and transport.
- Must provide a TAsyncClientManager instance to handle callbacks.
- Must encapsulate the callback method in an object that implements the AsyncMethodCallback interface.
- Synchronization may be required between the callback method and the code that makes the async RPC call, for example using a Condition or CountDownLatch in Java.

Asynchronous RPC client

```
…
static CountDownLatch latch = new CountDownLatch(1);
public static void main(String [] args) {
    try {
        TNonblockingTransport transport =
            new TNonblockingSocket(args[0], Integer.parseInt(args[1]));
        TProtocolFactory pf = new TBinaryProtocol.Factory();
        TAsyncClientManager cm = new TAsyncClientManager();
        MathService.AsyncClient client =
            new MathService.AsyncClient(pf, cm, transport);
        double arg = Double.valueOf(args[2]);
        client.sqrt(arg, new SqrtCallback());
        latch.await();
        transport.close();
    } catch …
```

Asynchronous RPC client

```
static class SqrtCallback implements
                                        AsyncMethodCallback<Double> {

    public void onComplete(Double response) {
        ...
        latch.countDown();
    }
    public void onError(Exception e) {
        ...
        latch.countDown();
    }
```

Multi-threaded server

- Multi-threaded servers offer substantial performance advantages, but require more careful programming.
- In most cases, the server uses a non-blocking socket and framed transport. (Exception: TThreadPoolServer).
- The server may have additional configuration options, for example to control the number of threads.
- The configuration options for each server implementation are defined in the inner "Args" class (e.g., THsHaServer.Args). Online API docs are available but beware that the method signatures vary across Thrift versions.

Multi-threaded server

```
…
public static void main(String [] args) throws TException {
    MathService.Processor processor =
      new MathService.Processor(new MathServiceHandler());
    TNonblockingServerSocket socket =
      new TNonblockingServerSocket(Integer.parseInt(args[0]));
    THsHaServer.Args sargs = new THsHaServer.Args(socket);
    sargs.protocolFactory(new TBinaryProtocol.Factory());
    sargs.transportFactory(new TFramedTransport.Factory());
    sargs.processorFactory(new TProcessorFactory(processor));
    sargs.maxWorkerThreads(5);
    TServer server = new THsHaServer(sargs);
    server.serve();
    ….
```

Java server implementations

- **TSimpleServer** uses a single thread and blocking I/O.
- **TNonblockingServer** uses a single thread and non-blocking I/O. It can handle parallel connections but executes requests serially just like TSimpleServer.
- **THsHaServer** uses one thread for network I/O and a pool of worker threads. It can process multiple requests in parallel.
- **TThreadedSelectorServer** uses a pool of threads for network I/O and a pool of worker threads for request processing.
- **TThreadPoolServer** uses one thread to accept connections and then handles each connection using a dedicated thread drawn from a pool of worker threads.

Java servers and resource limits

- **TThreadPoolServer** often yields the best performance as it uses minimal synchronization internally. However, it consumes one thread per connection, leading to resource exhaustion when the number of parallel connections is high.
- When a Java program attempts to create a large number (e.g., ~100) threads, it will encounter the following error: OutOfMemoryError: Unable to create new native thread
- No matter which multi-threaded server implementation you use, please limit the maximum size of your worker thread pool to 64 or less when running code on ecelinux, and run at most one Java process per machine (e.g., use distinct hosts for client and server processes).

Thrift protocols

- Thrift provides a number middleware layer protocols on top of the transport layer to encode RPC requests and responses.
- **TBinaryProtocol:** encodes numeric values in a straightforward binary format.
- **TCompactProtocol:** like TBinaryProtocol but more compact, uses variable-length encoding for integers.
- **TJSONProtocol:** uses human-readable JSON format.
- Additional protocols are supported in some languages, particularly C++.

Application protocol versioning

Thrift fields can be modified in a manner that provides compatibility between old and new protocol versions, provided that a few rules are followed:

1. The manually assigned numeric tags of existing fields should never be changed.
2. New fields can be added as long as they are declared `optional` and have appropriate default values.
3. Fields that are no longer needed can be removed as long as their tag numbers are never reused. Unrecognized fields are skipped by the parser, but old servers may be unable to process requests from new clients.
4. Default values can be changed.

Further reading

- Apache Thrift web site: https://thrift.apache.org/
- Source code in GitHub: https://github.com/apache/thrift
- The "missing guide": https://diwakergupta.github.io/thrift-missing-guide/
- Article on schema evolution: https://martin.kleppmann.com/2012/12/05/schema-evolution-in-avro-protocol-buffers-thrift.html