CSE 124
Distributed programming and Remote Procedure Calls (RPC): Apache Thrift

February 25, 2016, UCSD
Prof. George Porter
Announcements
RPC Components

• End-to-end RPC protocol
  – Defines messages, message exchange behavior, ...

• Programming language support
  – Turn “local” functions/methods into RPC
  – Package up arguments to the method/function, unpackage on the server, ...
  – Called a “stub compiler”
  – Process of packaging and unpackaging arguments is called “Marshalling” and “Unmarshalling”
High-level overview
Outline

• Thrift overview
• In-class development of a simple “ATM machine” service
Apache Thrift Overview

Thanks to Diwaker Gupta
http://diwakergupta.github.io/thrift-missing-guide/
Features

• Cross-platform RPC toolkit developed by Facebook
• Languages:
  – C++, C#, Cocoa, Java, OCaml, PHP, Ruby, Python, …
• Namespaces
  – (as compared to flat identifiers)
• Data types
  – Base, Structs, Constants, Enums, Containers (Set, List, …)
• Exceptions
• Services
  – The actual procedures you are remotely calling
IDL: Interface Definition Language

• Language-neutral way of specifying:
  – Data structures
  – Services, consisting of procedures/methods

• Stub compiler
  – Compiles IDL into Python, Java, etc.

```sh
$ thrift --gen py
```

```sh
$ thrift --gen java
```

Python

Java
IDL Base types

- bool: A boolean value (true or false)
- byte: An 8-bit signed integer
- i16: A 16-bit signed integer
- i32: A 32-bit signed integer
- i64: A 64-bit signed integer
- double: A 64-bit floating point number
- string: A text string encoded using UTF-8 encoding
IDL Containers

• **list<\textit{t1}>**
  - Ordered list of type \textit{t1}

• **set<\textit{t1}>**
  - Unordered set of unique items of type \textit{t1}

• **map<\textit{t1, t2}>**
  - Map of unique keys of type \textit{t1} to values of type \textit{t2}
IDL Services

• Defines procedures/methods to be invoked
• Similar to Java interfaces
  – You specify their type signature in the IDL
  – Then actually implement the methods in Java/Python/… files
    • (But Thrift helps you out by handling much of the cookie-cutter code generation)

```
service Calculator {
    i32 add(1:i32 num1, 2:i32 num2)
}
```
IDL Positional Arguments

• Why?
  – `i32 add(1:i32 num1, 2:i32 num2)`
• Instead of:
  – `i32 add(i32 num1, i32 num2)`
Making services *evolvable*

- Consider supporting multiple generations of services
  - (see optional paper on evolving services by Brewer et al.)
- Parameters can be added/dropped over time
  - `void addUser(String firstname, String lastname, i32 ID)`
  - Becomes
  - `void addUser(String fullname, i32 ID, i32 phonenum)`
- Confusion results; type information not enough to differentiate old vs. new service API
- Explicit numbering allows parameter order and the existence of parameters change
Explicit Parameter numbering

- void addUser(1:String firstname, 2:String lastname, 3:i32 ID)
- →
- void addUser(4:String fullname, 5:i32 phonenum, 3:i32 ID)
Parameter numbering and Structs

- Explicit parameter numbers applies to structures too
- Required/optional further constrains RPC interface

```c
struct Location {
  1: required double latitude;
  2: required double longitude;
}

struct Tweet {
  1: required i32 userId;
  2: required string userName;
  3: required string text;
  4: optional Location loc;
  16: optional string language = "english"
}
```
Thrift’s layered model

<table>
<thead>
<tr>
<th>Server</th>
<th>(single-threaded, event-driven etc)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Processor</td>
<td>(compiler generated)</td>
</tr>
<tr>
<td>Protocol</td>
<td>(JSON, compact etc)</td>
</tr>
<tr>
<td>Transport</td>
<td>(raw TCP, HTTP etc)</td>
</tr>
</tbody>
</table>
Transport Protocol

• Reading/writing to the network (or other channel)
• Can utilize TCP, or even HTTP
• Can also read and write to files on a disk
  – Facebook uses this feature to record `log()` calls in a logging system, and then “replays” them later to actually record the logs
Protocol

- Maps in-memory data structures to on-the-wire formats
- Knows how to convert each IDL data type
  - For each language
- Examples:
  - writeI32(i32)
  - readI32(i32)
  - writeString(string)
  - readString(string)
  - ...
- Text-based JSON, compact binary representation, ...
Processor and Server

• Processor
  – Compiler-generated “glue” between RPC protocol messages and your code

• Server
  – High-level controller of all we’ve talked about
  – Creates the transport (e.g., open TCP sockets, bind, listen, accept, …)
  – Creates input/output protocols
  – Creates a processor based on the input/output protocols
  – Wait for incoming connections and hand off to processor
ATM Server
Simple ATM Server

- Operations:
  - login
    - Account number + PIN
  - deposit
    - $$$
  - getBalance
  - logout
Simple ATM Server

- Keeping track of account + pin with "login tokens"
- After logging in, get a token
- Use token to deposit money, withdraw, transfer, ...
ATM Machine Project Structure

```
src
  client
    cl.py
  server
    build.xml
    run-server.sh
    src
      ATMHandler.java
      ATMServer.java
  thrift
    ATM.thrift
```

IDL