Protocol Buffers

Googles Approach for Distributed Systems

Slides partially based upon Majd F. Sakr, Vinay Kolar, Mohammad Hammoud and Google Protobuf Tutorial

https://developers.google.com/protocol-buffers/docs/tutorials
Google Case Study

■ Google provides a case study of Distributed Systems

■ Scalability:
  ■ Google has scaled from an initial production system in 1998 to handling 63,000 per Second.

■ Reliability and Fault-tolerance:
  ■ The main search engine has never experienced an outage.
  ■ Users can expect a query result in 0.2 seconds
    ■ 24x7 availability with 99.9% Service Level Agreement (for paying customers)

■ Variety of software applications are supported on Google Infrastructure
  ■ Google has built a generic infrastructure that handles many varying web applications (search, maps, voice, email, social networking, ads, docs, …)

■ Google is offering Platform As A Service
  ■ With the launch of Google App Engine, Google is stepping beyond providing software services
  ■ It is now providing its distributed systems infrastructure for application developers
Google has created a large distributed system from commodity PCs

Cluster
Approx 30 racks (around 2400 PCs)
2 high-bandwidth switches (each rack connected to both the switches for redundancy)
Placement and replication generally done at cluster level
Google Infrastructure – Conceptual HW View

Data centre architecture

To other data centres and the Internet
Google has developed a set of services that are tailored for various applications running on Google infrastructure.

We will study two important communication paradigms developed by Google:
- Google Protocol Buffer
- Google Publish-Subscribe

Diagram:
- Distributed computation
  - MapReduce
  - Sawzall
- Data and coordination
  - GFS
  - Chubby
  - Bigtable
- Communication paradigms
Communication Protocols
Google Publish-Subscribe

- Google adopts a *topic-based* PS system
  - A number of channels for event streams with channels corresponding to particular topics

- Event contains the following fields:
  - Header
  - Set of keywords
  - Payload: Opaque to the programmer

- Subscription request specify
  - Channel
  - Filter defined over the set of keywords

- Reliability
  - guarantees at least once
Google Protocol Buffer

- Google adopts a minimal (but efficient) remote invocation service

- Recall that: Remote invocation requires – among all the other services – the following two components
  - Serialization of data
  - Agreement on data representation (data-type size and format)

- Protocol Buffer (PB) is a common serialization format for Google
Data serialization (in Java):

1. RMI/Java built-in serialization
   - Ex.: writeObject/readObject(aObject)
   - Easy to use, but inefficient in terms of space (extra fields)
   - No language interoperability

2. Manual binary encoding
   - Ex.: 4 ints as - and aObject.getBytes()
   - Space efficient, but time efficiency depends on parsing methods
   - Difficult for complex objects
   - Language interoperable

3. CORBA
   - Part of Java
   - But deprecated technology
   - Language interoperable

3. Human-readable formats
   - Ex.: Using XML, JSON, DOM, SAX, STAX, JAXB, JAXP, etc.
   - Inefficient (space and time w/ human readable format)
   - Language interoperable
## Comparison with some other Products

<table>
<thead>
<tr>
<th>Protobuf</th>
<th>Avro</th>
<th>Thrift</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Complex Types</strong></td>
<td><strong>Unions (Choice Type)</strong></td>
<td><strong>Self-References (Trees)</strong></td>
</tr>
<tr>
<td>Yes</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td><strong>Avro</strong></td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td><strong>Thrift</strong></td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td><strong>XML</strong></td>
<td>Yes</td>
<td>Yes</td>
</tr>
</tbody>
</table>

© Tony Ng
Protocol Buffer

- Protocol Buffers (Protobuf) is Google's approach to data serialization:
  - A description language
  - A compiler
  - A library

- Easy-to-use, efficient automatic binary encoding

- Created by and in production at Google Inc.

- Publicly launched in 2008.

- Language interoperable:
  - Java, Go, C++, and Python, C, C#, Erlang, Perl, PHP, Ruby, etc.
The goal of Protocol Buffer is to provide a language- and platform-neutral way to specify and serialize data such that:

- Serialization process is efficient, extensible and simple to use
- Serialized data can be stored or transmitted over the network

In Protocol buffers, Google has designed a language to specify messages
Sample for an IDL Definition (.proto File)

- Define the interface in an IDL File

```proto2
syntax = "proto2";
package tutorial;
option java_package = "com.example.tutorial";
option java_outer_classname = "AddressBookProtos";
option java_generic_services = true; // generate stubs for rpc

message Person {
  required string name = 1;
  required int32 id = 2;
  optional string email = 3;

  enum PhoneType {
    MOBILE = 0;
    HOME = 1;
    WORK = 2;
  }

  message PhoneNumber {
    required string number = 1;
    optional PhoneType type = 2 [default = HOME];
  }

  repeated PhoneNumber phone = 4;
}
```
Protocol Buffer Language

- Message contains uniquely numbered fields
- Field is represented by
  
  <field-type, data-type, field-name, encoding-value, [default value]>

- Available data-types
  - Primitive data-type
    - `int`, `float`, `bool`, `string`, `raw-bytes`
  - Enumerated data-type
  - Nested Message
    - Allows structuring data into an hierarchy

```protobuf
syntax="proto2";
package tutorial;
option java_package = "ccm.example.tutorial";
option java_outer_classname = "AddressBookProtos";
message Person {
  required string name = 1;
  required int32 id = 2;
  optional string email = 3;
}

enum PhoneType {
  MOBILE = 0;
  HOME = 1;
  WORK = 2;
}

message PhoneNumber {
  required string number = 1;
  optional PhoneType type = 2 [default = HOME];
}

  repeated PhoneNumber phone = 4;
}
Protocol Buffer Language (cont’d)

- **Field-types can be:**
  - Required fields
  - Optional fields
  - Repeated fields
    - *Dynamically sized array*

- **Encoding-value**
  - A unique number (=1,=2,…) represents a tag that a particular field has in the binary encoding of the message

```protobuf
syntax="proto2";
package tutorial;
option java_package = "com.example.tutorial";
option java_outer_classname = "AddressBookProtos";
message Person {
  required string name = 1;
  required int32 id = 2;
  optional string email = 3;

  enum PhoneType {
    MOBILE = 0;
    HOME = 1;
    WORK = 2;
  }

  message PhoneNumber {
    required string number = 1;
    optional PhoneType type = 2 [default = HOME];
  }

  repeated PhoneNumber phone = 4;
}
```
Compiling the Description

■ Compile the *.proto File

```
c:> protoc --java_out=. name_of_protoc.proto
```

■ tutorial

- src
  - com.example.tutorial
  - AddressBookProtos.java
Using in a Java Program

- Import in your Source

```java
package com.example.tutorial;
import com.example.tutorial.AddressBookProtos.AddressBook;
import com.example.tutorial.AddressBookProtos.Person;
```

- Instantiate an Object

```java
Person john = Person.newBuilder()
    .setId(12345)
    .setName("John Foo Bar")
    .setEmail("john@foobar.pt")
    .addPhone(Person.PhoneNumber.newBuilder()
        .setNumber("+351 999 999 999")
        .setType(Person.PhoneType.HOME)
        .build())
    .build();
```

- The **Builder** class:
  - Messages are immutable in protocol buffer, Builder class is mutable
File as Storage

■ Write to a file

FileOutputStream aOutput = new FileOutputStream("theFilename");
Person aPerson = Person.newBuilder().set... // instantiate a Person
aPerson.writeTo(aOutput);
aOutput.close();

■ Read from a file

Person aPerson = Person.parseFrom(new FileInputStream("theFilename");
// Do something with the received Person
TCP/IP Socket for Date Transfer

- **Server Side (receive data)**

```java
ServerSocket aServerSocket = new ServerSocket(10000);
Socket aConnection = aServerSocket.accept();
Person aPerson = Person.parseDelimitedFrom(aConnection.getInputStream());
```

- **Client Side (send data)**

```java
Person aPerson = Person.newBuilder().set... //instantiate a Person
Socket aSocket = new Socket("127.0.0.1", 10000);
aPerson.writeDelimitedTo(aSocket.getOutputStream());
```
UDP for Date Transfer

■ Receive Data

```java
DatagramSocket aServerSocket = new DatagramSocket(10000);
byte[] aReceiveData = new byte[1024];
DatagramPacket aReceivePacket = new DatagramPacket(aReceiveData, aReceiveData.length);
aServerSocket.receive(aReceivePacket);
ByteArrayInputStream aInput = new ByteArrayInputStream(aReceiveData);
Person aPerson = Person.parseDelimitedFrom(aInput);
```

■ Send Data

```java
DatagramSocket aClientSocket = new DatagramSocket(10001);
ByteArrayOutputStream aOutput = new ByteArrayOutputStream(1024);
Person aPerson = Person.newBuilder().set... // instantiate a Person
aPerson.writeDelimitedTo(aOutput);
byte aSendData[] = aOutput.toByteArray();
InetAddress alp = InetAddress.getLocalHost(); // or aReceivePacket.getAddress();
DatagramPacket aSendPacket = new DatagramPacket(aSendData, aSendData.length, alp, 10001);
aClientSocket.send(aSendPacket);
```
Remote Procedure Call
Extensibility of PB

- In addition to being language- and platform-neutral, PBs are also agnostic with respect to underlying RPC protocol.

- PB library provides two abstract interfaces:
  - RpcChannel:
    - Provides a common interface to underlying communication
    - e.g., Programmer can specify if HTTP or FTP has to be used for communicating data
  - RpcController:
    - Providing common control interface
Supporting RPC using Protocol Buffers

- PB produces a serialized data that can be used for storage or communications
- Most common use is to use PB for RPCs
- Example:
  - RequestType can correspond to list of keywords
  - ResponseType can then correspond to a list of books matching the keywords

```java
service SearchService {
  rpc Search(RequestType) returns (ResponseType)
}
```

- `protoc` compiler takes this specification and produces
  - Abstract interface SearchService
  - A stub that supports type-safe RPC calls
  - Attention only if: `option java_generic_services = true;`
The Definition of a Service Interface

- proto-File to describe the Messages and Services

```protobuf
syntax="proto2";
package tutorial;

option java_package = "com.example.tutorial";
option java_outer_classname = "HelloProtos";
  //if you don't do this, protoc won't generate the stubs you need for rpc
option java_generic_services = true;

message HelloRequest
{
  required string name = 1;
}

message HelloReply
{
  required string message = 1;
}

//In generated class, this class is abstract class
//that extends service method need to extends this
service GreetingService {
  rpc sayHello(HelloRequest) returns (HelloReply);
}
```

```
c:> protoc --java_out=. name_of_protoc.proto
```

A "wrapper" class HelloProtos is generated.
Implementation of the Services

- MyServiceImpl extends abstract generated Class "Service"

```java
package com.example.tutorial;

import java.util.logging.*;
import com.google.protobuf.RpcCallback;
import com.google.protobuf.RpcController;
import com.example.tutorial.HelloProtos.*;

public class MyGreetingServiceImpl extends GreetingService {
    private static final Logger log = Logger.getLogger(MyGreetingServiceImpl.class.getName());

    private static HelloReply createReplyMessage(String name) {
        HelloReply.Builder helloReplyBuilder = HelloReply.newBuilder();
        helloReplyBuilder.setMessage("Hello:"+name);
        return helloReplyBuilder.build();
    }

    @Override
    public void sayHello(RpcController controller, HelloRequest helloRequest, RpcCallback<HelloReply> done) {
        log.info("got request:"+helloRequest.getName());
        HelloReply helloReply = createReplyMessage(helloRequest.getName());
        done.run(helloReply);
    }
}
```

get all inner classes
... Implementation of the Services

Register and Start the Server

```java
package com.example.tutorial;

import java.util.concurrent.Executors;
import com.googlecode.protobuf.socketrpc.RpcServer;
import com.googlecode.protobuf.socketrpc.ServerRpcConnectionFactory;
import com.googlecode.protobuf.socketrpc.SocketRpcConnectionFactories;

public class ServerCode {

    public static void main(String[] args) {
        ServerRpcConnectionFactory rpcConnectionFactory =
            SocketRpcConnectionFactories.createServerRpcConnectionFactory(4446);
        RpcServer server = new RpcServer(rpcConnectionFactory,
            Executors.newFixedThreadPool(5), true);
        server.registerService(new MyGreetingServiceImpl());
        server.run();
    }
}
```
package com.example.tutorial;

import com.example.tutorial.HelloProtos.*;
import com.google.protobuf.*;
import java.util.logging.*;
import com.googlecode.protobuf.socketrpc.*;
import java.util.concurrent.*;

public class ProtoClient {
    private static String host = "127.0.0.1";
    private static int port = 4446;
    private static final Logger log = Logger.getLogger(ProtoClient.class.getName());

    private static GreetingService.Stub connectToAsyncService() {
        // Create a thread pool
        ExecutorService threadPool = Executors.newFixedThreadPool(1);
        // Create channel
        RpcConnectionFactory connectionFactory = SocketRpcConnectionFactories.createRpcConnectionFactory(host, port);
        RpcChannel channel = RpcChannels.newRpcChannel(connectionFactory, threadPool);
        GreetingService.Stub myService = GreetingService.newStub(channel);
        return myService;
    }

    private static GreetingService.BlockingInterface connectToBlockingService() {
        GreetingService.BlockingInterface helloService;
        RpcConnectionFactory connectionFactory = SocketRpcConnectionFactories.createRpcConnectionFactory(host, port);
        BlockingRpcChannel channel = RpcChannels.newBlockingRpcChannel(connectionFactory);
        GreetingService.Stub myService = GreetingService.newStub(channel);
        return myService;
    }

    // Client for Calling the Service
    // Connector Methods for Sync and Blocking Calls
... Client for Calling the Service

Do an Async or Blocking Call

```java
private static HelloRequest createRequestMessage(String name) {
    HelloRequest.Builder helloRequestBuilder = HelloRequest.newBuilder();
    helloRequestBuilder.setName(name);
    return helloRequestBuilder.build();
}

public static void main(String[] args) throws Exception {
    RpcController controller = new SocketRpcController();
    HelloRequest helloRequest = createRequestMessage("hugo");

    // Call service sync
    GreetingService.BlockingInterface myServiceRpc = connectToBlockingService();
    HelloReply reply = myServiceRpc.sayHello(controller, helloRequest);
    log.info("Received Sync Response: " + reply.getMessage());

    // Call service async
    GreetingService.Stub myService = connectToAsyncService();
    myService.sayHello(controller, helloRequest,
            new RpcCallback<HelloReply>() {
                public void run(HelloReply reply) {
                    log.info("Received Async Response: " + reply.getMessage());
                }
            });
    Thread.sleep(1000);
    System.exit(0);
}
```
Java Installation (Version 2.4.1)

- Minimal for Java

- Protoc Compiler (protoc-2.4.1-win32.zip) for Windows
  - https://github.com/google/protobuf/releases/tag/v2.4.1

- Protobuff Java Runtime
  - http://search.maven.org/remotecontent?filepath=com/google/protobuf/protobuf-java/2.4.1/protobuf-java-2.4.1.jar

- Protobuff socket-rpc
Compare PB with traditional RPCs

■ In messages, field-types are encoded as numbers. Hence, lesser data needs to be communicated.

■ RPCs using PB restricts single input parameter and single result parameter
  ■ Supports extensibility and software reusability
  ■ Pushes the complexity towards data

■ Programmer can control protocols used for communication by writing their own RpcChannel
  ■ But, we have studied that RPC was designed to relieve programmer from communication. Discuss this dilemma.