Computer Science Project Proposal

Protocol Buffers in Standard ML

R. S. Voroneanu, Queens' College

Originator: Prof L. Paulson

16 October 2015

Project Supervisor: Dr N. Sultana

Director of Studies: Dr A. Rice

Project Overseers: Dr J. Crowcroft & Dr T. Sauerwald

Introduction and Description of Work

Protocol Buffers is a language-neutral, platform-neutral extensible way of serializing structured data as series of bytes. They were designed by Google, as an efficient and simple way of exchanging information between different systems. In terms of functionality, they are similar to XML or JSON, but aim to be more compact. As part of the Protocol Buffers, Google designed a description language that has to be used when defining the structure of data. The definition of such a structure has to be written in a separate file which is then passed to a language-specific tool. This tool then generates code in a desired programming language, capable of handling the defined data. Currently supported programming languages include C++, Java and Python.

The Protocol Buffers description language has a format rather like C structs and offers a wide variety of basic datatypes. Among them we can include multiple scalar value types and strings used to represent the data, maps used to keep unordered collections of key-value pairs, enums used to predefine lists of constants and many others. The language also provides ways of choosing the coding schema that will be used, from a set of predefined schemas, and even what optimizations the generated code will aim for. Protocol Buffers can be used in a variety of places, ranging from database storage to Remote Procedure Calls. The latter is directly supported, by having a primitive structure of the description language represent a service and enclosing a set of message-based RPC's in it. All a developer has to do is define the request and response message for each RPC. The language-specific tools will then generate stub code that abstract the transport layer from the application. Because request or response messages can sometimes change, the Protocol Buffer language allows deprecation of certain components so that all clients that have not yet updated their binaries can still make use of the service.

The goal of this project is to implement the language-specific tool for the Standard ML language. The code generated by the tool will follow an API that is compatible with the Poly/ML system.

Resources Required

One computer capable of running the Poly/ML system will be required – my personal computer would suffice. Most of the work will be done on it using an online repository system (github, most likely) for back-up purposes.

The project idea was originated from Dr. L. Paulson in collaboration with Lucas Dixon.

Starting Point

Currently, I am quite comfortable with the Protocol Buffers API in Java and C++, having used them on multiple occasions in the past. I have also skimmed through the different encodings and optimizations. I am relatively comfortable with Standard ML, but expect to have to learn it in more depth. I have not yet written any code.

Substance and Structure of the Project

As mentioned above, the objective of my Part II project is to implement the Protocol Buffers language-specific tool for the Standard ML language that is compatible with the Poly/ML system. The tool itself will be composed of two main components: the accessors/serialization and the services components.

The first of them will generate accessor functions to the different fields defined, and code capable of serializing and deserializing the data. As mentioned above, the developer is able to choose which optimization will be used when handling the data, by providing an option parameter *optimize_for*. I intend to support the CODE_SIZE and SPEED optimizations which, as the name suggests, aim to reduce the length of the code generated or improve the speed at which it will run. Each of the two methods will be tested and compared against each other.

The second component of the tool provides support for the message-based RPC's, by generating code capable of abstracting the transport layer from the application. The core of this code will be composed of stubs that are capable of asynchronously running functions in a remote fashion and at a desired location, but have abstract methods for sending and receiving the data. The abstract methods will be independent of the actual Protocol Buffer definition, and thus I will also provide a default implementation for them in the form of a simple client-server interaction. The clients will be able to send HTTP requests, while the server will be able to handle multiple such requests and solve them in parallel. Therefore, users will get a choice between using an existing code and specializing their own implementation with preferred protocols.

For testing purposes, I will create similar clients and servers in supported languages, such as C++ and Java, and aim to successfully interact with them.

Extensions

There are a few possible extensions that could be included in the project:

- Create a component capable of translating JSON/XML into the Protocol Buffers underlying schema and thus allow users to improve performance without losing the ability to use their preferred language.
- Provide support for the *optimize_for=LITE_RUNTIME* option. The generated code will depend only on the *lite* version of the Protocol Buffers language. This is particularly useful for applications running on constrained platforms like mobile phones.
- Improve the default basic implementation of the RPC by providing load-balancing, streaming or even encryption.
- Evaluate the runtime and memory performance of the generated code, and compare it to another supported programming language. This could be done by setting up a set of machines as clients to a main machine acting as server and recording metrics on all of them.

Success Criteria

- The tool must be able to generate Standard ML structure definitions which access, serialize and deserialize the data correctly following the Protocol Buffers format.
- The tool must generate code for the different optimizations mentioned, for SPEED or CODE_SIZE. Each must outperform the other in its designated resource – runtime speed and code length, respectively.
- The tool should generate signature code that facilitates RPC's following a flexible design. This must be demonstrated by providing default code for a basic RPC client-server interaction.

Timetable and Milestones

26th Oct - 8th Nov

Preparation work. Exercise Standard ML by creating Protocol Buffers examples and hand-writing possible code. This would allow analysing different signature approaches and deciding on a final one. Also, read how the two different optimizations (for SPEED and for CODE_SIZE) are dealt with in different supported languages and design an equivalent version for ML.

9th Nov - 22nd Nov

Continue working on example code and analyse possible approaches for the client-server signatures. By the end of week four I expect to have the signatures and the design ready.

23rd Nov - 29th Nov

By the end of this week I am expecting to finish the lexing and parsing component.

30th Nov - 27th Dec

In these weeks I intend to implement the accessor/serialization/deserialization code generation and write relevant chapters in the dissertation. I am expecting to split this time period equally between the two optimizations.

28th Dec - 17th Jan

Add support for RPC's and write relevant chapters in the dissertation. By the end of week eleven I expect to finish the signature generation component and then write the basic default client-server code by the end of week twelve.

18th Jan - 21 Feb

Evaluation of the different components. Analysing the data and preparing it for write up.

22 Feb - 10 May

Dissertation writing.