Protocol Buffers
Chat Application Design

NICHOLAS MAHLANGU
TIANYU LIU
Harvard College
March 2, 2016

Introduction

Protocol Buffers are an efficient way of serializing structured data, providing similar functionality as XML but in a faster and more robust package. By defining the structure of your data once, you can read and write your structured data in a variety of different languages and data streams. Using this technology, we built a chat application using *Meteor*.

Interfaces Being Looked At

The client and server communicate over HTTP by sending serialized data. All requests on both the client and server are encoded / decoded using JavaScript, as Meteor runs the same JavaScript code for both parties.

Assumed Environment

The client and server must have access to HTTP passed over a port (3000 is the default one we use) and must be able to run JavaScript. Furthermore, the server needs to have npm and Meteor installed.

System Design

Our system was mainly built using *Meteor*, the popular JavaScript app framework, because the same code runs on the client and the server, it ports really easily to iOS and Android, and has a highly reactive GUI thanks to the Blaze framework. In order to integrate Protocol Buffers into *Meteor*, we had to dig

into its implementation files and modify the package, called *ddp-common*, responsible for *Meteor's* JSON communication. In this package is a file called *utils.js* which contains the functions **parseDDP()** and **stringifyDDP()**, both of which only rely on JSON. The implication here is that the entire *Meteor* framework assumes objects are being passed around as JSON objects, so we had to create another layer of abstraction that takes a JSON object, converts it into a Protocol Buffer object, and then serializes it. The implementation for this is done in **json_proto.js**: we manually define the Protocol Buffer abstraction for JavaScript objects, including the basics types (integer, double, string, boolean, etc.), arrays, and objects (the latter 2 can have nested properties within).

One snag we ran into is that we couldn't use npm's **require()** statements because Meteor would ignore them on the front-end. Since *ddp-common* is used on the front-end and the back-end, we had to find a way to include all relevant packages without relying on **require()**. The way we did this is by saving all the source files, manually importing them into the directory, and removing all **require()** statements in our code. In essence, we created a manual dependency tree by including files in a certain sequence.

Back to *utils.js*, we see custom serialization being done with **JSONproto.protoify()** and deserialization with **JSONproto.parse()**. After serialization, the data is converted to a hex string, much fewer bytes are sent over the wire. This is incredibly useful when doing intense database operations and one of the main benefits of using Protocol Buffers.

Resources Conserved

Meteor relies on 2 technologies for keeping the DOM up-to-date with the server: *Blaze*, a declarative library for creating live-updating, reactive user experiences; and *DDP*, a simple protocol for fetching structured data from a server and receiving live updates when data changes (basically REST for web sockets). Because these technologies are built straight into *Meteor*, we minimize the number of requests flowing between the client and the server.

Furthermore, Protocol Buffers offer us a number of other benefits. When compared to XML, Protocol Buffers are 3 - 10 times smaller and 20 - 100 times faster, so we minimize the size of the data we're sending between the client and server and are also able to do it much faster.

Resources Wasted

As mentioned in the implementation section, the entire *Meteor* framework assumes objects are being passed around as JSON objects. This means we

had to do added work to incorporate Protocol Buffers. First, before we send data, we have to convert a JSON object into a Protocol Buffer object and then serialize it. When we receive data, we have to deserialize the data, convert it to a Protocol Buffer object, then convert it to a JSON object for use after. Because we do both of these operations for everything sent between the client and server, we do have a performance hit in this area. However, the application is still fast enough to the point where this seems to be unnoticeable.

Failure Conditions

- Failstops

  A failstop could occur in our system anytime the server goes down. Thanks to *Meteor's* technologies the client will be notified within seconds if the server goes down and will display the message "Trying to connect. There seems to be a connection issue".

- Byzantine Failures

  The correctness of our code is unproven, so while we are unaware of where a byzantine failure may occur, we recognize that such a failure is theoretically possible in our system.

- Receive, Send, and General Omissions

  The client and server are always kept up-to-date with each other thanks to *Meteor's* technologies, so any change that happens on either end will be pushed to the other party. In other words, we expect no omissions to occur.