Asynchronous Chat:
Comparing RESTful and Protocol Buffer Implementations

Serena Booth, Michelle Cone, Nicholas Mahlangu, Tianyu Liu

March 2, 2016

1 Introduction

Since 1999, RESTful communication protocols have taken the web by storm. In the past few years, the web has moved towards using WebSockets, which enable bidirectional client-server communication. In this design specification, we compare the once-wildly popular RESTful communication protocol with alternate protocol buffers used to serialize data passed via WebSockets. 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. Finally, in this paper we compare these communication protocols through a discussion of the results of implementing a web application to facilitate online chat between users and groups using each of these communication protocols.

1.1 Interfaces Under Scrutiny

For our RESTful application, a server and client communicate over HTTP get and post requests. On the client side, these requests are decoded in JavaScript; on the server side, these requests are decoded in Python; they are then dispatched as select, update, insert, or delete SQL requests.

For our Protocol Buffer and WebSocket application, the client and server communicate over HTTP by sending serialized data. We run this application using the Meteor JavaScript framework. All requests on both the client and server are encoded/decoded as Protocol Buffers using JavaScript, as Meteor runs the same JavaScript code for both parties.

1.2 Assumed Environment

For our RESTful application, the client's environment must have access to HTTP passed over port 8080 and must be able to run JavaScript. The server must likewise be able to access HTTP passed over port 8080, via an HTTPServer library. The server must also run Python.

For our Protocol Buffers and WebSocket application, the client and server must have access to HTTP passed over a port (3000 in this case) and must be able to run JavaScript. Furthermore, the server needs to have npm and Meteor installed.

2 RESTful Design

In implementing our web chat application using the REST-ful communication protocol, we opted for a high frequency polling approach, wherein the client opens a webpage which initiates frequent, constant communication with the server by means of a series of GET requests.

We extend the high frequency polling approach to include a success handshake so as to confirm message receipt. In this way, if a message send operation is unsuccessful, we retry sending the message.

This high frequency polling approach operates as follows:

- The client opens a webpage which initiates a GET request using AJAX. This GET request is open until one of the following occur: (a) a success response is received or (b) an error response is received.
- In either (a) or (b), a completion script runs in which the AJAX request is dispatched again after a set amount of time. In the case of (a), there is no delay; in the case of (b), there is a 1000 millisecond delay.
- The server receives the GET request. It looks up messages for the particular client, based on a cookie passed in the request header, and sends an unread message back to the client, along with a success response. Further, the server sends the ids of the message returned in the response header. Lastly, the server updates the MySQL database of messages to indicate that this message is currently being sent.
- On (a), the success response, the client receives a message to display and an id corresponding to that message from the MySQL database. The client then dispatches an additional GET request containing the id of the message.
- On receiving the second confirmation GET request, the server updates the MySQL database to indicate that the message corresponding to a particular id has been received.
- If the server does not receive a success response a minute after sending a message, on the next client-initiated GET message corresponding to a request for new messages, the server updates the MySQL database to indicate that the formerly sent message was not received.

2.1 Resources "Conserved"

Without using WebSockets, which break the RESTful design paradigm, it is not possible to establish a persistent connection between client and server using HTTP. The implications of this are that it isn't possible for a RESTful application to have server-driven events. Hence, in order to create a low-latency application, a polling technique wherein the client repeatedly requests information from the server is necessary. Our approach of high frequency polling, while sub-ideal in comparison to a multi-threaded long-polling approach, results in a fairly low-latency chat application.

We further limit the number of polling requests made by decreasing the frequency of those requests in response to HTTP error responses received; we achieve this by adding a time delay before allowing the client to query the server following an error response.

2.2 Resources "Wasted"

This RESTful application could be improved to conserve CPU usage by implementing a procedure of long polling, via multiple threads running on the server, in place of its current high frequency polling procedure. Further, the overhead used to implement high frequency polling is wasteful, as HTTP headers are repeatedly passed between server and client. While this would likewise be true of long polling, the number of requests would be curbed substantially.

Further, in order to create a chat application, having both server and client-driven events would conserve resources such as CPU and decrease latency, as instead of requiring the client to continually request updates from the server, updates could be automatically routed both to and from the server on reception via bidirectional communication. This could be achieved by using WebSockets, a technique which is becoming increasingly popular on the Internet for communication problems of this variety.

An additional resource wastage occurs in our continual passing of HTTP headers between server and client. Beyond continual passing of HTTP headers, we also continually pass JSON-encoded data between the client and server in response to GET and POST requests. While our headers and data sizes are quite minimal, this is nonetheless wasteful with respect to the amount of data which could be transmitted for such an application in theory. The number of passes of headers could be limited by using long polling instead of high frequency polling. Further, it would be unwise to remove the aspect of state transfer data altogether.

2.3 Failure Conditions

Our system is subject to the following failure models: failstops, crashes, and byzantine failures:

- Failstops

  A failstop could occur in our system when the client submits an HTTP request to the server when the server has halted. This will result in a connection refused error, which informed the client of the server's failure.

- Crashes

  A crash could occur in our system when the client submits an HTTP request to the server when the server is initially online to receive this request. If the server then crashes before responding to the client, the client continually waits for a response from the server, with no knowledge that this failure has occurred. This crash could be circumvented by using a timeout procedure, wherein the client abandons a request after some amount of time. However, this fault tolerance would raise additional questions about the stability of the system, as the server may have already dispatched the

data received from the client to the database, and reflecting this state would become complex.

- 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

  A receive, send, or general omission could occur in our system when a client dispatches an HTTP request to the server, the client awaits an HTTP response. If the server does not receive this message, as by a lossy network, the client then awaits the response. Likewise, if the server receives the message and dispatches a response but, as by a lossy network, the client does not receive the response, the client continues to await the response.

3 Protocol Buffer System Design

Our system was mainly built using *Meteor*, a popular JavaScript application framework, because: the same code runs on the client and the server, it ports 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 an additional 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 primitive types (integer, double, string, boolean, etc.), arrays, and objects – of which, the latter two can have nested properties within.

One implementation issue 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.

Returning to *utils.js*, we see write custom serialization 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.

3.1 Resources Conserved

Meteor relies on two 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; this could be considered as REST for WebSockets. Because these technologies are built directly into *Meteor*, we minimize the number of requests flowing between the client and the server.

These WebSockets provide pervasive connections and do not require the server and client to continually communicate large headers back and forth over the network. Events can be server-initiated, meaning the client is not required to continually request updates; instead, the updates can be sent to the client when available.

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. Given that the latency is not, in fact, zero, this represents an exceptional performance increase. Ultimately, we minimize the size of the data we're sending between the client and server and are thus able to communicate this data faster.

3.2 Resources Wasted

As mentioned in the system design 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.

In our Protocol Buffer implementation, we are also leaving many connections between client and server open, meaning resources must be devoted both client and server-side to this communication, regardless of the client's interactivity with the server. Hence, while in a RESTful system we can scale client-side without necessarily scaling on the server-side, this is patently untrue when using Protocol Buffers and WebSockets.

3.3 Failure Conditions

- Failstops

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

- Byzantine Failures

  As is true of our RESTful application, 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-server communication in our system is WebSocket based so receive, send, and general omissions are possible. A WebSocket will, if a connection is made between client and server and one of those processes halts detectably, send an error and close the connection. However, if the client or server sends information and the would-be receiver halts before reception, one such omission has occurred.

4 Approach Comparison

REST has several advantages over Protocol Buffers and WebScokets: one major advantage of REST is its flexibility and lack of complexity. Protocol Buffers require a system to predefine the format of the data before hand, making it a very static protocol. On the other hand, with REST one can just hit an endpoint and get back data in whatever format is specified at that endpoint. Furthermore this means the organization of an API can be changed drastically without any modifications of the clients. REST also does not require an open connection per client; depending on the application, despite the overhead required to send a RESTful request, this may be beneficial and overall less costly when scaling up.

On the other hand, Protocol Buffers and especially WebSockets are becoming increasingly popular, and with good reason. As discussed above, in a RESTful application, we cannot establish a persistent connection between client and server; this can be achieved via WebSockets. This means we cannot communicate server-driven events and that the client must repeatedly send the server requests, asking if anything has updated. In comparison, Protocol Buffers just affect how the data is transmitted, meaning if a framework like *Meteor* is used, the client can continually be updated with server driven events. Secondly, in a RESTful application, a client can continually wait for a response from a server that may be down. Since Protocol Buffers are just a way of serializing data, this can be circumvented in many cases. Furthermore, Protocol Buffers significantly compress the data being sent, especially in comparison to standard JSON or XML formats.

Ultimately, we cannot recommend that, going forward, one of these technologies is favored over the other. While many applications will warrant a solution of Protocol Buffers and WebSockets and the implied bidirectional data communication, other applications should avoid persistent connections in favor of a RESTful implementation.