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. Our comparison of these communication protocols is substantiated 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, or delete SQL requests.

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.

2 RESTful Design

2.1 High Frequency Polling

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.

2.2 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.

2.3 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.

2.4 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 Design

3.1 Resources "Conserved"

3.2 Resources "Wasted"

3.3 Failure Conditions

4 Approach Comparison

5 Conclusion