An Implementation of a SPDY Web Server in Go

Sherwin Yu

February 26th, 2012

1 Motivation

The purpose of this project is to implement and benchmark a minimalistic SPDY web server using Go. Go was designed as a systems programming language and is relatively new; writing a web server from scratch will be an excellent case study of its effectiveness as a systems programming language. SPDY is also a relatively new experimental protocol, and as of writing, no open source Go implementation of a SPDY server exists.

1.1 Motivation

SPDY is a new network protocol developed at Google designed to reduce web page load latency. It is an application-layer protocol that complements HTTP by improving how HTTP requests and responses are handled. SPDY prioritizes and multiplexes web page resources to minimize the number of connections needed; by multiplexing concurrent streams, many requests and responses are interleaved on the same channel, so greater network efficiency is achieved. SPDY supports different request priorities to prevent the client from blocking requests on limited bandwidth situations. The client can assign different priorities to each item it requests and the server will respond appropriately. SPDY also features HTTP header compression, resulting in less data to transfer.

SPDY was designed to minimize deployment complexity: because it is built on top of TCP, modifications are needed only at the server and client endpoints. Some Google servers already support SPDY; SPDY compatible builds of Chromium and Chrome exist.

1.2 The Go programming language

Go is a systems programming language developed and supported by Google with native support for concurrency. Go was designed with expressiveness, conciseness, and code cleanliness in mind for writing applications on multicore and networked machines. Go is compiled, statically typed, and garbage collected, yet was designed to have the expressiveness of a dynamically typed interpreted language. Go features Goroutines, lightweight functions executing with other
goroutines in the same address space. Goroutines are multiplexed onto multiple OS threads by the Go runtime, hiding many of the complexities of thread creation and management.

2 Goals

2.1 Implementation of a minimalistic SPDY server

Implementation of the core features of SPDY will be supported, depending on how much time is available. Accordingly, a minimalistic testing client will be created as well. Testing can also be done using a SPDY compliant browser Chromium or Chrome.

2.2 Comparison of web server architectures

If time allows, implementation of different web server architectures such as asynchronous, thread per request, and thread pool to handle concurrent requests. Doing so will provide valuable experience on the actual use of Go’s support for concurrency in Goroutines and channels. Benchmarking different servers will also provide insight into Go’s performance; comparison to other HTTP server implementations will also give an idea of Go’s performance relative to other frameworks.

2.3 Network benchmarking

Comparison of HTTP vs SPDY server implementations and network analysis under varying situations will provide insight to SPDY as an actual protocol; though much of this analysis has already been done, this can be an independent verification of Google’s claims and also an analysis of the core features of SPDY.

3 Deliverables

These are the deliverables for this project.

3.1 Code

The code will be written in Go and will be made available and open sourced via GitHub.

3.2 Report

I will complete a report covering the background, problem, goals, methods, and results.