A SPDY Server in Go

Sherwin Yu

April 30 2012

1 Introduction

The purpose of this project was to design and implement a minimalistic SPDY web server using Go. Go was designed as a systems programming language by Google and is relatively new. SPDY is also a relatively new experimental protocol, and prior to this, no up-to-date open source Go implementation of a SPDY server exists. This paper discusses the design and implementation of a SPDY server as well as the lessons learned from using Go for this project.

2 Background

2.1 Shortcomings of HTTP

HTTP and TCP are the common protocols that the internet is built on. HTTP is the application level protocol for hypertext request and response between client and server, TCP is the transport level protocol that provides reliable, in-order guaranteed delivery. HTTP, however, was not designed for low latency communication, and modern web pages have evolved significantly since HTTP 1.1 was designed over a decade ago, with an ever increasing emphasis on real-time communication and speed. The shortcomings of HTTP include:

- Each request/response requires its own TCP connection. This means that HTTP can only request one resource at a time. Furthermore, because modern web pages include dozens of resources, without the Keep-Alive header, each request incurs TCP connection set up and teardown overhead.

- No server-initiated requests. HTTP specifies that only the client can initiate requests, even when the server (frequently) knows what the client will need in advance.

- Uncompressed request and response headers.

- Redundant headers.
2.2 SPDY: A SPeeDY Protocol

SPDY is a new network protocol developed at Google designed to address these shortcomings of HTTP and to ultimately reduce web page load latency. It introduces a "session" layer between the transport and application layers; in practice, this allows concurrent streams between client and server to be multiplexed over a single TCP connection. SPDY prioritizes and multiplexes web page resources to minimize the number 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 in 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 makes the following definitions:

- connection: a end to end TCP connection
- endpoint: either the server or the client
- frame: a header-prefixed sequence of bytes sent over a SPDY session stream – this is the equivalent of a SPDY "packet".
- session: A sequence of freames sent over a single connection. A session can contain multiple streams.
- stream: a bi-directional flow of bytes across a virtual channel within a SPDY session. Multiple streams are multiplexed within a single session.

2.2.1 Framing layer

The SPDY framing layer runs on top of the transport layer; each session has a dedicated TCP connection. Multiple streams can be started in the same session; each stream represents a traditional HTTP request/response pair. There are two types of frames: control frames and data frames. Control frames

Several control frames are supported; the more common and important ones are reviewed below. For further reading, consult.

- SYN_STREAM – indicates the start of a new stream. The stream ID of the new stream is included inside the payload.
- SYN_REPLY – indicates acceptance of a stream creation by the receiver of the SYN_STREAM frame.
- RST_STREAM – indicates abrupt closing of a stream
- HEADERS – the equivalent of the HTTP headers, along with additional options. The name value pair block are sent compressed.
Once a stream has been established, communication can begin. The HTTP response is sent via data frames, which include the stream ID to which the frame belongs. This way, multiple frames from multiple streams (multiple HTTP request/response pairs) can be multiplexed over the same TCP connection asynchronously.

2.2.2 SPDY deployment

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. Google servers are already serving data via SPDY; SPDY compatible builds of Chromium and Chrome exist.

2.3 The Go programming language

Go is a systems programming language developed and supported by Google with native support for concurrency [1]. 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 in parallel 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.

For a more detailed review of Go, consult [2], [3], and [4]

3 SPDY Server Design

A version of the SPDY 2 protocol found at [5] was used as a starting point. From that, I wrote SPDY web implementing the SPDY 2 protocol.

3.1 Protocol.go

The existing SPDY 2 library defines a Frame type:

type Frame struct {
    Header [4]byte
    Flags FrameFlags
    Data  []byte
}

No distinction is made between ControlFrames and DataFrames; frame.isControl() merely checks the first bit of the Header byte array, per the SPDY 2 draft. The length is implicit and calculated only when writing out. Likewise, frame.StreamID() computes the stream ID from the Header byte array.

Protocol also defines helper functions to create ControlFrames, DataFrames, to read frames from an io.Reader and write to an io.Writer.
In addition to handling SPDY frames, protocol.go also handles compression for the headers via the `HeaderReader` and `HeaderWriter` types, which contain references to zlib decompressor/compressors. `HeaderWriter` wraps an existing `Writer`, applying compression to yield the output stream; the reverse applies for decompression.

### 3.2 Server.go

The SPDY server is implemented using a traditional listening socket that spawns additional threads to handle incoming requests. Individual sessions are run in their own goroutines, and each stream writes its data in its own goroutine as well. The implementation of the SPDY web server is contained in server.go.

### 3.3 ListenAndServe

The server’s `ListenAndServe()` function sets up a Go `listener` on the specified port, which is essentially an abstraction for a welcome socket. Upon accepting a new TCP connection, it creates a new `session` with the new connection, and running that session in a new goroutine. This is different from traditional thread based HTTP servers; instead of directly processing the request, a separate session is created which will handle all of the request/responses multiplexed over a single TCP connection.

### 3.4 Session

The `session` type encapsulates a SPDY session. It contains a in and out `Frame` channels. when `session.run()` is called, two separate goroutines are created, one to `receiveFrames()` and one to `sendFrames()`. These goroutines each utilize the `Frame` channels of the session. The code for `session, session.serve(), session.sendFrames(), session.receiveFrames()` is shown below.

```go
type session struct {
    conn net.Conn
    handler http.Handler
    frameIn, frameOut chan Frame
    streams map[uint32]*serverStream
    headerReader *HeaderReader
    headerWriter *HeaderWriter
}

func (sess *session) run() {
    fmt.Println("running new session!")
    defer sess.conn.Close()
    go sess.receiveFrames()
    go sess.sendFrames()
    for {
```
frame := <-sess.frameIn:
switch frame.IsControl() {
    sess.handleControlFrame(frame)
    case false: // is data frame
        sess.handleDataFrame(frame)
    case true: // is control frame
        sess.handleDataFrame)
    }
}

The main loop for a starts goroutines for sending and receiving frames, then
waits for input to sess.frameIn, which it then handles appropriately depending
on the frame type.

func (sess *session) receiveFrames() {
    for {
        frame, err := ReadFrame(sess.conn)
        if err != nil {
            break
        }
        sess.frameIn <- frame
    }
}

func (sess *session) sendFrames() {
    for frame := range sess.frameOut {
        frame.WriteTo(sess.conn)
    }
}

receiveFrames() attempts to read a frame (using the SPDY protocol func-
tion ReadFrame) from the TCP connection socket used to create the session;
when it does, it pushes the newly read frames to the frameIn channel for the
main session loop to handle. This is the only point where we read from the
TCP connection, so data is safely read synchronously. sendFrames() waits for
input on the frameOut channel and writes out frames to the TCP connection
when. Note that this is the only point where writing to the TCP connection
occurs, so data is safely written in order and frames are not interleaved.

3.4.1 Compliance with http.Handler

The server was designed to be compatible with Go’s existing http.Handler
interface type, which provides an abstraction for handling HTTP requests. http.Handler.ServeHTTP(ResponseWriter, *Request) writes response headers and
data to the ResponseWriter. To make the SPDY server compatible with this
existing paradigm, we need only implement the ResponseWriter interface in
the stream object. This includes
• Write([]byte) (int, error)
• Header() Header
• WriteHeader(int)

ServeHTTP calls Write([]byte), which simply writes the header if it has not been written yet; otherwise, breaks bytes up into as many data frames as possible, pushing each one into the frameOut channel.

4 Results

The implementation of the SPDY server was completed and local tests passed. To test the SPDY server with Chrome, I began by serving HTTP normally, and appending an Alternate-Protocol: 5555:npn-spdy/2 to the response header. Then, I set the server to listen for and serve SPDY connections on port 5555. I had a hard time testing the SPDY server with Chrome, however. Using the Alternate-Protocol response header, I successfully advertised the server’s SPDY capabilities on the first HTTP response, and Chrome indeed opens a TCP connection to the alternate port, where the SPDY server is serving. However, this was very occasional – frequently Chrome did not obey the Alternate-Protocol, even when the Alternate-Protocol mapping to my SPDY server showed up in Chrome’s net-internals diagnostics window. Furthermore, the requests that Chrome was sending to the SPDY server were not recognized as belonging to the SPDY spec; these requests were not standard HTTP requests either. A sample message from Chrome is shown below:

22 3 1 0 187 1 0 0 183 3 1 79 158 55 41 236 251 239 175 54
10 33 177 186 51 133 155 57 104 21 55 87 67 249 137 40 251
7 21 162 250 197 248 0 0 72 192 10 192 20 0 136 0 135 0 57
0 56 192 15 192 5 0 132 0 132 0 192 7 192 9 192 17 192 19 0
69 0 68 0 102 0 51 0 50 192 12 192 14 192 2 192 4 0 150 0
65 0 4 0 5 0 47 192 8 192 18 0 22 0 19 192 13 192 3 254
255 0 10 2 1 0 0 69 0 0 0 25 0 23 0 0 20 118 101 108 108
101 105 116 121 46 109 99 46 121 97 108 101 46 101 100
117 255 1 0 1 0 10 0 8 0 6 0 23 0 24 0 25 0 11 0 1 0 0
0 35 0 0 51 116 0 0 0 5 0 5 1 0 0 0 0

5 Discussion

5.1 Lessons

One challenge I encountered was dealing with the open source community maintaining Go. The first version of the protocol.go I found was out of date and unmaintained; I updated it to Go version 1. The code was poorly documented and
a large amount of time was spent understanding the existing SPDY protocol library. The SPDY protocol drafts are well documented but not particularly clear – there are occasional errors, as the drafts are receiving ongoing updates from both the open source community and the Chromium project. These are both issues I expected to encounter when working with such nascent technologies.

The other big challenge I encountered was testing the SPDY implementation. I attempted to test with Chrome, but on occasion, Chrome would stop respecting the Alternate-Protocol response header that indicates the server supports SPDY. It is unclear why this behavior was observed. Furthermore, more puzzling was why Chrome sent data on the SPDY tcp connection that fits neither the HTTP nor the SPDY protocol. This is certainly an area for further work.

5.2 Discussion of Go

One of the goals of this project was to assess Go as a systems programming language. While Go has several idiosyncracies that take a while to get used to, I found writing in Go quite natural as became more familiar with it, and

5.2.1 Variables and types

One of the interesting design choices of Go was that variables are declared first with their name, followed by their type. This is backwards from many languages, yet . I found that this change was easy to get used to, and made statements such as int * a, b considerably clearer. Furthermore, this removes confusion in C/Java like languages regarding how to specify an array of ints: int[4] arr or int arr[4]? In Go, this is simply [4]int: an array of four ints.

Having slices and maps as primitive types is a excellent choice; this bypasses the verbosity of Java and data structures, which are implemented as objects, and makes Go feel much more like a dynamic language (e.g., Python or Ruby). Indeed, much of Go’s syntax is borrowed from Python and Ruby, such as using arr[begin:end] as a slicing operator.

Go’s use of clear types (uint8, uint16, uint32, uint64, int8, int16, int32, int64) was very helpful when dealing with byte level operations, such as for packing bytes into SPDY frames. For a systems programming language, clear and efficient syntax for control of memory representation for integers is important, and Go succeeds in this regard. Furthermore, when such specificity is not needed, Go focuses on simplicity and clarity: x:=7 automatically assumes int (represented as int32) as the type.

5.2.2 Implicit interfaces and Embedding

Go utilizes implicit interfaces for grouping. For example, one can define an interface Reader which has the Read() method. Then, any struct that implements the Read() method can be used as a Reader, and that struct can be used
anywhere a Reader interface is expected. One criticism of Go was that the non-
implicit hierarchy made reading code more difficult. Looking at the definition of
a ByteReader does not immediately tell me that that it implements the Reader
interface; I have to be familiar with the Reader interface and also inspect in-
dividual methods of ByteReader until I confirm that it has implemented the
Reader interface method set. This may become easier to follow with more expe-
rience and familiarity with common interface types; for example, Go maintains
a convention to name single-method interfaces the name of the method + er,
e.g. Writer.

Furthermore, Go does not support inheritance; instead, it uses embedding,
which allows object composition via including anonymous fields in the struct
definition. For example, the following code creates a ReadWriter interface in
which Reader and Writer are anonymous fields; this promotes the Read and
Write methods from Reader and Writer to the embedding interface:

```
type ReadWriter interface {
    Reader
    Writer
}
```

This approach, while considerably different from inheritance, appeared just
as expressive in my use. It was slightly unnatural at first but after becoming
familiar with it, I could not identify any major drawbacks.

### 5.2.3 Concurrency

Go goroutines, which are light weight threads that are simple to execute. Go
treats function literal as closures, and function pointers can be passed around
as objects. This is convenient for general functional programming, but is par-
ticularly useful for use with the `go` keyword to trigger go routines; starting a
new thread can be as simple as appending `go` in front of a function literal.

This is considerably shorter and clearer than using threads in other languages
such as Java. For example, upon creating a new stream, writing the response
(chunked into data frames) can be done asynchronously in the background by
starting a new goroutine with an anonymous function:

```
sess.streams[stream.id] = stream
go func() {
    sess.handler.ServeHTTP(stream, stream.Request())
    stream.finish()
}()
```

I also found channels to be a very light weight and elegant model for writing
multithreaded code; for example, the session main loop uses channels to com-
 municate between the `sendFrames()` and `receiveFrames()` goroutines. Go’s
philosophy regarding shared memory is ”don’t communicate by sharing mem-
ory; share memory by communicating” and channels exemplify this. By having
each shared variable accessed directly by only one goroutine, and relying on communication for sharing, writing multithreaded code becomes significantly neater. Furthermore, when more precise control is necessary, Go’s sync package contains other utilities like locks and mutexes.

6 Conclusions

I designed and implemented a SPDY server in Go. I was unable to get Chrome to successfully communicate with the SPDY server. Further work would involve solving that communication error and also involve performing benchmarks for the SPDY server in go. Of particular interest is the performance of goroutines: do lightweight goroutines in the thread-per-request server model suffer from the same problems of using threads? Overall, I found that Go was a very expressive language for systems programming purposes; in particular, the support for concurrency was painless and elegant. While both SPDY and Go are new technologies, community support is growing and exciting developments await in the future.

6.1 Code

References