HTTP

Introduction

The World Wide Web is a major distributed system, with millions of users. A site may become a Web host by running an HTTP server. While Web clients are typically users with a browser, there are many other "user agents" such as web spiders, web application clients and so on.

The Web is built on top of the HTTP (Hyper-Text Transport Protocol) which is layered on top of TCP. HTTP has been through three publically available versions, but the latest - version 1.1 - is now the most commonly used.

In this chapter we give an overview of HTTP, followed by the Go APIs to manage HTTP connections.

Overview of HTTP

URLs and resources

URLs specify the location of a *resource*. A resource is often a static file, such as an HTML document, an image, or a sound file. But increasingly, it may be a dynamically generated object, perhaps based on information stored in a database.

When a user agent requests a resource, what is returned is not the resource itself, but some *representation* of that resource. For example, if the resource is a static file, then what is sent to the user agent is a copy of the file.

Multiple URLs may point to the same resource, and an HTTP server will return appropriate representations of the resource for each URL. For example, an company might make product information available both internally and externally using different URLs for the same product. The internal representation of the product might include information such as internal contact officers for the product, while the external representation might include the location of stores selling the product.

This view of resources means that the HTTP protocol can be fairly simple and straightforward, while an HTTP server can be arbitrarily complex. HTTP has to deliver requests from user agents to servers and return a byte stream, while a server might have to do any amount of processing of the request.

HTTP characteristics

HTTP is a stateless, connectionless, reliable protocol. In the simplest form, each request from a user agent is handled reliably and then the connection is broken. Each request involves a separate TCP connection, so if many reources are required (such as images embedded in an HTML page) then many TCP connections have to be set up and torn down in a short space of time.

Thera are many optimisations in HTTP which add complexity to the simple structure, in order to create a more efficient and reliable protocol.

Versions

There are 3 versions of HTTP

- Version 0.9 - totally obsolete
- Version 1.0 - almost obsolete
- Version 1.1 - current

Each version must understand requests and responses of earlier versions.

HTTP 0.9

Request format

```
Request = Simple-Request

Simple-Request = "GET" SP Request-URI CRLF
```

Response format

A response is of the form

```
Response = Simple-Response

Simple-Response = [Entity-Body]
```

HTTP 1.0

This version added much more information to the requests and responses. Rather than "grow" the 0.9 format, it was just left alongside the new version.

Request format

The format of requests from client to server is

```
Request = Simple-Request | Full-Request

Simple-Request = "GET" SP Request-URI CRLF

Full-Request = Request-Line
               *(General-Header
               | Request-Header
               | Entity-Header)
               CRLF
               [Entity-Body]
```

A Simple-Request is an HTTP/0.9 request and must be replied to by a Simple-Response.

A Request-Line has format

```
Request-Line = Method SP Request-URI SP HTTP-Version CRLF
```

where

```
Method = "GET" | "HEAD" | POST |
         extension-method
```

e.g.

```
GET http://jan.newmarch.name/index.html HTTP/1.0
```

Response format

A response is of the form

```
Response = Simple-Response | Full-Response

Simple-Response = [Entity-Body]

Full-Response = Status-Line
               *(General-Header
               | Response-Header
               | Entity-Header)
               CRLF
               [Entity-Body]
```

The Status-Line gives information about the fate of the request:

```
Status-Line = HTTP-Version SP Status-Code SP Reason-Phrase CRLF
```

e.g.

```
HTTP/1.0 200 OK
```

The codes are

```
Status-Code =    "200" ; OK
               | "201" ; Created
               | "202" ; Accepted
               | "204" ; No Content
               | "301" ; Moved permanently
               | "302" ; Moved temporarily
               | "304" ; Not modified
               | "400" ; Bad request
```

```
                | "401" ; Unauthorised
                | "403" ; Forbidden
                | "404" ; Not found
                | "500" ; Internal server error
                | "501" ; Not implemented
                | "502" ; Bad gateway
                | "503" | Service unavailable
                | extension-code
```

The Entity-Header contains useful information about the Entity-Body to follow

```
Entity-Header =Allow
                | Content-Encoding
                | Content-Length
                | Content-Type
                | Expires
                | Last-Modified
                | extension-header
```

For example

```
HTTP/1.1 200 OK
Date: Fri, 29 Aug 2003 00:59:56 GMT
Server: Apache/2.0.40 (Unix)
Accept-Ranges: bytes
Content-Length: 1595
Connection: close
Content-Type: text/html; charset=ISO-8859-1
```

HTTP 1.1

HTTP 1.1 fixes many problems with HTTP 1.0, but is more complex because of it. This version is done by extending or refining the options available to HTTP 1.0. e.g.

- there are more commands such as TRACE and CONNECT
- you should use absolute URLs, particularly for connecting by proxies e.g

```
        GET http://www.w3.org/index.html HTTP/1.1
```

- there are more attributes such as If-Modified-Since, also for use by proxies

The changes include

- hostname identification (allows virtual hosts)
- content negotiation (multiple languages)
- persistent connections (reduces TCP overheads - this is very messy)
- chunked transfers
- byte ranges (request parts of documents)
- proxy support

The 0.9 protocol took one page. The 1.0 protocol was described in about 20 pages. 1.1 takes 120 pages.

Simple user-agents

User agents such as browsers make requests and get responses. The response type is

```
type Response struct {
    Status     string // e.g. "200 OK"
    StatusCode int    // e.g. 200
    Proto      string // e.g. "HTTP/1.0"
    ProtoMajor int    // e.g. 1
    ProtoMinor int    // e.g. 0

    RequestMethod string // e.g. "HEAD", "CONNECT", "GET", etc.

    Header map[string]string

    Body io.ReadCloser

    ContentLength int64

    TransferEncoding []string

    Close bool

    Trailer map[string]string
}
```

We shall examine this data structure through examples. The simplest request is from a user agent is "HEAD" which asks for information about a resource and its HTTP server. The function

```
func Head(url string) (r *Response, err os.Error)
```

can be used to make this query.

The status of the response is in the response field `Status`, while the field `Header` is a map of the header fields in the HTTP response. A program to make this request and display the results is

```
/* Head
 */

package main

import (
        "fmt"
        "net/http"
```

```
        "os"
)

func main() {
        if len(os.Args) != 2 {
                fmt.Println("Usage: ", os.Args[0], "host:port")
                os.Exit(1)
        }
        url := os.Args[1]

        response, err := http.Head(url)
        if err != nil {
                fmt.Println(err.Error())
                os.Exit(2)
        }

        fmt.Println(response.Status)
        for k, v := range response.Header {
                fmt.Println(k+":", v)
        }

        os.Exit(0)
}
```

When run against a resource as in `Head http://www.golang.com/` it prints something like

```
200 OK
Content-Type: text/html; charset=utf-8
Date: Tue, 14 Sep 2010 05:34:29 GMT
Cache-Control: public, max-age=3600
Expires: Tue, 14 Sep 2010 06:34:29 GMT
Server: Google Frontend
```

Usually, we are want to retrieve a resource rather than just get information about it. The "GET" request will do this, and this can be done using

```
func Get(url string) (r *Response, finalURL string, err os.Error)
```

The content of the response is in the response field `Body` which is of type `io.ReadCloser`. We can print the content to the screen with the following program

```
/* Get
 */

package main

import (
        "fmt"
        "net/http"
```

```
        "net/http/httputil"
        "os"
        "strings"
)

func main() {
        if len(os.Args) != 2 {
                fmt.Println("Usage: ", os.Args[0], "host:port")
                os.Exit(1)
        }
        url := os.Args[1]

        response, err := http.Get(url)
        if err != nil {
                fmt.Println(err.Error())
                os.Exit(2)
        }

        if response.Status != "200 OK" {
                fmt.Println(response.Status)
                os.Exit(2)
        }

        b, _ := httputil.DumpResponse(response, false)
        fmt.Print(string(b))

        contentTypes := response.Header["Content-Type"]
        if !acceptableCharset(contentTypes) {
                fmt.Println("Cannot handle", contentTypes)
                os.Exit(4)
        }

        var buf [512]byte
        reader := response.Body
        for {
                n, err := reader.Read(buf[0:])
                if err != nil {
                        os.Exit(0)
                }
                fmt.Print(string(buf[0:n]))
        }
        os.Exit(0)
}

func acceptableCharset(contentTypes []string) bool {
        // each type is like [text/html; charset=UTF-8]
        // we want the UTF-8 only
        for _, cType := range contentTypes {
                if strings.Index(cType, "UTF-8") != -1 {
                        return true
                }
        }
        return false
}
```

Note that there are important character set issues of the type discussed in the previous chapter. The server will deliver the content using some character set encoding, and possibly some transfer encoding. Usually this is a matter of negotiation between user agent and server, but the simple `Get` command that we are using does not include the user agent component of the negotiation. So the server can send whatever character encoding it wishes.

At the time of first writing, I was in China. When I tried this program on `www.google.com`, Google's server tried to be helpful by guessing my location and sending me the text in the Chinese character set Big5! How to tell the server what character encoding is okay for me is discussed later.

Configuring HTTP requests

Go also supplies a lower-level interface for user agents to communicate with HTTP servers. As you might expect, not only does it give you more control over the client requests, but requires you to spend more effort in building the requests. However, there is only a small increase.

The data type used to build requests is the type `Request`. This is a complex type, and is given in the Go documentation as

```
type Request struct {
    Method     string // GET, POST, PUT, etc.
    RawURL     string // The raw URL given in the request.
    URL        *URL   // Parsed URL.
    Proto      string // "HTTP/1.0"
    ProtoMajor int    // 1
    ProtoMinor int    // 0


    // A header maps request lines to their values.
    // If the header says
    //
    //  accept-encoding: gzip, deflate
    //  Accept-Language: en-us
    //  Connection: keep-alive
    //
    // then
    //
    //  Header = map[string]string{
    //          "Accept-Encoding": "gzip, deflate",
    //          "Accept-Language": "en-us",
    //          "Connection": "keep-alive",
    //  }
    //
    // HTTP defines that header names are case-insensitive.
    // The request parser implements this by canonicalizing the
    // name, making the first character and any characters
```

```
    // following a hyphen uppercase and the rest lowercase.
    Header map[string]string

    // The message body.
    Body io.ReadCloser

    // ContentLength records the length of the associated content.
    // The value -1 indicates that the length is unknown.
    // Values >= 0 indicate that the given number of bytes may be read from
Body.
    ContentLength int64

    // TransferEncoding lists the transfer encodings from outermost to
innermost.
    // An empty list denotes the "identity" encoding.
    TransferEncoding []string

    // Whether to close the connection after replying to this request.
    Close bool

    // The host on which the URL is sought.
    // Per RFC 2616, this is either the value of the Host: header
    // or the host name given in the URL itself.
    Host string

    // The referring URL, if sent in the request.
    //
    // Referer is misspelled as in the request itself,
    // a mistake from the earliest days of HTTP.
    // This value can also be fetched from the Header map
    // as Header["Referer"]; the benefit of making it
    // available as a structure field is that the compiler
    // can diagnose programs that use the alternate
    // (correct English) spelling req.Referrer but cannot
    // diagnose programs that use Header["Referrer"].
    Referer string

    // The User-Agent: header string, if sent in the request.
    UserAgent string

    // The parsed form. Only available after ParseForm is called.
    Form map[string][]string

    // Trailer maps trailer keys to values.  Like for Header, if the
    // response has multiple trailer lines with the same key, they will be
    // concatenated, delimited by commas.
    Trailer map[string]string
}
```

There is a lot of information that can be stored in a request. You do not need to fill in all fields, only those of interest. The simplest way to create a request with default values is by for example

```
```

```
request, err := http.NewRequest("GET", url.String(), nil)
```

Once a request has been created, you can modify fields. For example, to specify that you only wish to receive UTF-8, add an "Accept-Charset" field to a request by

```
request.Header.Add("Accept-Charset", "UTF-8;q=1, ISO-8859-1;q=0")
```

(Note that the default set ISO-8859-1 always gets a value of one unless mentioned explicitly in the list.).

A client setting a charset request is simple by the above. But there is some confusion about what happens with the server's return value of a charset. The returned resource *should* have a `Content-Type` which will specify the media type of the content such as `text/html`. If appropriate the media type should state the charset, such as `text/html; charset=UTF-8`. If there is no charset specification, then according to the HTTP specification it should be treated as the default ISO8859-1 charset. But the HTML 4 specification states that since many servers don't conform to this, then you can't make any assumptions.

If there is a charset specified in the server's `Content-Type`, then assume it is correct. if there is none specified, since 50% of pages are in UTF-8 and 20% are in ASCII then it is safe to assume UTF-8. Only 30% of pages may be wrong :-(.

The Client object

To send a request to a server and get a reply, the convenience object `Client` is the easiest way. This object can manage multiple requests and will look after issues such as whether the server keeps the TCP connection alive, and so on.

This is illustrated in the following program

```
/* ClientGet
 */

package main

import (
        "fmt"
        "net/http"
        "net/url"
        "os"
        "strings"
```

```
)

func main() {
        if len(os.Args) != 2 {
                fmt.Println("Usage: ", os.Args[0], "http://host:port/page")
                os.Exit(1)
        }
        url, err := url.Parse(os.Args[1])
        checkError(err)

        client := &http.Client{}

        request, err := http.NewRequest("GET", url.String(), nil)
        // only accept UTF-8
        request.Header.Add("Accept-Charset", "UTF-8;q=1, ISO-8859-1;q=0")
        checkError(err)

        response, err := client.Do(request)
        if response.Status != "200 OK" {
                fmt.Println(response.Status)
                os.Exit(2)
        }

        chSet := getCharset(response)
        fmt.Printf("got charset %s\n", chSet)
        if chSet != "UTF-8" {
                fmt.Println("Cannot handle", chSet)
                os.Exit(4)
        }

        var buf [512]byte
        reader := response.Body
        fmt.Println("got body")
        for {
                n, err := reader.Read(buf[0:])
                if err != nil {
                        os.Exit(0)
                }
                fmt.Print(string(buf[0:n]))
        }

        os.Exit(0)
}

func getCharset(response *http.Response) string {
        contentType := response.Header.Get("Content-Type")
        if contentType == "" {
                // guess
                return "UTF-8"
        }
        idx := strings.Index(contentType, "charset:")
        if idx == -1 {
                // guess
                return "UTF-8"
        }
        return strings.Trim(contentType[idx:], " ")
}
```

```
func checkError(err error) {
        if err != nil {
                fmt.Println("Fatal error ", err.Error())
                os.Exit(1)
        }
}
```

Proxy handling

Simple proxy

HTTP 1.1 laid out how HTTP should work through a proxy. A "GET" request should be made to a proxy. However, the URL requested should be the full URL of the destination. In addition the HTTP header should contain a "Host" field, set to the proxy. As long as the proxy is configured to pass such requests through, then that is all that needs to be done.

Go considers this to be part of the HTTP transport layer. To manage this it has a class `Transport`. This contains a field which can be set to a *function* that returns a URL for a proxy. If we have a URL as a string for the proxy, the appropriate transport object is created and then given to a client object by

```
proxyURL, err := url.Parse(proxyString)
transport := &http.Transport{Proxy: http.ProxyURL(proxyURL)}
client := &http.Client{Transport: transport}
```

The client can then continue as before.

The following program illustrates this:

```
/* ProxyGet
 */

package main

import (
        "fmt"
        "io"
        "net/http"
        "net/http/httputil"
        "net/url"
        "os"
)

func main() {
        if len(os.Args) != 3 {
```

```
                fmt.Println("Usage: ", os.Args[0], "http://proxy-host:port
http://host:port/page")
                os.Exit(1)
        }
        proxyString := os.Args[1]
        proxyURL, err := url.Parse(proxyString)
        checkError(err)
        rawURL := os.Args[2]
        url, err := url.Parse(rawURL)
        checkError(err)

        transport := &http.Transport{Proxy: http.ProxyURL(proxyURL)}
        client := &http.Client{Transport: transport}

        request, err := http.NewRequest("GET", url.String(), nil)

        dump, _ := httputil.DumpRequest(request, false)
        fmt.Println(string(dump))

        response, err := client.Do(request)

        checkError(err)
        fmt.Println("Read ok")

        if response.Status != "200 OK" {
                fmt.Println(response.Status)
                os.Exit(2)
        }
        fmt.Println("Reponse ok")

        var buf [512]byte
        reader := response.Body
        for {
                n, err := reader.Read(buf[0:])
                if err != nil {
                        os.Exit(0)
                }
                fmt.Print(string(buf[0:n]))
        }

        os.Exit(0)
}

func checkError(err error) {
        if err != nil {
                if err == io.EOF {
                        return
                }
                fmt.Println("Fatal error ", err.Error())
                os.Exit(1)
        }
}
```

If you have a proxy at, say, XYZ.com on port 8080, test this by

```
```

```
go run ProxyGet.go http://XYZ.com:8080/ http://www.google.com
```

If you don't have a suitable proxy to test this, then download and install the Squid proxy to your own computer.

The above program used a known proxy passed as an argument to the program. There are many ways in which proxies can be made known to applications. Most browsers have a configuration menu in which you can enter proxy information: such information is not available to a Go application. Some applications may get proxy information from an `autoproxy.pac` file somewhere in your network: Go does not (yet) know how to parse these JavaScript files and so cannot use them. Linux systems using Gnome have a configuration system called `gconf` in which proxy information can be stored: Go cannot access this. *But* it can find proxy information if it is set in operating system environment variables such as HTTP_PROXY or http_proxy using the function

```
func ProxyFromEnvironment(req *Request) (*url.URL, error)
```

If your programs are running in such an environment you can use this function instead of having to explicitly know the proxy parameters.

Authenticating proxy

Some proxies will require authentication, by a user name and password in order to pass requests. A common scheme is "basic authentication" in which the user name and password are concatenated into a string "user:password" and then BASE64 encoded. This is then given to the proxy by the HTTP request header "Proxy-Authorisation" with the flag that it is the basic authentication

The following program illlustrates this, adding the Proxy-Authentication header to the previous proxy program:

```
/* ProxyAuthGet
 */

package main

import (
        "encoding/base64"
        "fmt"
        "io"
        "net/http"
        "net/http/httputil"
```

```
        "net/url"
        "os"
)

const auth = "jannewmarch:mypassword"

func main() {
        if len(os.Args) != 3 {
                fmt.Println("Usage: ", os.Args[0], "http://proxy-host:port
http://host:port/page")
                os.Exit(1)
        }
        proxy := os.Args[1]
        proxyURL, err := url.Parse(proxy)
        checkError(err)
        rawURL := os.Args[2]
        url, err := url.Parse(rawURL)
        checkError(err)

        // encode the auth
        basic := "Basic " + base64.StdEncoding.EncodeToString([]byte(auth))

        transport := &http.Transport{Proxy: http.ProxyURL(proxyURL)}
        client := &http.Client{Transport: transport}

        request, err := http.NewRequest("GET", url.String(), nil)

        request.Header.Add("Proxy-Authorization", basic)
        dump, _ := httputil.DumpRequest(request, false)
        fmt.Println(string(dump))

        // send the request
        response, err := client.Do(request)

        checkError(err)
        fmt.Println("Read ok")

        if response.Status != "200 OK" {
                fmt.Println(response.Status)
                os.Exit(2)
        }
        fmt.Println("Reponse ok")

        var buf [512]byte
        reader := response.Body
        for {
                n, err := reader.Read(buf[0:])
                if err != nil {
                        os.Exit(0)
                }
                fmt.Print(string(buf[0:n]))
        }

        os.Exit(0)
}

func checkError(err error) {
```

```
        if err != nil {
                if err == io.EOF {
                        return
                }
                fmt.Println("Fatal error ", err.Error())
                os.Exit(1)
        }
}
```

HTTPS connections by clients

For secure, encrypted connections, HTTP uses TLS which is described in the chapter on security. The protocol of HTTP+TLS is called HTTPS and uses https:// urls instead of http:// urls.

Servers are required to return valid X.509 certificates before a client will accept data from them. If the certificate is valid, then Go handles everything under the hood and the clients given previously run okay with https URLs.

Many sites have invalid certificates. They may have expired, they may be self-signed instead of by a recognised Certificate Authority or they may just have errors (such as having an incorrect server name). Browsers such as Firefox put a big warning notice with a "Get me out of here!" button, but you can carry on at your risk - which many people do.

Go presently bails out when it encounters certificate errors. There is cautious support for carrying on but I haven't got it working yet. So there is no current example for "carrying on in the face of adversity :-)". Maybe later.

Servers

The other side to building a client is a Web server handling HTTP requests. The simplest - and earliest - servers just returned copies of files. However, any URL can now trigger an arbitrary computation in current servers.

File server

We start with a basic file server. Go supplies a `multi-plexer`, that is, an object that will read and interpret requests. It hands out requests to `handlers` which run in their own thread. Thus much of the work of reading HTTP requests, decoding them and branching to suitable functions in their own thread is done for us.

For a file server, Go also gives a `FileServer` object which knows how to deliver files from the local file system. It takes a "root" directory which is the top of a file tree in

the local system, and a pattern to match URLs against. The simplest pattern is "/" which is the top of any URL. This will match all URLs.

An HTTP server delivering files from the local file system is almost embarrassingly trivial given these objects. It is

```
/* File Server
 */

package main

import (
        "fmt"
        "net/http"
        "os"
)

func main() {
        // deliver files from the directory /var/www
        //fileServer := http.FileServer(http.Dir("/var/www"))
        fileServer := http.FileServer(http.Dir("/home/httpd/html/"))

        // register the handler and deliver requests to it
        err := http.ListenAndServe(":8000", fileServer)
        checkError(err)
        // That's it!
}

func checkError(err error) {
        if err != nil {
                fmt.Println("Fatal error ", err.Error())
                os.Exit(1)
        }
}
```

This server even delivers "404 not found" messages for requests for file resources that don't exist!

Handler functions

In this last program, the handler was given in the second argument to `ListenAndServe`. Any number of handlers can be registered first by calls to `Handle` or `handleFunc`, with signatures

```
func Handle(pattern string, handler Handler)
func HandleFunc(pattern string, handler func(*Conn, *Request))
```

The second argument to `HandleAndServe` could be `nil`, and then calls are dispatched to all registered handlers. Each handler should have a different URL pattern. For

example, the file handler might have URL pattern "/" while a function handler might have URL pattern "/cgi-bin". A more specific pattern takes precedence over a more general pattern.

Common CGI programs are `test-cgi` (written in the shell) or `printenv` (written in Perl) which print the values of the environment variables. A handler can be written to work in a similar manner.

```
/* Print Env
 */

package main

import (
        "fmt"
        "net/http"
        "os"
)

func main() {
        // file handler for most files
        fileServer := http.FileServer(http.Dir("/var/www"))
        http.Handle("/", fileServer)

        // function handler for /cgi-bin/printenv
        http.HandleFunc("/cgi-bin/printenv", printEnv)

        // deliver requests to the handlers
        err := http.ListenAndServe(":8000", nil)
        checkError(err)
        // That's it!
}

func printEnv(writer http.ResponseWriter, req *http.Request) {
        env := os.Environ()
        writer.Write([]byte("<h1>Environment</h1>\n<pre>"))
        for _, v := range env {
                writer.Write([]byte(v + "\n"))
        }
        writer.Write([]byte("</pre>"))
}

func checkError(err error) {
        if err != nil {
                fmt.Println("Fatal error ", err.Error())
                os.Exit(1)
        }
}
```

Note: for simplicity this program does not deliver well-formed HTML. It is missing html, head and body tags.

Using the `cgi-bin` directory in this program is a bit cheeky: it doesn't call an external program like CGI scripts do. It just calls a Go function. Go does have the ability to call external programs using `os.ForkExec`, but does not yet have support for dynamically linkable modules like Apache's `mod_perl`

Bypassing the default multiplexer

HTTP requests received by a Go server are usually handled by a multiplexer the examines the path in the HTTP request and calls the appropriate file handler, etc. You can define your own handlers. These can either be registered with the default multiplexer by calling `http.HandleFunc` which takes a pattern and a function. The functions such as `ListenAndServe` then take a `nil` handler function. This was done in the last example.

If you want to take over the multiplexer role then you can give a non-zero function as the handler function. This function will then be totally responsible for managing the requests and responses.

The following example is trivial, but illustrates the use of this: the multiplexer function simply returns a "204 No content" for *all* requests:

```
/* ServerHandler
 */

package main

import (
        "net/http"
)

func main() {

        myHandler := http.HandlerFunc(func(rw http.ResponseWriter, request
*http.Request) {
                // Just return no content - arbitrary headers can be set,
arbitrary body
                rw.WriteHeader(http.StatusNoContent)
        })

        http.ListenAndServe(":8080", myHandler)
}
```

Arbitrarily complex behaviour can be built, of course.

Source: http://jan.newmarch.name/go/http/chapter-http.html