How to Write an HTTP Proxy for I2P in Go

In this short guide, I will show you how to create a standalone HTTP proxy which will only ever make connections over the I2P network as an introduction to I2P application development in Go. Although most of the readers will have an understanding of HTTP proxies already, and there is already an HTTP proxy to I2P in the core I2P software, there is a method to this madness:

Why another Go HTTP Proxy Tutorial?

There are tons of tutorials out there to write HTTP proxies in Go(1). There is a ton of example code out there you can just copy-and-paste, modify to suit your needs, and use. So why another one now?

Well one reason there are so many is that Go makes it very easy to write a reasonably reliable HTTP proxy. It's a simple project that shows why Go can be good at helping you do the things you need, if what you need is an HTTP client or server. Likewise, the HTTP proxy makes a good introduction to why I2P can be good at helping you do the things you need, if what you need to do is make HTTP requests in privacy. So it exists to bridge the mental gap between the Go application and I2P client in a way which is accessible to people who learned Go from the internet like me.

warning: Be careful and do not use this example code unmodified in your real application. It deliberately references an earlier state of the application in the root of this repository. This is a teaching tool and not production code.

What is an HTTP Proxy?

At it's simplest it's two things, it's an HTTP server, and an HTTP Client. When the HTTP server recieves a request, it handles it by forwarding it to the HTTP client, which then retrieves and returns it to the server, which forwards it back to the requestor. In-between recieving the request and forwarding it to the client, the server can make changes to how the requests are handled. So in our example, we'll be leaving the example HTTP server in place, but modifying the HTTP client to handle requests by routing them to I2P.

Step One: Setting up the HTTP Server Structure

Creating a custom HTTP server in Go is easy. We just need to create a type called Proxy, which implements the http.Server that forwards you to the client. First, import the goSam library.

```
import (
    "github.com/eyedeekay/goSam"
)
```

goSam implements a transport which is compatible with Go's http.Client. The proxy will then need to contain a goSam.Client and an http.Client.

```
type Proxy struct {
    Sam    *goSam.Client
    Client *http.Client
}
```

In order to implement the http Server interface, we need the ServeHTTP function within the Proxy structure, which has the following signature:

```
    func (p *Proxy) ServeHTTP(wr http.ResponseWriter, req *http.Request)
```

Where req is the request made by the user, and wr is the the stream where the response will be written and returned to the user.

When you create an instance of this struct, you should make sure to first set up an instance of goSam.Client and pass it to the struct, like this:

```
    sam, err := goSam.NewClientFromOptions(
        goSam.SetUnpublished(true),
    )
    if err != nil {
        log.Fatal(err)
    }
    handler := &i2phttpproxy.Proxy{
        Sam: sam,
    }
```

Note that goSam has been set up to use an Unpublished leaseset in this case, because it will be used as a client and not a service and doesn't need to be discovered until it starts communicating with a service.

Step Two: Anonymize the application

If you're developing a privacy-aware application, it's important to anonymize more than just the connection, but also the application as well. In the case of an HTTP proxy, you should probably at least scrub some headers out. To do this, make a slice with the headers you want to delete at the first hop, and a function which deletes them from the underlying structure.

```
    var hopHeaders = []string{
        "Accept-Language",
        "Connection",
        "Keep-Alive",
        "Proxy-Authenticate",
        "Proxy-Authorization",
        "Proxy-Connection",
        "Trailers",
        "Upgrade",
        "X-Forwarded-For",
        "X-Real-IP",
    }

    func delHopHeaders(header http.Header) {
        for _, h := range hopHeaders {
            header.Del(h)
        }
        ....
```

If you want, you can also re-write the user agent string in this function as well.

```
        ....
        if header.Get("User-Agent") != "MYOB/6.66 (AN/ON)" {
            header.Set("User-Agent", "MYOB/6.66 (AN/ON)")
        }
    }
```

Because our proxy will only be used for I2P addresses, we'll want ServeHTTP to ignore non-I2P addresses. We can do this by examining and dropping the request when the server recieves it, like this:

```
    func (p *Proxy) ServeHTTP(wr http.ResponseWriter, req *http.Request) {
        if req.URL.Scheme != "http" && !strings.HasSuffix(req.URL.Host, ".i2p") {
            msg := "unsupported protocal scheme " + req.URL.Scheme
            http.Error(wr, msg, http.StatusBadRequest)
            log.Println(msg)
            return
        }
    ...
```

Once you're done, add the header-scrubbing function to the ServeHTTP function.

```
    ...
        delHopHeaders(req.Header)
    ...
```

Step Three: Actually handle the request

Now that we've got the application anonymized, it's time to handle the request.

```
    ...
        p.get(wr, req)
    }
```

To keep the steps of the procedure cleanly separated, I chose to contain this in it's own function. It just carries the same signature of the ServeHTTP function, which

```
    func (p *Proxy) get(wr http.ResponseWriter, req *http.Request) {
        req.RequestURI = ""
        resp, err := p.Client.Do(req)
        if err != nil {
            log.Println("ServeHTTP:", err)
            return
        }
        defer resp.Body.Close()

        wr.WriteHeader(resp.StatusCode)
        io.Copy(wr, resp.Body)
    }
```

But wait! Where's all the I2P stuff? What will this do if it's not connected to I2P? It will just be an HTTP Proxy that doesn't work, because it's already blacklisting non-I2P addresses.

Step Four: Set up the I2P Client

Setting up the I2P Client is really easy, but a good example would be a little long so I decided to extract it out to it's own function and walk through the steps. In the end, we need to come up with a new http.Client which uses i2p as a Transport. In order to ensure that it uses the same settings as the goSam client, the function is part of the Proxy struct.

```
    func (p *Proxy) NewClient() *http.Client {
        return &http.Client{
```

In order to use the resulting http.Client with i2p, you'll need to alter it's DialContext function to use the one from goSam instead.

```
            Transport: &http.Transport{
                DialContext:          p.Sam.DialContext,
            ....
```

It may also be helpful to limit connections, set some timeouts(Go by default expects you to set timeouts). Some example settings you might change would be:

```
            ....
                MaxConnsPerHost:      1,
                MaxIdleConns:         0,
                MaxIdleConnsPerHost:  1,
                DisableKeepAlives:     false,
                ResponseHeaderTimeout: time.Second * 600,
                IdleConnTimeout:       time.Second * 300,
            },
            CheckRedirect: nil,
            Timeout:       time.Second * 600,
        }
    }
```

Step Five: Create the main() function

Now it's finally time to put it all together and make our code runnable. First, lets make it accept a flag which allows the user to determine which port it runs on. I don't know what ports other people use:

```
    func main() {
        var addr = flag.String("addr", "127.0.0.1:7950", "The addr of the application.")
        flag.Parse()
    ...
```

Next, set up the goSam client and hand it off to a new instance of *Proxy:

```
    ...
        sam, err := goSam.NewClientFromOptions(
            goSam.SetHost("127.0.0.1"),
            goSam.SetPort("7656"),
            goSam.SetUnpublished(true),
            goSam.SetInLength(uint(2)),
            goSam.SetOutLength(uint(2)),
            goSam.SetInQuantity(uint(1)),
            goSam.SetOutQuantity(uint(1)),
            goSam.SetInBackups(uint(1)),
            goSam.SetOutBackups(uint(1)),
            goSam.SetReduceIdle(true),
            goSam.SetReduceIdleTime(uint(2000000)),
        )
        if err != nil {
            log.Fatal(err)
        }
        handler := &Proxy{
            Sam: sam,
        }
    ...
```

Use that SAM connection to set up an HTTP client:

```
    ...
        handler.Client = handler.NewClient()
    ...
```

And serve it up. It's just that easy.

```
    ...
        log.Println("Starting proxy server on", *addr)
        if err := http.ListenAndServe(*addr, handler); err != nil {
            log.Fatal("ListenAndServe:", err)
        }
    }
```

:)

Credits/Notes:

1. This guide is based on a guide for the clearnet and that is only 10% because it

was easier for me. 90% of the reason is to illustrate that *using I2P in your application is, in all likelihood, not that different than what you're already doing.** In all, this example is only 111 lines of code long minus comments.

- if the rest of your application only sends and recieves that which is non-linkable and absolutely necessary to its operation.