Concurrent and Distributed Systems

Project: GO Language Implementation of the TOR browser
Document Version: 1.1 - Updates on Output and Consistency
Submission Deadline: 17th April 2017

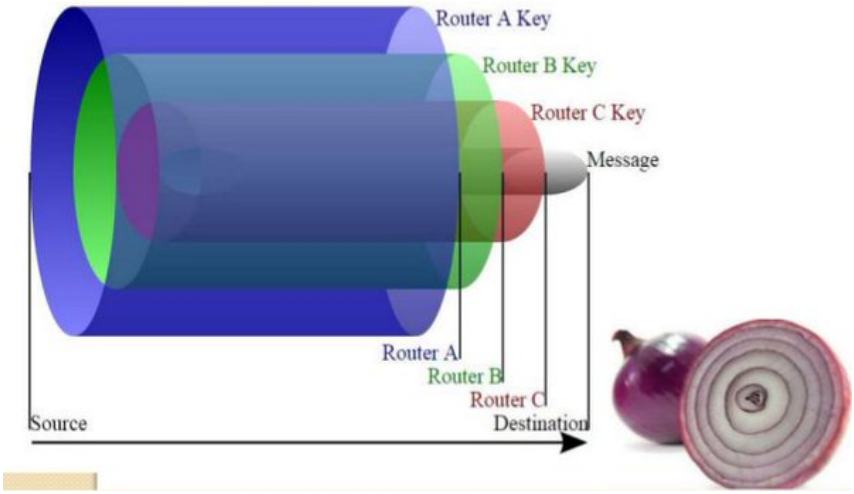
TOR Overview:

At a basic level, TOR works by using a set of intermediate nodes, called relays, to reach the destination. There are three basic kinds of relay nodes as mentioned below:

- **Entry Relay** – These nodes signify the entry point to the TOR network. They have an important role and some entry relays are already provided and are trusted ones. In addition, a relay can be selected to serve as an entry relay, if it is stable and has been live for some significant time duration.
- **Middle Relay** - Middle relays are nodes being used to transport traffic from the entry relay to the exit relay (see below). This provides some protection, as the entry and exit relays cannot identify each other.
- **Exit Relay** - These nodes are at the other edge of TOR network and signify the exit point. These relays send data to the eventual destination and the destination assumes the requests are coming from the Exit relays instead of the original client.

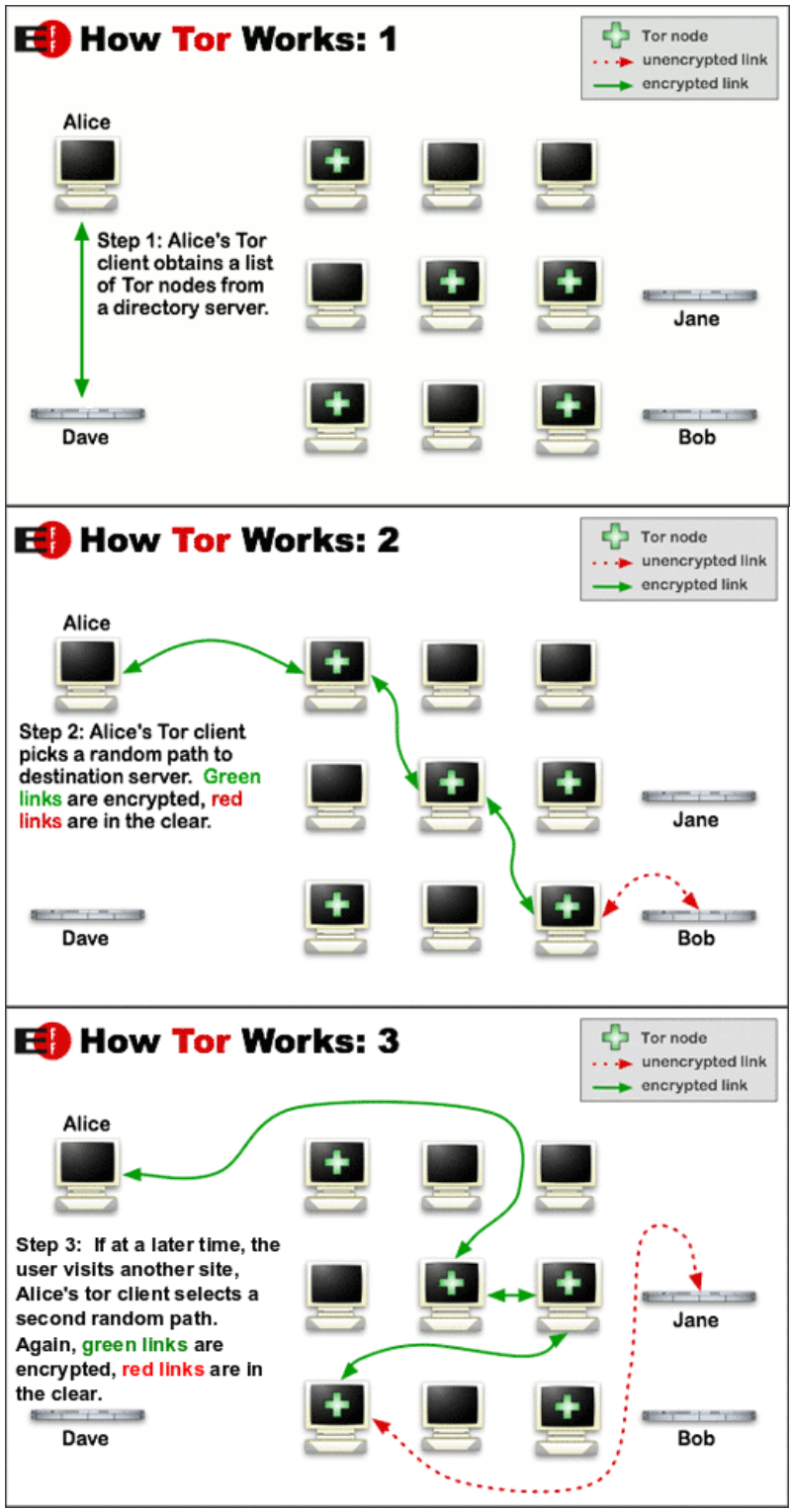
TOR extensively uses encryption to prevent any malicious relay nodes to learn anything, see Figure below[1]. This is done using the steps below:

- First, the client system encrypts the original contents so that none other than exit relay can decrypt the request.
- The resulting encrypted data is encrypted again so that none other than middle relay can decrypt it.
- Then the resulting encrypted data is encrypted once again so that none other than entry relay can decrypt it.



[1] https://pbs.twimg.com/media/CidHEw0WgAAtFsF.jpg

The following scenario[2] better explains the overall routing process.



[2] https://www.torproject.org/about/overview.html.en

Project Overview:

The GO based project you are required to implement is called *FASTOR,* its intended step by step behavior is as below:

- When a client (anyone willing to use FASTOR) launches the application, it asks the client that if it is willing to act as either middle or exit relay? The client can also refuse to participate as a relay but FASTOR would only work if sufficient number of Nodes exists.
- The application then starts a Webserver and indicates to the client that at what port, our FASTOR application is listening for connections.
- The client access the Webserver using http://ip:port/fastor/url-of-pagetovisit, for instance http://localhost:8080/fastor/www.google.com
- On receiving a Client request, the application contacts the directory server, which maintains a master list of all the relays. The client choses at random a path to destination i.e. entry, middle and exit relays.
- The application starts relaying the request to fetch the desired page that is www.google.com. Your request is then routed to the entry node, which forwards it to the middle node, which forwards it to the exit node, which eventually fetches the page and sends back on the same path.

Technical Details:

You have already created Web applications in GO, so the part to intercept and run appropriate code once client accesses http://localhost:8080/fastor/someurl is trivial. In order to fetch a Webpage, you can use http.Get, obviously from the http package, as given in the example below.

```
package main

import (
	"fmt"
	"io/ioutil"
	"log"
	"net/http"
)

func main() {
	res, err :=
http.Get("http://info.cern.ch/hypertext/WWW/TheProject.html")
	if err != nil {
		log.Fatal(err)
	}
	robots, err := ioutil.ReadAll(res.Body)
	res.Body.Close()
	if err != nil {
		log.Fatal(err)
	}
	fmt.Printf("%s", robots)
}
```

Similarly you are well aware of how to create distributed applications in GO. It is trivial to write servers and clients and for completion, server and client code is as below:

```
// server.go
package main

import (
	"fmt"
	"log"
	"net"
)

func main() {
	ln, err := net.Listen("tcp", ":6000")
	if err != nil {
		log.Fatal(err)
	}
	for {
		conn, err := ln.Accept()
		if err != nil {
			log.Println(err)
			continue
		}
		go handleConnection(conn)
	}
}

func handleConnection(c net.Conn) {

	fmt.Println("A client has connected", c.RemoteAddr())
	c.Write([]byte("Hello world"))

}


// client.go
package main

import (
	"fmt"
	"net"
)

func main() {
	conn, err := net.Dial("tcp", "localhost:6000")
	if err != nil {
		// handle error
	}
	fmt.Println("Connection successful!!", conn.RemoteAddr())
	recvdSlice := make([]byte, 11)
	conn.Read(recvdSlice)
	fmt.Println(string(recvdSlice))

}
```

Note: If you just write this back to client, you will most probably have broken links. The links that should be of the form actualwebsite/somelink would appear as localhost:8080/somelink. You thus need to parse the returned html, find the links and fix them. This function from the net/url package may be useful …

```
// ResolveReference resolves a URI reference to an absolute URI from
// an absolute base URI, …
func (u *URL) ResolveReference(ref *URL) *URL
```

It may be difficult to do so for all the links in all the Webpages so we have compiled a ***minimal*** list of URLs (and their sub links) that your application should display properly. They are listed below:

http://help.websiteos.com/websiteos/example_of_a_simple_html_page.htm

https://www.sheldonbrown.com/web_sample1.html http://www.meaningfultype.com/

http://info.cern.ch/hypertext/WWW/TheProject.html http://www.jackcallister.com/

http://www.muskfoundation.org/ https://jakebarry.us/ http://elbowsydney.com.au/

https://justinjackson.ca/words.html http://wadegarrett.com/

http://csb.stanford.edu/class/public/pages/sykes_webdesign/05_simple.html

Bonus (6 absolutes):

The project carries 15 absolutes with an additional 6 bonus absolutes (half of sessional marks!!) if your project meets ALL of the following criteria:

- The core functionality and requirements of project are completed, bonus can only be awarded if the project if complete first.
- Your application is able to handle and browse a number of Webpages, such as www.cnn.com, nu.edu.pk, bbc.com and others.
- You implement encryption features, as specified in TOR Overview section.

Some clarifications – v1.1:

- You need to keep the *masterlist,* at the directory server, up-to-date once an existing relay node disconnect. This should be done with a *heartbeat* mechanism where relay nodes periodically send *IamAlive* message to the directory server and if the directory server doesn't get this message for some time, it assumes the node is down and is removed from the master list.

- You need to have a flag in your code, such that if it is ON, it displays all the information being used and computed internally, on the terminal. This would help you better demonstrate your approach. Information to be displayed includes a number of things such as contents of master list retrieved by client, the path chosen, the steps being done and so on.