# Onion Soup: Anonymity Network Based on Tor

**Dalvir Khaira (s0f9), Ryan Wong (h4u9a), Sopida Chotwanwirach (h5j7), Tiffany Lam (n7f9) proj2_h4u9a_h5j7_n7f9_s0f9**

## Introduction

Onion routing is a protocol that attempts to provide anonymity to clients that browse within the network [1]. Our implementation will be a scaled down version of the Tor implementation of onion routing [2]. An anonymity network provides various challenges in the distributed context. We will further explore these challenges and their proposed high level solutions in the following sections.

## Background

Onion routing provides anonymity by encrypting data messages into fixed byte cells along each hop to relay nodes [3]. By encapsulating a layer of encryption on top of messages, cells remain the same size, meaning traffic analysis cannot determine how many hops a cell has taken [3]. Information is heavily obfuscated throughout the entire network; onion nodes will only know where the message came from and where to send it to. Exploiting this system of transfer, onion routers do not know if the previous node is the originator or just another relay node, maintaining anonymity throughout the network.

In order to set up an anonymity network, there must be a set of well trusted servers. These servers will contain list of IP and public key pairs (as well as various other pieces of information) which represents trusted onion nodes. A client must first connect to a server and choose a subset of IP-public key pairs to use as their network path, also known as a circuit. Once the subset of nodes have been finalized, a key exchange must be done to construct the circuit [1]. Once the circuit has been created, a client may freely browse using that circuit. In order to mitigate traffic analysis, circuits are refreshed periodically [1]. Additionally, if onion routers fail and disconnect, a teardown protocol is required to destroy all data in transit.

Once a circuit has been established, the encryption layered message can be propagated along the circuit, with each onion router decrypting its respective layer until the message has reached an exit node. An subsequent responses from the destination will travel along the same circuit but in reverse.

## Project Overview

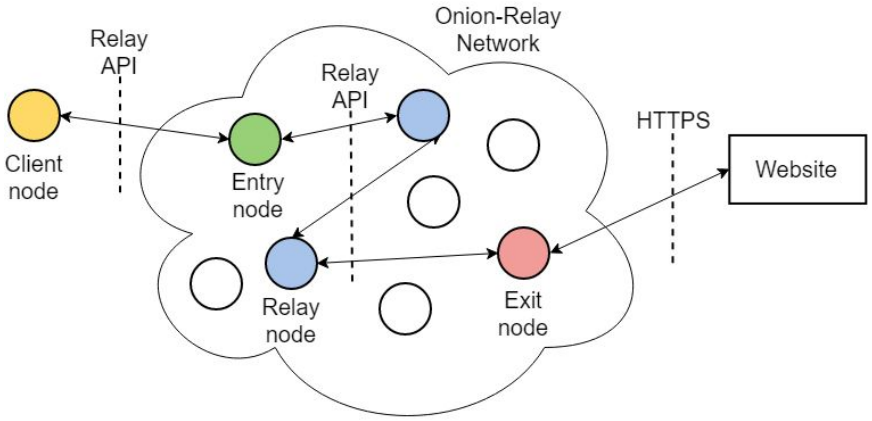
We propose to develop a Tor based network that will allow end users to browse websites anonymously. Our network will utilize two types of nodes: client nodes and onion nodes. Client nodes consist of an API and an application. The API provides an interface for the application to access the network. Onion nodes can be either an entry, relay or exit node. Entry, relay and exit nodes are considered indistinguishable from each other and will be named accordingly based on the context that the node plays in a particular circuit. For example, a given node may have both the responsibility of an entry node as well as a relay node at any given time depending on the circuit topology in the network. A client node connects to a directory server which stores a list of nodes which the client node can use to construct a circuit. Upon constructing a circuit, messages can be sent to and from a particular destination anonymously.

**Distributed Challenges and Scope:**
Our network will consider multiple connections ranging from client applications to the various nodes defined in our system. Our network will support multiple clients web browsing concurrently. The network must provide adequate performance while providing a client anonymity from others when browsing through the network. Onion nodes must accept multiple connections to act as part of a circuit. Every node must multiplex connections to the correct nodes. Because our network will support multiple clients, there will be concurrency challenges when multiple nodes form circuits simultaneously. We will also have replicated directory servers that stores IP addresses and public keys of participating nodes in our network to increase the availability of our system.

Our network will also consider various security threats (refer to the **Threat Model** section for more details). The security issues that have the highest priority in defending against include: eavesdropping, observing payloads along intermediate hops and malicious nodes that cause denial of service within the network. Taking these security threats into consideration, denial of service attacks on exit nodes or to the directory servers and end-to-end timing attacks will not be within this project's scope.

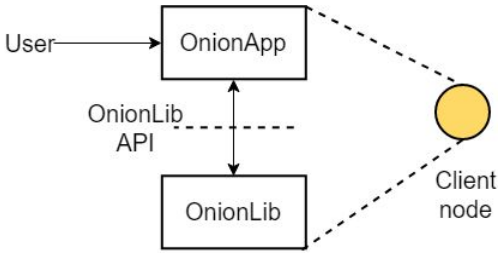
We expect that onion nodes and servers will fail and disconnections will occur. When a server fails, the subset of onion nodes it is responsible for attempt to connect to another server. Server failures are handled through a flooding protocol to other directory servers. When an onion node fails, the server that it is connected to is notified and floods this information to other directory servers. Other onion nodes and client nodes handle this failure using different procedures (refer to the **Failure of Onion Nodes in an Established Circuit** section for details on various node failure cases).

**System Components:**



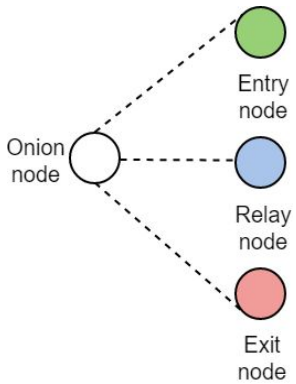

The onion network includes client node, onion nodes, directory servers, and an application. The topology of the network is a mesh because each onion node can be a part of multiple circuits.

Client Node



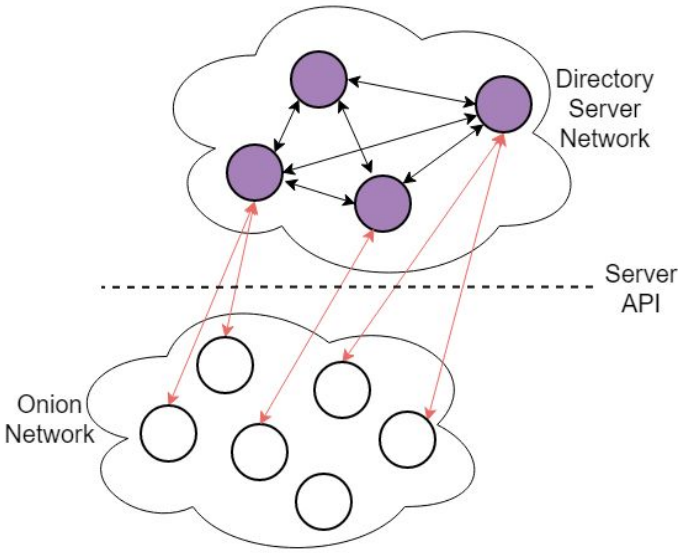

A client node consists of an OnionApp and OnionLib. A user interacts with an OnionApp which sends requests to the OnionLib. The OnionLib handles circuit and stream operations such as creation, teardown, and request-response propagation.

Onion Node



An onion node is either an entry, relay or exit node. Each node is registered with a directory server. Once the node is registered, heartbeat messages are sent periodically to the connected server. If the connected server goes down, the node will attempt to register with another directory server. All nodes in an established circuit will send heartbeat messages to their immediate neighbouring nodes. Similarly, all nodes in an established circuit will receive heartbeat messages from its immediate neighbours. If a node in the circuit fails, a proper truncation and possible extension of the circuit is required (see section **Failure of Onion Nodes in an Established Circuit**).

Directory Server




There are k number of servers, each server can be connected to 1..k-1 other directory servers. Each server keeps a replicated list of n trusted nodes (IP and public key pair), but each server only connects to a subset of n nodes. When an onion node $O_1$ is registered with a server $S_1$, server $S_1$ adds node $O_1$ to its list and uses heartbeat to observe the connectivity of node $O_1$. If node $O_1$ crashes and goes offline, node $O_1$ is then removed from server $S_1$ list of connected nodes. The server then floods the updated information to other servers $S_2$, ..., $S_k$. Note that servers $S_2$, ..., $S_k$ are not directly connected to node $O_1$, but rely on information from $S_1$. Ideally, all servers will contain the same list of registered onion nodes. However, the flooding of information is not guaranteed to be successful. The message may be lost or the server with the updated information is down before the flooding procedure is initiated. Hence, there is a possibility of inconsistency in the list of connected nodes that each server has. In this project, the consistency semantics are best effort.

Application

Application is a program that utilizes OnionLib. Please refer to the **Demo Plan** section for further details on the application.

# Project Details

### APIs

#### *OnionLib API: OnionApp - OnionLib*

ok, err <- JoinOnionNetwork(minNode, numRetry,  privKey)

- Join the onion network by giving a minimum number of nodes required to be in a circuit, and public and private keys to identify the client node. numRetry specifies the number of times the client application wants the OnionLib to rebuild the circuit on onion nodes failure. Return error. If error not set return a boolean whether the operation was successful.

response, err <- GetRequest(address, command)

- Make a request given a command, and address of the destination. Return error. If error not set return the response from the request.

ok, err <- UnlinkFromNetwork()

- Unlink from the onion network. Return error. If error not set return a boolean whether the operation was successful.

#### *Server API: Onion node - Server*

nodes, err <- GetNodes()

- Get a list of all connected nodes from the server. Return error. If error not set return a list of known connected onion nodes.

ok, err <- Register(nodeAddress)

- Register an onion node with a server given node's IP and port. Return error. If error not set return a boolean whether the operation was successful.

#### *Server API: Server - Server*

ok, err <- FloodDirectory(nodeDirectory)

- Flood a list of onion nodes and its connectivity status provided by nodeDirectory when the list get updated. Return error. If error not set return a boolean whether the operation was successful.

ok, err <- Register(serverAddress)

- Register a server with another server given IP and port. Return error. If error not set return a boolean whether the operation was successful.

#### *Relay API: Node - Node*

otherDHKey, sharedKey, err <- CreateCircuit(circuitID, myDHKey)

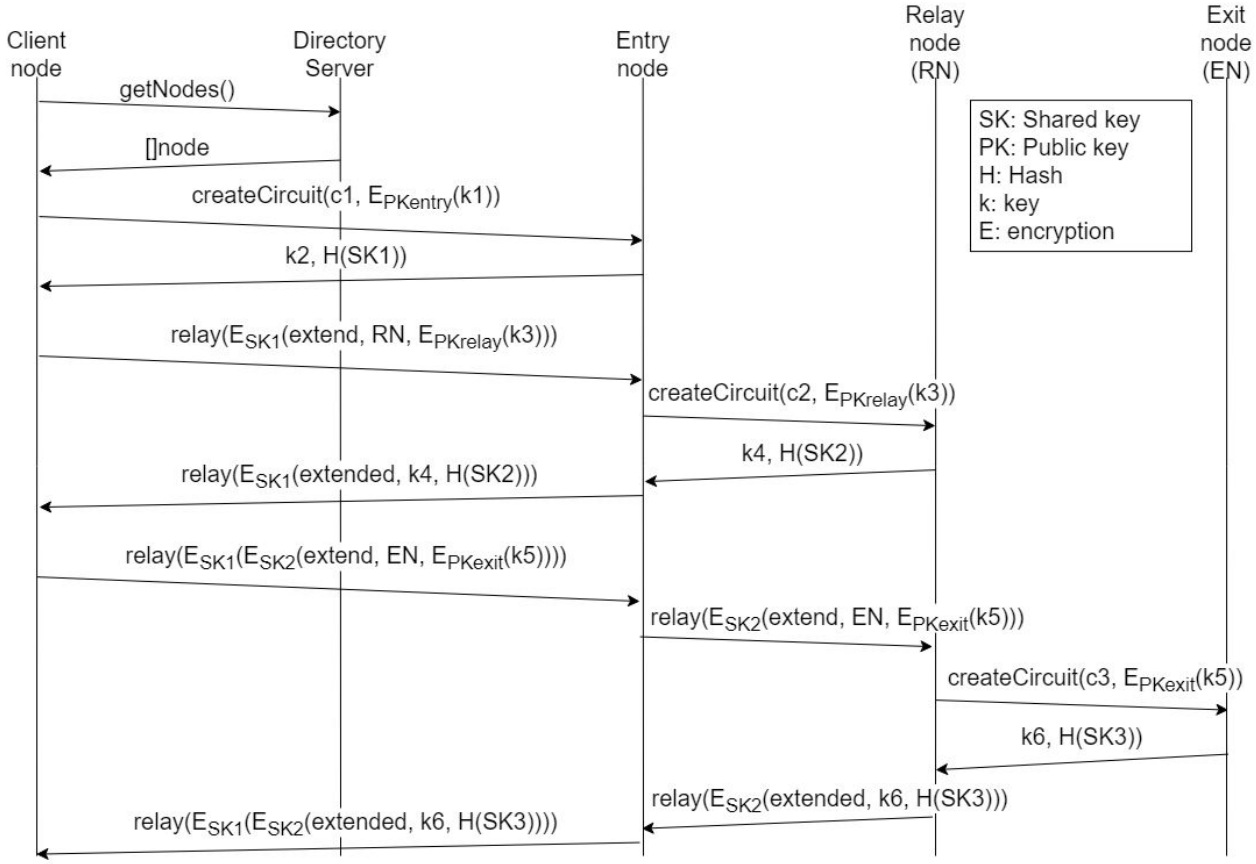
- Create a circuit with given circuit id and half of DH handshake. Return error. If error not set return the other half of DH handshake and hash shared key between the two nodes.

err <- Relay(encryptedCommand)

- Relay an encrypted command which includes

- extend: extends a circuit by one node by providing public key of the node to be extended to and half of DH handshake.
- extended: returns response from “extend” command with the other half of DH handshake and hash shared key between the two nodes.
- truncate: delete a node from a circuit.
- truncated: returns response from “truncate” command whether the truncation of the circuit was successful.
- begin: initialize a TCP stream given address of the destination.
- connected: returns response from “begin” command whether the connection to the destination was successful.
- data: sends request given a command and data to the destination.
- response: return response from “data” command from the destination.
- end: close a TCP stream given the address of the destination.
- disconnected: return response from “end” command whether closing of a TCP stream was successful.
- dropped: an onion node sends a message to OnionLib to inform that its successor node fails.

**Establishing a Circuit**




The application requests to join the onion network using OnionLib API. The OnionLib then contacts the server to obtain a list of connected nodes. A minimum number of nodes set by user

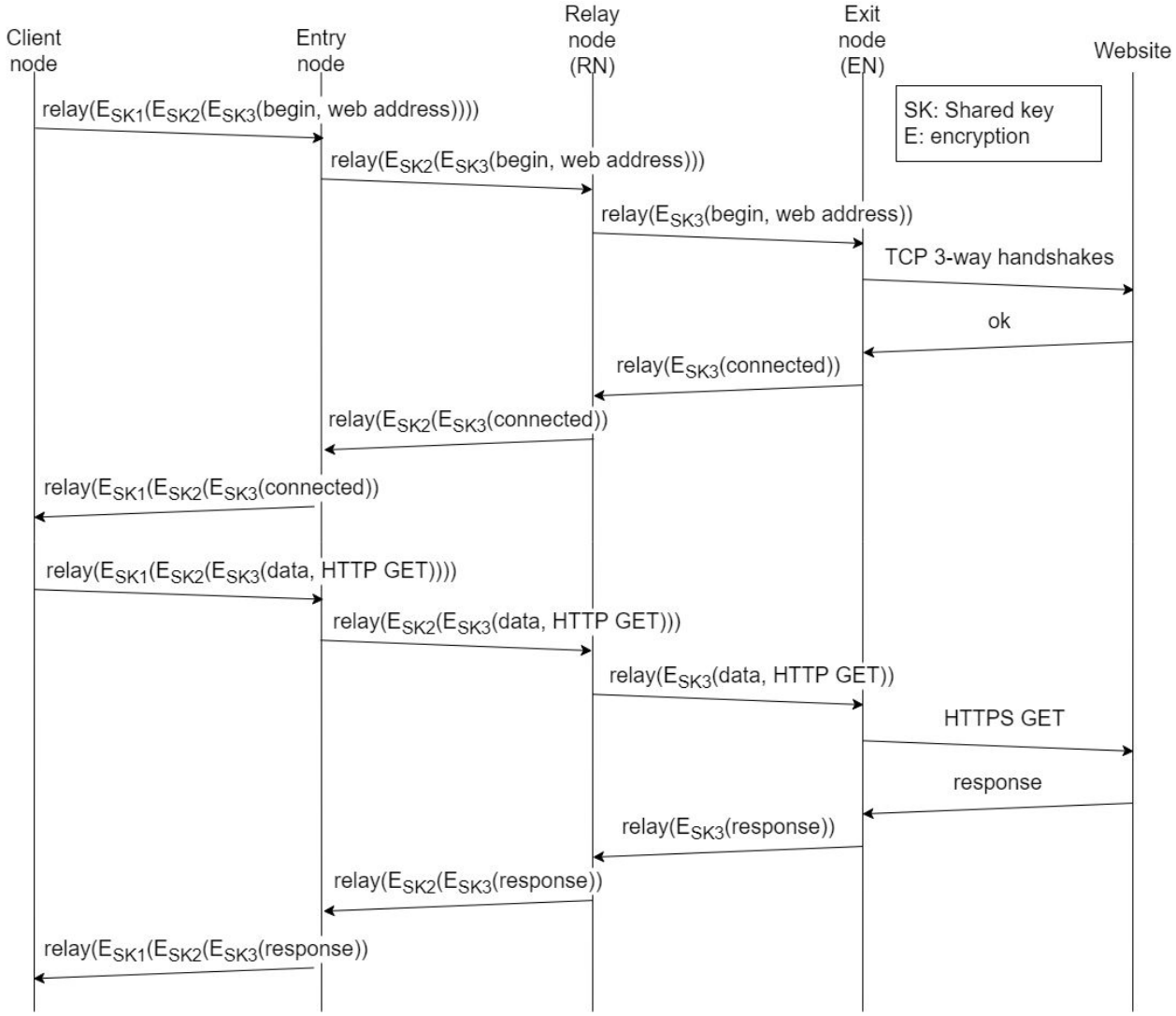
are randomly selected to build a circuit. Each node in the circuit is contacted to initiate key exchange. Many TCP streams can be created on a single circuit. The OnionLib starts a timer when a new circuit is built and periodically checks if there is a used circuit that has no open stream in 1 minute interval. If so, the circuit will be reconstructed using a new set of randomly chosen nodes. If the same circuit is used with open stream(s) for 10 minutes, the circuit will be forced to refresh. To provide a smooth transition to a new circuit, the new circuit will be built in the background before the old circuit is terminated. Circuit switching could potentially drop some TCP packets that are in flight. In this case, the OnionLib can resend the request on the new circuit. The application should not observe any transition from old to new circuit.

**Onion Message Encryption**

There are two sets of keys: identity and shared keys. The identity key is a long-term key that is used to identify each onion node which is generated by ECDSA. The shared key is a short-term key that is negotiated between client and onion nodes when constructing a circuit, and generated by Diffie-Hellman (DH) algorithm. To negotiate a shared key, the client node first creates half of DH handshake and encrypts it with the public key of the entry onion node using RSA encryption. The client node then sends a request to the entry node to create a new circuit with a circuit ID that hasn't been previously used between them, and the encrypted handshake. The entry node decrypts the half DH handshake from the client node with its own private key. After creating the other half of DH handshake, the entry node uses both halves of DH handshake to compute a shared key. The entry node replies to the client node with its half of DH handshake and hashed shared key. Now, the client node can compute the shared key and compares its hashed key with the hashed shared key from entry node.

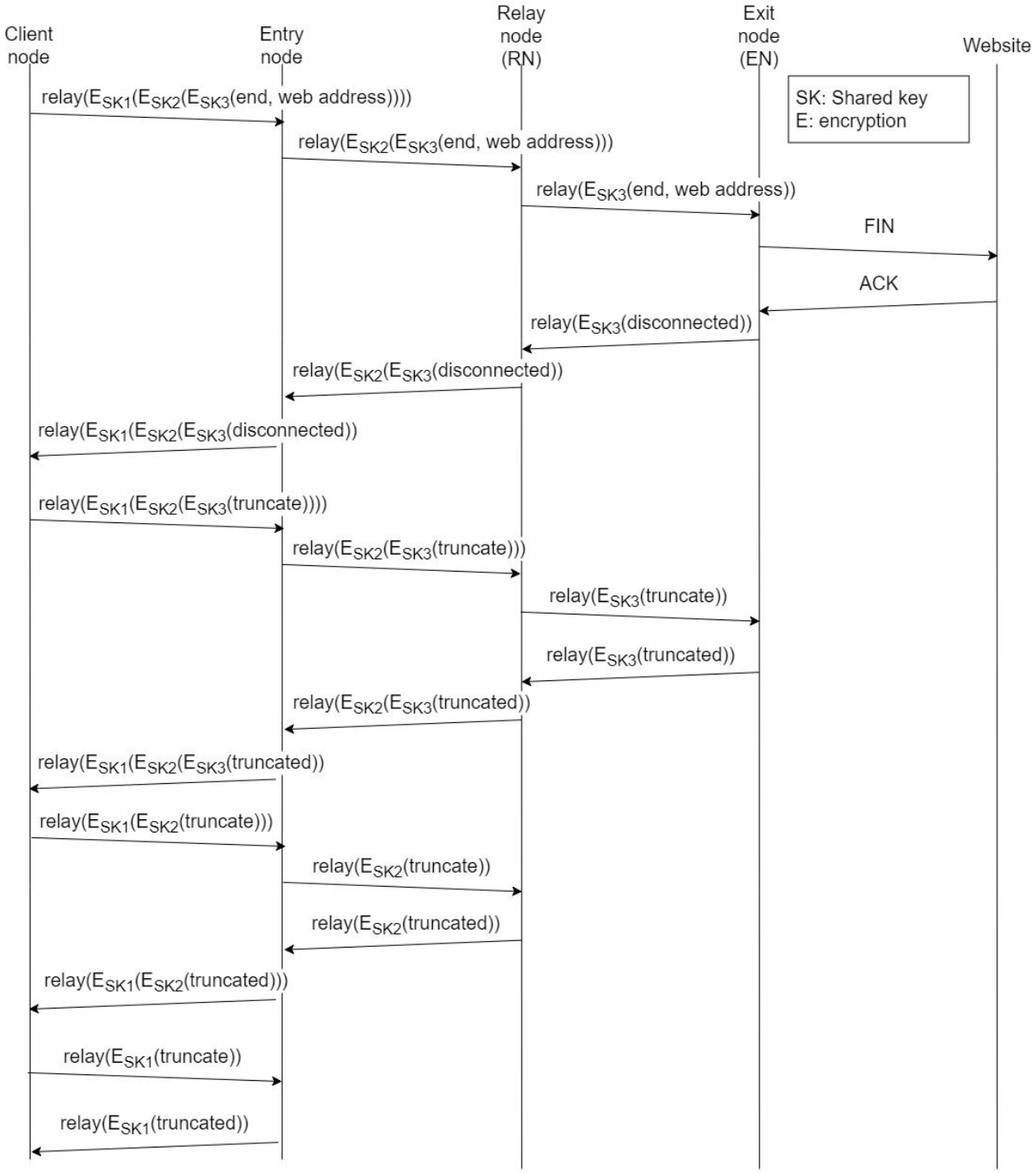
To extend the circuit to node k, the client encrypts an "extend" command, the address of node k, and a new half DH handshake that is encrypted with public key of node k with the shared keys with node k-1, k-2, …, entry node. The client node then relays the encrypted message to entry node. The layered encryptions prevent nodes from reading messages that are not meant for them. The entry node decrypts the message with the shared key and relays the message until the node k-1 in the circuit is reached. Node k-1 then request to negotiated a shared key in the same manner described above.

**Sending Messages via the Circuit**



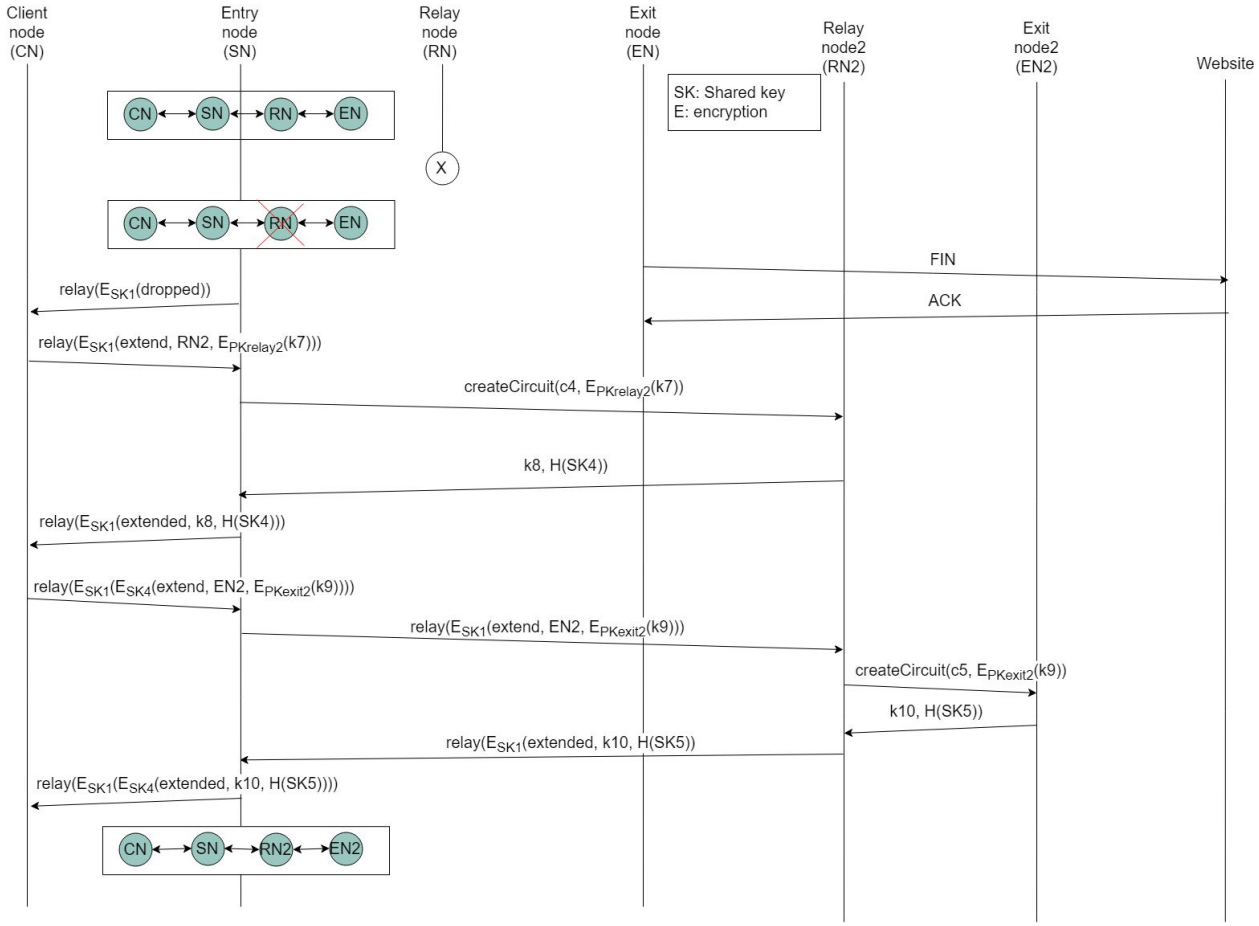

Once the circuit is constructed, multiple TCP streams can be established. If a circuit has k nodes, the client node encrypts a “begin” command, and address of the destination with shared keys with node k, k-1, …, entry node using AES encryption. The entry node decrypts the message with the shared key and further relays the message to the next node until the message reaches node k. Node k then performs TCP 3-way handshake with the destination website and returns “connected” message if the connection is established successfully. The connected message is encrypted with the shared key between the client and node k, and relayed to node k-1. Node k-1 encrypts the message with its shared key and relays the message to node k-2. The same process is repeated until the client node is reached. The client node can decrypt the message with shared keys with nodes 1, 2, …, k to read the response. To make a HTTPS request, The client node uses the same process described above with command “data” and HTTPS request to relay the message to the website. The response can be read by “peeling the onion” (decrypting the message with shared keys).

**Termination of a Circuit**




An application can initiate termination of a circuit at any given time by making a call to OnionLib. The receiving end of the circuit or destination will not have a control over when the circuit is destroyed. To terminate a circuit, any opened stream(s) will first be closed with “end” command. Then, “truncate” command is relayed to each node in the circuit to shorten one node at a time starting from the exit node. The node will delete the shared key and drop connection to its neighboring nodes for that particular circuit.

**Failure of Onion Nodes in an Established Circuit**




***Relay Node Disconnection:*** When a relay node $O_k$ is disconnected, its immediate predecessor $O_{k-1}$ will detect the disconnected node due to missing heartbeat. A “dropped” relay command will be propagated to OnionLib. This message notifies the client node that the circuit is shortened. If the number of nodes falls below the minimum nodes required in a circuit, OnionLib will call “extend” command to extend the circuit until the minimum number of nodes is reached. If OnionLib is unable to meet the minimum number of nodes, it will return error to the application. The immediate successor $O_{k+1}$ of the disconnected node will no longer receive heartbeat messages from the disconnected node. Note that node $O_{k+1}$ can no longer communicate with OnionLib. The successor will delete the shared key and drops out of the circuit. Node $O_{k+2}$ will then detect that node $O_{k+1}$ fails and follow the same termination procedure mentioned for node $O_{k+!}$. This process repeats until the exit node is reached. The exit node will first close the existing TCP stream(s) before deleting the shared key and dropping out of the circuit.

***Entry Node Disconnection:*** When an entry node is disconnected, OnionLib will only know that it is disconnected when it attempts to send a request to the entry node. The first relay node will detect that the entry node is disconnected when it stops receiving heartbeat messages from the entry node. The successive nodes after the failed node follows the same termination procedure described above. The OnionLib will have to create a new circuit that satisfies the minimum

nodes requirement. If reconstructing a new circuit is unsuccessful, an error will be returned to the application.

***Exit Node Disconnection:*** When an exit node is disconnected, The last relay node will detect that the exit node is disconnected when heartbeat fails. The last relay node will send “dropped” command back to OnionLib notifying the occurrence of node failure. The OnionLib might have to reselect an exit node if number of nodes in the circuit is lower than the minimum nodes needed.

**What the servers observe when an onion node fails:**
The server that an onion node is connected to will notice the disconnection through heartbeat failure. This results in connectivity status update of onion node list and flooding of updated information to neighboring servers. The flooding protocol will avoid cyclic paths and ensure that the flooding will terminate.

**What the client application observes when an onion node fails:**
Initially the client application will not observe onion node disconnections. The OnionLib will try to rebuild the circuit as many times as the client application explicitly stated through numRetry in JoinOnionNetwork method. If it can rebuild a circuit with the minimum number of nodes required and numRetry as stated by the client application, then a circuit is successfully rebuilt. Otherwise OnionLib will return an error to the client application stating that there is not enough onion nodes online to satisfy its minimum nodes requirement to rebuild the circuit.

**Scenarios Where Client Application Receives an Error:**

- No servers can be reached
    - Client node cannot obtain a list of available Onion nodes from the server
- Some online servers contain a stale list of Onion nodes that are either all disconnected or number of connected node is less than the minimum node requirement
    - Client node cannot build a circuit with unavailable or too few nodes
- No nodes are online in the Onion network
    - Client node cannot build a circuit with unavailable nodes
- One or more Onion nodes in the existing circuit fails and the number of available Onion nodes is lower than the client’s required minimum

**Server Failures:**
When a server fails, neighboring servers detect this through heartbeat failures and flood this information to other servers. The subset of nodes that the failed server was connected to will detect this failure through missed heartbeats. These nodes will attempt to connect to another directory server.

**Threat Model:**
Within our network, there could be aggressive attacks involving active participation in the network or interference, and passive attacks through traffic analysis. It is assumed that only trusted server(s) are used for directory information in the network. Because our network trusts

any new onion nodes entering the system, malicious onion nodes can be injected into the system. Malicious onion nodes may attempt to observe the payload along intermediate hops in their respective circuits. These malicious onion nodes can refuse to send requests within circuits they are participating in. They can also attempt to corrupt requests and interfere with circulation of data through TCP streams. These nodes may also attempt to deny service to other onion nodes by sending any number of duplicate valid requests or bad requests to its available neighbors. A potential passive attack is attempting to perform traffic analysis on the network to determine the circuit patterns which may result in revealing the identity of a user. Our network will defend against all of the attacks mentioned above.

Our network is vulnerable to denial of service attacks on entry and exit onion nodes within the network by external entities such as ISP providers. Since the directory information is publicly available, it is easy to block onion nodes. This halts any clients from using the network and the services it provides. We assume that a denial of service attack from external entities will not target our nodes. Another example would be timing attacks between entry onion node(s) and exit onion node(s) that analyze the time taken to execute a request through the circuits in the network. This can lead to a malicious entity determining circuit paths being used. We assume the system is safe under end-to-end timing attacks. Attackers do not have access to the traffic information coming out of the source and arriving at the destination. Finally, any threat will not have access to our complete system, meaning as administrators we can acquire logs of the full system to ensure that the system behaves as intended.

**Test Plan:**

We plan to test our system on Azure right from the beginning. During initial stages of development, we will test parts of the system individually with unit tests. During these initial stages, localhost testing might be used for the ease of convenience. First, we will test a single circuit and stream constructions with one application. Then, we will test circuit switching: i) forced switch and ii) periodic switch when there is no open stream in a circuit. Later, we will test our system with multiple circuits and streams that are running sequentially and concurrently. Once our system takes shape, we will test the system with the demo setup on Azure extensively. Moreover, we will also test the disconnection of onion nodes by killing off any running onion nodes that are participating in circuits. Also, we will inject malicious node(s) that will attempt to disrupt the network, and test security of our system. Finally, some performance testing will be done through timing of the system when fulfilling single and multiple requests with one or more concurrent circuits.

**Demo Plan:**

1. Servers startup.
2. N trusted nodes are started and ready to receive connections.
3. pcap is started up on VMs where nodes, applications and server is running respectively.
    a. Each node runs on a different VM.
    b. Each application runs on a different VM.
    c. The server runs on a different VM.

4. To demonstrate that web pages can be retrieved through our onion network, client A will make a HTTPS GET request to retrieve plain HTML of a webpage.
5. To show that our onion network can handle multiple clients concurrently, client A, B, C and D will send HTTPS GET requests to retrieve HTML of 4 different webpages at the same time.
6. To demonstrate that our onion network indeed maintains anonymity of the clients that sends out HTTPS GET requests, we will use pcap to observe traffic at each of the nodes and applications, such that we can see packets sent to the the onion network nodes.
7. To demonstrate that server failures are handled by our network, we will terminate server processes and show that requests are still being fulfilled by our network
8. To demonstrate that onion node failures are handled by our network, we will terminate onion node processes and show successful requests when user's minimum node requirement and circuit retry number are met.
9. To demonstrate that onion node failures are handled by our network, we will terminate onion node processes and show requests where clients minimum number of nodes and circuit retry numbers are not satisfied. This should result in appropriate error being returned to the application.

**Alternative Demo Plan:**

1. Build a proxy server that uses OnionLib . This proxy server will be accepting connections via HTTPS.
2. Next use Chrome to connect to the proxy server with a proxy configuration file [4]. We can now browse the internet with Chrome via our Onion network.
3. The rest of the steps are similar to those in the original demo plan. The only difference is that we use Chrome with the proxy server to browse the Internet instead of retrieving plain HTML.

**Timeline:**

| **Deadline** | **Task(s)** |
|---|---|
| March 9 | Project proposal |
| March 12 | Implementation goals for this week:<br>Start stubs and refine the following APIs:<br>OnionApp <-> OnionLib **(Team)**<br>Onion node <-> Server **(Team)**<br>Server <-> Server **(Team)**<br>Node <-> Node **(Team)** |
| March 15 | - Implement server, such that it can accept registration from new onion nodes. It should also be able to a list of connected onion nodes when requested **(Ryan)**<br>- Implement server such that the server can register and flood other servers **(Ryan)** |

| | |
|---|---|
| | - Implement onion node, such that it can establish heartbeat with other onion nodes and can detect when a predecessor or successor node fails. Onion nodes can form a path such that failures can propagate to successors. **(Dalvir)**<br>- Implement OnionLib, such that it can initiate a connection with an onion node to create circuit (the entire process of create circuit need not be completed) **(Tiffany)**<br>- Start working on encryption library to provide encryption/decryption functionalities **(Sopida)** |
| March 19 | Implementation goals for this week:<br>- Onion nodes can connect to the server **(Ryan)**<br>- Circuit refresh **(Tiffany)**<br>- Client node can establish a circuit with at least one relay node. **(Tiffany)**<br>- Client node can send a HTTPS GET request through a circuit with at least one relay node and receive a response back **(Ryan)**<br>- Client node can terminate a circuit **(Dalvir)**<br>- Client node can establish a circuit with more than one relay nodes **(Dalvir)**<br>- More than one client node can create a circuit in onion network (their circuit do not overlap) **(Sopida)**<br>- When one circuit is terminated, the other circuit that shares an onion node should not be interrupted and does not observe the termination of the other circuit **(Sopida)** |
| March 23 | Schedule a meeting with our team's assigned TA to discuss project status |
| March 26 | Implementation goals for this week:<br>- Two client nodes can create a circuit in onion network with multiple common nodes between two circuits **(Tiffany)**<br>- Two clients with multiple common nodes, if one client node terminate the circuit, the other client node with a circuit still should not observe any changes **(Sopida)**<br>- System should be able to handle failure of onion nodes **(Dalvir)** |
| April 2 | - Complete any tasks leftover from the previous week **(Team)**<br>- Continue debugging the system **(Team)** |

| | |
|---|---|
| | - Start working on the final report **(Team)** |
| April 6 | Project code and final report |
| April 7 - Demo date (TBD) | Demo preparation<br><br>If time permits, we will implement a proxy server that uses OnionLib. We will demo with Chrome connected to our proxy server. |
| TBD | Demo Day! |

**SWOT Analysis:**

| | |
|---|---|
| **Strengths:**<br>- Team members have worked together in previous projects and have established a working workflow<br>- Team members have a genuine interest in the security space which this project facilitates in a distributed context<br>- Team has a good track record for being able to meet frequently throughout the week<br>- Team members are comfortable in developing in Go | **Weaknesses:**<br>- The project is based on an existing system, there is less room for implementing new creative solutions.<br>- The team has no prior experience or knowledge of Tor network<br>- It is uncertain how much of Tor specification that we can fully implement without diving into coding<br>- Project scope might have too many moving parts (different APIs and failure cases) which will take significant effort to harmonize |
| **Opportunities:**<br>- Our onion routing system is based on Tor. The success rate of this project is high, since we can look for solutions in existing implementations.<br>- There are existing security threats that are not solved by Tor; for instance, the timing attack. We can potentially look for creative solutions to prevent these attacks.<br>- Tor provides a clearly defined specification for implementations<br>- We can use pcap to observe the TCP packets on Azure to ensure anonymity | **Threats:**<br>- Various group members have other project deliverables from other courses closer towards the end of the term<br>- We cannot access Tor documentation on campus (From our experience, UBC networks block any Tor related information)<br>- There are limited official libraries provided for cryptography (DH key exchange) |

**References:**

[1] Paul F. Syverson, David M. Goldschlag, Michael G. Reed. 1998. Anonymous Connections and Onion Routing. IEEE Journal on Selected Areas in Communications Volume 16, Issue 4. http://ieeexplore.ieee.org/document/668972
[2] https://en.wikipedia.org/wiki/Onion_routing
[3] https://svn.torproject.org/svn/projects/design-paper/tor-design.pdf
[4] https://developer.chrome.com/extensions/proxy