Onion Soup: Anonymity Network Based on Tor

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

1 Introduction

1.1 Background

Onion routing provides an anonymous browsing experience by hiding a user's identity from its desired destination. Anonymity is achieved by encrypting data messages into fixed byte cells along each hop to relay nodes [1]. By encapsulating a layer of encryption on top of messages, cells remain the same size, meaning traffic analysis cannot be used to determine how many hops a cell has taken [1]. Information is heavily obfuscated throughout the entire network; onion nodes will only know where the message came from and where to send it to. Utilizing this system of transfer, onion routers do not know if the previous node is the originator or just another relay node, thus maintains 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 user must first connects 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 selected, a key exchange is performed to construct the circuit [2]. Once the circuit has been created, a client may freely browse using that circuit. In order to mitigate traffic analysis, circuits are refreshed periodically [2]. Additionally, if onion routers fail and disconnect, a teardown protocol is required to destroy all data in transit.

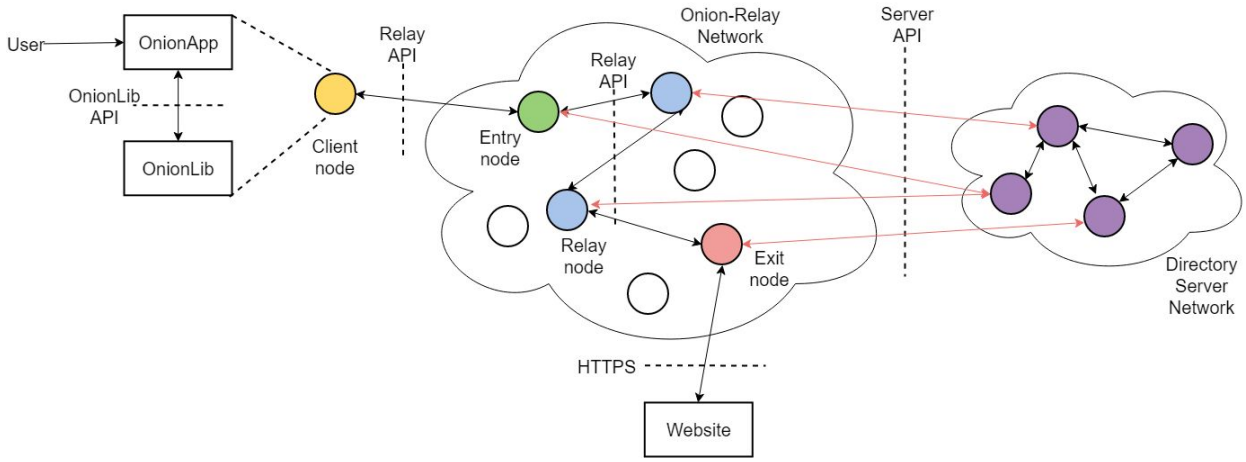
1.2 Overview

Our anonymity network was implemented as a scaled down version of the Tor implementation. The intent was to provide similar fault tolerance to the network as well as anonymity to users. Many design decisions were based heavily on the Tor specifications for onion routing [1]. Our network utilizes three types of nodes: client, onion and server 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 either be an entry, relay or exit node. These nodes are considered indistinguishable from one other and will be named based on each node's role in a particular circuit. For example, a given onion node can be both entry and relay node in different circuits depending on the circuit topology in the network. A client node connects to a directory server which provides a list of onion nodes that are available to participate in a circuit. Upon constructing a circuit, messages can be sent to and from a particular destination anonymously.

The system is implemented to handle concurrent users creating circuits and sending requests throughout the network while simultaneously handling node churn. The system is intended to work with an arbitrary amount of circuit topologies, potentially causing users to share onion nodes in their respective circuits. Node failures are handled gracefully and disrupted requests are re-sent for a number of retries.

2 Design

2.1 System Components



Client Node: A client node consists of an OnionApp and an OnionLib. A user interacts with an OnionApp which calls API methods defined in the OnionLib. The OnionLib handles circuit and stream operations such as creation, destruction, and request-response propagation.

Onion Node: There are three types of onion nodes: entry, relay and exit nodes. The entry node is the first onion node in the circuit that is connected to the client node. The exit node is the last node in the circuit which is connected to the destination address. The relay nodes are responsible for relaying messages from the entry node to the exit node and vice versa. Each onion node registers with one of the available directory servers in the system. Once the node is registered, uni-directional heartbeat messages are sent periodically to the connected server. Similarly, bi-directional heartbeat messages are sent and received among the neighboring nodes in an established circuit.

Directory Server: There are k directory servers in the system. These servers can connect to 1 to k-1 other directory servers. A subset of onion nodes are connected to each server. Each server stores a list of all connected onion nodes in the system. When a new onion node is registered with a server, the server will update its list of connected onion nodes and flood this information to its neighboring servers. Ideally, all servers should 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 inconsistent connected-node list across directory server network. In this project, the consistency semantics are best effort.

2.2 Relay Commands

Relay commands are used for relaying messages in a built circuit.

- extend: extends a circuit by one node by providing address of the target onion node to be extended to and half of DH handshake. Receiver of the “extend” command should make a ReceiveCreateRequest RPC call to the target onion node.
- extended: returns response of “extend” command with the other half of DH handshake and hash shared key between the two nodes.

- truncate: removes a target onion node from a circuit. Receiver of the “truncate” command should make a SendDestroyRequest RPC call to the target onion node.
- truncated: returns response from “truncate” command whether the truncation of the circuit was successful. Alternatively, this command is propagated to the client node when an onion node in the circuit detects that its immediate successor has disconnected.
- begin: initializes a TCP stream at the exit node given the web address of the destination.
- connected: returns response from “begin” command indicating the connection to the destination was successful.
- data: sends request given a command (GET request) and data to the destination.
- response: returns response of “data” command.
- teardown: returns erroneous response of “begin” or “data” command, indicating that connection to the destination was broken.
- end: closes TCP stream(s) at the exit node that are opened on given circuit id

2.3 Onion Packets and Cells

A relay cell with a fixed size of 512 bytes is used to represent a relay message. An onion packet is a relay cell that has been encrypted with multiple encryption keys (refer to *section 2.4 Onion Message Encryption* for more detail on message encryption). Each relay cell consists of five fields: a relay command, streamID, digest, payload length and payload.

Relay Command (1 byte)	StreamID (2 bytes)	Digest (16 bytes)	Actual Payload Length (2 bytes)	Payload (491 bytes)

Relay command is one of the commands listed in section 2.2. The streamID is an unique identifier for a TCP stream in a circuit. The digest is a MD5 hash of the 491 bytes payload. The payload is the actual data portion of the message. Note that if the data size is less than 491 bytes, zeros are padded to the front of the data to form a 491 bytes payload. The fix sized cell allows the user to maintain anonymity during transit. The payload length indicates the actual length of the data without any zero paddings. The actual payload length field is crucial for the message receiver to properly decode the relay cell.

The content of the payload differs depending on the relay command. The “extend””, “truncate” and “truncated” commands contain a target node address in the payload; 2 bytes IP address and port. The “begin” command contains web address of the destination in the payload. The “data” message contains the GET request in the payload. Lastly, a “response” message payload consists of three fields, 2 bytes packet sequence number, 2 bytes total packet number and 487 bytes response data from the destination. The responses are broken down into chunks of 487 bytes. Packet sequence number guarantees that the client node can rearrange the packet in the correct order in case they arrive out of order.

2.4 Onion Message Encryption

There are two sets of keys: identity and shared keys utilized in the onion network. The identity key generated by RSA is a long-term key that is used to identify each onion node. The shared key is a short-term key that is negotiated between the client and each onion node during construction of a circuit.

Key Negotiation: The Diffie-Hellman (DH) algorithm is used to negotiate a shared key. The client node first creates half of DH handshake and encrypts it with the public key of the target onion node using RSA encryption. The client node then sends the encrypted handshake to the target node. The target node decrypts the half DH handshake from the client node with its own private key. Similarly, the target node

generates its own half of DH handshake. With two halves of DH handshake, the target node can compute a shared key. The size of shared key is 128 bytes and must be reduced to 16 bytes to adhere with AES-128 key size standard. Therefore, anytime shared key is mentioned in this report, it is the MD5 hash of the 128 bytes shared key. The target 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 shared key with the hashed shared key from target node.

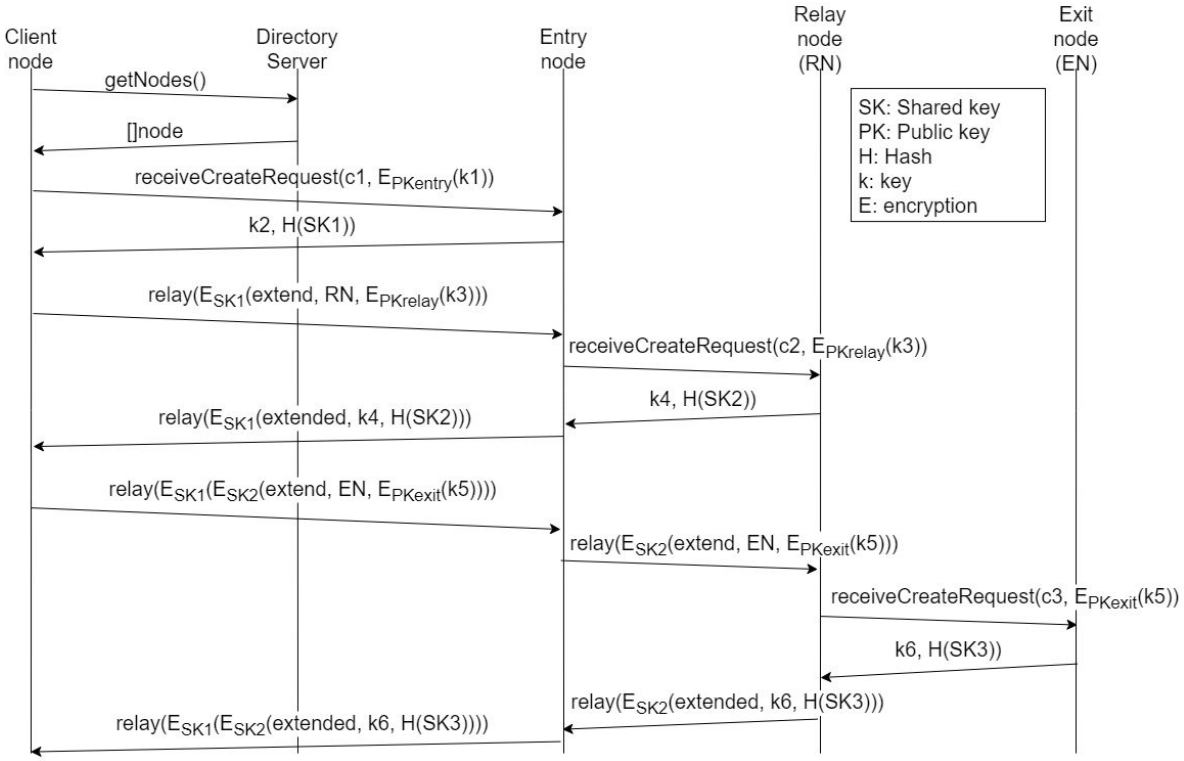
Packet Encryption: Before the client node sends a request, it encrypts the packet with the shared keys exchanged with all the onion nodes in the circuit. AES with CTR mode is selected for packet encryption. The layered encryption starts with the shared key of the target node and ends with the shared key of entry node, working outward:

$$E_{on,1}(E_{on,2}(E_{on,3} \dots (E_{on,target}(message))))$$

where *on* refers to onion node. When a packet is being relayed forward through the circuit, the onion nodes use their shared key to decrypt the message. Each relay "peels" a layer of the onion. The layered encryption prevents onion nodes from reading messages during the relay process if they are not the target node. For an onion node to relay messages backward to the OnionLib, it encrypts the message with its shared key and sends the encrypted message to its parent. Similarly, the parent node encrypts the received message with its shared key and relays the message to its predecessor. The OnionLib can read the response by "peeling the onion"; decrypting the entire message starting from the shared key with the entry node. The OnionLib checks the digest against the payload after every decryption to validate the message. Once the original message is reached, the decryption process stops.

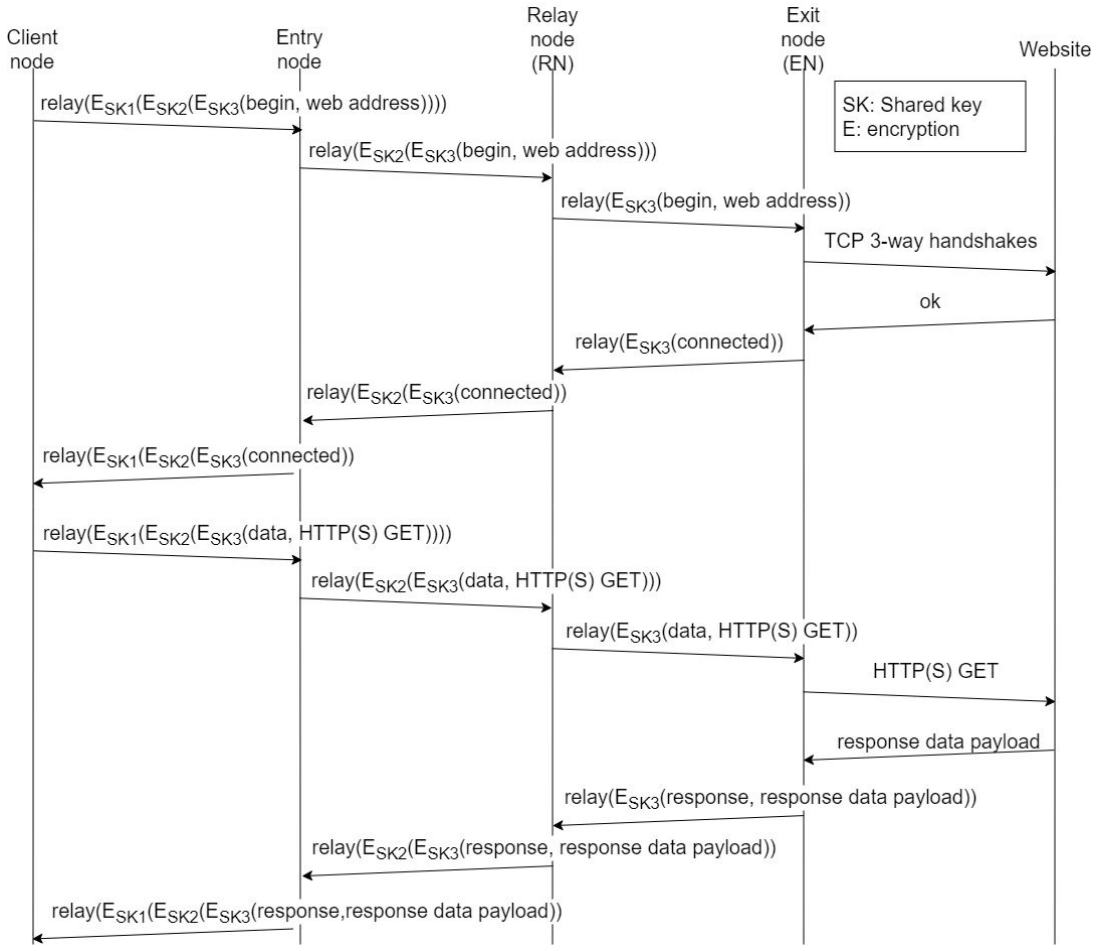
3 Implementation Approach

3.1 Circuit Construction



The client application can request to join the onion network through the OnionLib API. The OnionLib first obtains a list of connected onion nodes from the server. A minimum number of nodes set by user are randomly selected to build a circuit. Each chosen node is contacted to initiate key exchange. The OnionLib sends a ReceiveCreateRequest to the entry node with a circuit UUID (universally unique identifier) and its encrypted DH handshake. Once key negotiation with the entry node is completed, the OnionLib creates a relay cell with “extend” command, address of the target node and its encrypted DH handshake. The packet is then encrypted with the shared key and sent to the entry node. When the entry node receives a packet, it decrypts the packet with its shared key for that circuit and sends ReceiveCreateRequest to the target node. The target node computes a shared key, and sends its DH handshake and hashed shared key back to the entry node. The entry node creates a relay cell with “extended” command together with the information from the target node. The entire packet is encrypted with the shared key and relayed back to the OnionLib. The OnionLib now has two shared keys: one with entry node and one with the first relay node. This process is repeated until the circuit is constructed with all the chosen nodes.

3.2 Stream Construction And Data Request

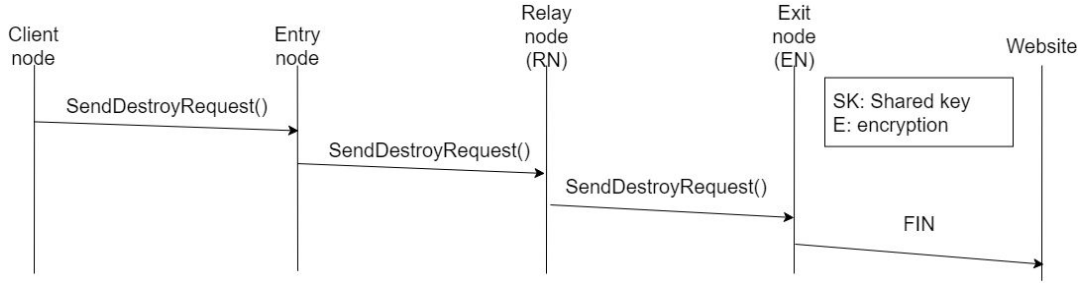


Multiple TCP streams can be established on a single circuit. The OnionLib creates a relay cell with “begin” command and address of the destination. The packet is encrypted with shared keys of all the onion nodes in the circuit and relayed to the exit node. Upon receiving the “begin” command, the exit node decrypts the packet and performs TCP 3-way handshake with the destination. The exit node returns a “connected” message if the connection is established successfully. If there is an error in TCP handshake, a “teardown”

message is propagated back to the OnionLib. The OnionLib will retry the “begin” command specified by number of times as set by the user.

To make a HTTP(S) GET request, the client node uses the same process described above with command “data” and HTTP(S) GET request to relay the message to the destination website. The exit node returns a “response” command together with the data if the GET request was successful; otherwise, a “teardown” message is returned. The OnionLib handles the “teardown” command in the same fashion described above.

3.3 Circuit Termination



An application can destroy a circuit at any time by making a call through OnionLib API. To terminate a circuit, the OnionLib calls SendDestroyRequest on the entry node. The entry node will relay the SendDestroyRequest request to its immediate child. The entry node then removes the shared key that was exchanged with the client node for that particular circuit. The connection to the client node is also removed. Similarly, a child node that received a destroy request will remove the shared key with the client node. If the child node is not the exit node, it relays the destroy request to its immediate successor. If the child node is the exit node, it will remove the shared key and close any open TCP streams. By removing the shared key, onion and client nodes can no longer participate in the current circuit. Hence, the circuit is terminated.

3.4 Circuit Force Refresh

If the current circuit has been built for longer than a refresh threshold, the circuit will be forced to refresh. During circuit refresh, the current circuit is first destroyed. Then, an updated list of currently available onion nodes is fetched from the server before building a new circuit. If the server that the onion node was initially registered with cannot be reached, the onion node will attempt to connect to another available server. If the number of available onion nodes falls below the required minimum nodes, an error will be returned to the client application. 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 timer is based off a specified constant variable and scales linearly with the size of the circuit. The application should not observe any transition from the old to the new circuit.

4 Discussion

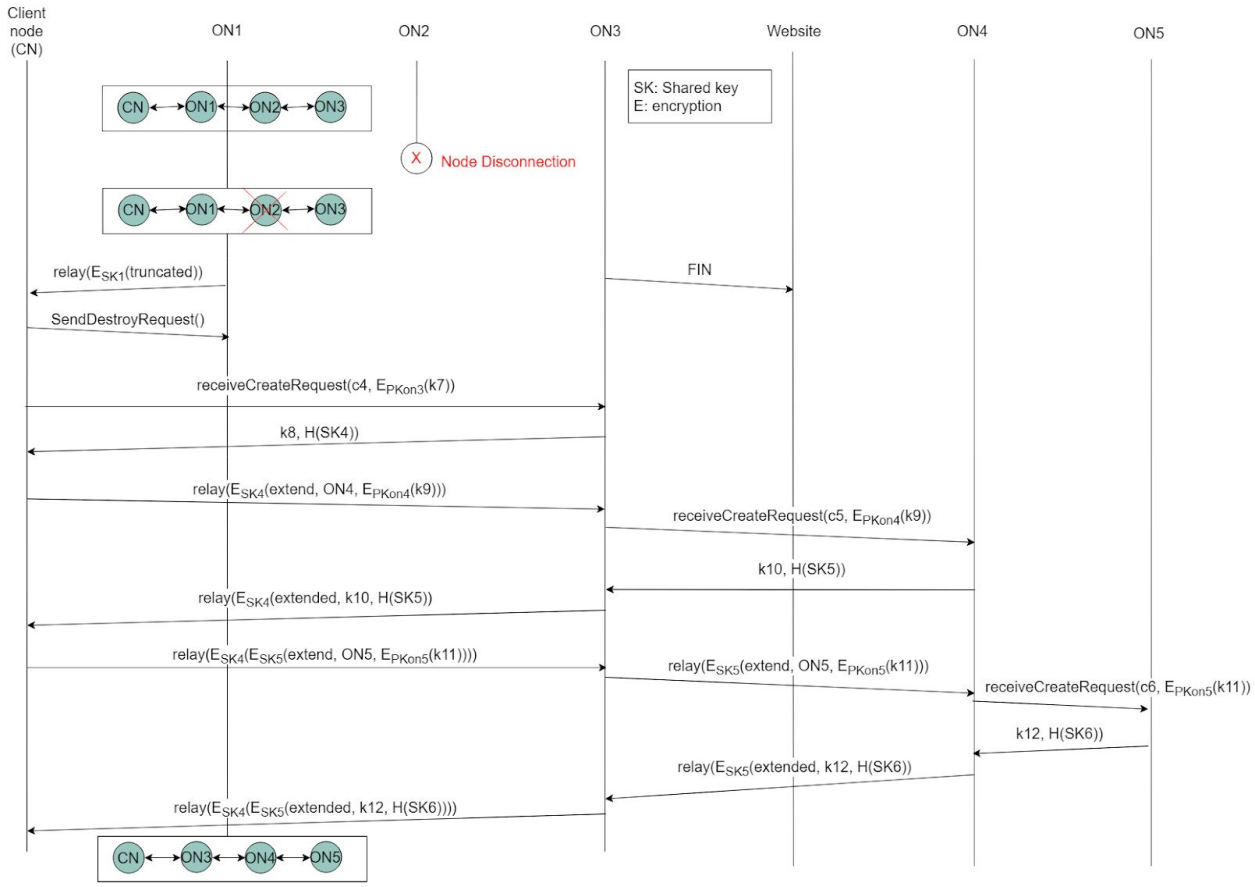
4.1 Normal Operation

Under normal operation, a user is able to join a network using the OnionLib API; performing GET requests for websites supporting HTTP(S). Our network services the requests while keeping the user’s identity anonymous. After a user has finished browsing, they are able to unlink from the network.

4.2 Node Failures

Server Node Failure: A subset of onion nodes are connected to each directory server. If a server fails, the onion nodes that are connected to this server will detect missing heartbeats and attempt to register with another directory server. If all the servers are down, an error will be propagated to the client application since the OnionLib can no longer obtain a list of available onion nodes. The server nodes can detect their neighboring server disconnection through heartbeat failures and flood this information to other connected servers.

Client Node Failure: After client disconnection, the entry node can detect missed heartbeats and send a destroy circuit request to its child node. The destroy request is relayed to the rest of the onion nodes in the circuit in the same manner as circuit termination.



Onion Node Failure: The onion and client nodes that participate in a circuit keep track of their immediate neighbouring heartbeats. If an onion node detects that its circuit predecessor fails, it will propagate a destroy circuit request to its successor. Conversely, if failing of immediate successor is detected by an onion node, it will propagate a message to OnionLib indicating that the circuit has been “truncated”. Once the OnionLib receives a message with a “truncated” command, it triggers a circuit refresh. Note that the client application will not observe any disconnections except when the circuit reconstruction fails due to lack of nodes. The server that the failed onion node is connected to can observe disconnection through heartbeat failure. The server will then update its list of connected nodes and flood the information to its neighboring servers. The flooding protocol checks for cyclic paths to ensure that the flooding will be terminated.

4.3 Node Joins

Server Node Join: A server can join the directory server network at anytime by connecting to server(s) in the network. In this project, failed servers are assumed to never rejoin the system once it is down.

Onion Node Join: An onion node can join the onion network anytime by registering with one of the servers. Once an onion node is registered, it will listen for incoming connections from other onion nodes seeking to establish a circuit.

Client Node Join: A client node can join the onion network using OnionLib API. The request to join the network triggers the OnionLib to create a circuit for the client.

4.4 Threat Model

Malicious Nodes: Our network trusts any new onion nodes entering the system; hence, malicious onion nodes can be injected into the system. When malicious nodes participate in a circuit, they can attempt to observe the payload along intermediate hops. They can also initiate denial of service attacks. This can be accomplished by refusing to send and receive requests or corrupting requests. If malicious nodes act as an exit node, they can close streams or send incorrect responses back to client nodes.

We utilize various techniques to address the participation of malicious nodes in circuits. Our system establishes shared keys with each onion node participating in a circuit with the client node. The client node has access to all the shared keys and encrypts any payload with the shared keys of the onion nodes within the circuit. Any malicious node that tries to observe a payload would be unable to decrypt past its shared key. Refusing to send and receive requests and closing streams is dealt through a timer. The timer is based off a specified constant variable and scales linearly with the size of the circuit. Once the timer is set off the circuit is destroyed and a new one is created to handle client requests. Corrupting packets and sending of incorrect responses are dropped. If a client node receives either of these, the circuit is destroyed and a new one is created to handle the pending client requests. A circuit refresh will attempt to use nodes outside the current circuit and to fill in any remaining nodes through its current circuit in a best effort attempt to get rid of the malicious node(s).

External Entities: Any entity can attempt to perform traffic analysis on the network to determine the circuit patterns which may reveal the identity of a client. This is handled by having a forced circuit refresh based off a timer. Note this timer is different from the one stated for denial of service attacks. The timer for denial of service attacks is only applicable when a client makes requests. The forced circuit refresh timer is used on any circuits whether or not the client is making requests. A forced circuit refresh will not happen if a circuit refresh occurred recently.

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 that 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 does 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.

4.5 Limitations

Server Best Effort Semantics: Replicated directory servers attempt to flood information to its neighbouring servers. However, depending on the server topology and network reliability, consistency of which onion node it contains in its directory is not guaranteed.

Malicious Nodes Repeating Messages: Although our system mitigates the presence of malicious nodes in a circuit for extended periods of time by periodically refreshing circuits, we are unable to handle malicious nodes that repeatedly sends the same message to other relay nodes. Since messages are encrypted, relay nodes cannot distinguish whether or not it is receiving duplicate messages. The current mechanism for ensuring data integrity comes from using a checksum. If an end user sends duplicate requests, the checksum will be identical. Thus, intermediate relay nodes may not know if duplicate cells are valid and from user or invalid and due to malicious nodes. As long as relay nodes receive cells that they can service, they will continue to send them to the next node. For future improvement, a timestamp can be included in the cell payload so that checksums will be unique even for duplicate requests sent by the user.

Inability to Blacklist Malicious Nodes: If a malicious node were to act in a circuit, our system cannot identify which node is malicious. As a result of best effort semantics, circuits may include malicious nodes when a circuit gets refreshed. Because we cannot distinguish between poor network reliability and a malicious node, there is no blacklist mechanism in place for when malicious activity occurs. Thus our system relies on a sufficient number of well behaved nodes in our network to provide reliable browsing.

Timeouts: To combat node failures and malicious nodes, our system requires that we specify timeout intervals to handle edge case failures. The timeout intervals are used to force a circuit refresh when network issues or threats arise. It is difficult to find suitable timeout intervals for any given network due to network variability. Also, depending on where a failure happens and the length of a circuit, there may be excessive overhead when a circuit must be refreshed due to the choice of timeouts. For future improvements, we will need to run tests on different network environments and look into the best relationship with circuit length to choose an adequate time to recover from network issues and threats while improving responsiveness a user would perceive from our system.

5 Reflection

One of the most challenging aspects of the project was dealing with a system with multiple moving parts. The various interactions between different nodes led to many states that were not accounted for during the design phase. Thus exposing various edge cases that had to be dealt with when handling node churn. One of the issues was the timing of a node crash. There could be multiple asynchronous processes that occur concurrently and in different sequences, hence a node crash could happen at any point during the execution of these processes. It is also challenging to determine when connection bugs were fixed due to their non-deterministic nature. Since our time and resources are limited, we have to place a reasonable benchmark to determine when non-deterministic bugs are fixed.

Appendix: External Libraries Used
dhkx**:** Implementation of the Diffie-Hellman key exchange algorithm
go.uuid: Implementation of a UUID generator

References:

[1] https://svn.torproject.org/svn/projects/design-paper/tor-design.pdf
[2] http://ieeexplore.ieee.org/document/668972