TorIRC

An Onion Routing Protocol built in Go to preserve anonymity on the TorIRC

Daniel Chen (u3b9), Daniel Choi (u5g0b), Stella Fang (q5e0b), Qingzhou (Colby) Song (j4w9a)

Introduction

Onion routing is a technique for preserving anonymity on a public network. Anonymous communication is achieved by an originator encapsulating a message in multiple layers of encryption, like an onion. It then sends the encapsulated message through an onion routing network consisting of a collection of nodes called onion routers (ORs). Within the network the encrypted message traverses through a circuit, a series of ORs or nodes. Upon reaching the first node, the guard node, a layer of encryption is removed, like a layer of an onion, revealing the next destination. The message is then sent to this destination and the process of stripping each layer is repeated for each intermediary node, relay nodes, in the circuit. Once the final layer is peeled away by the last node, the exit node, the message is sent to the final destination.

Anonymity is preserved by bouncing the message around onion routers until it is sent to the final destination. This makes it difficult to trace where the message originated from. Each node in the circuit only knows about the previous node and the next node. They are unable to determine whether the previous node was the originator, the next node is the exit node, or if either node is just a relay node like itself, thus preserving anonymity. Only the exit node is able to determine they are the last node in the circuit. In addition, layered encryption increases the difficulty for intermediary nodes within a circuit to view the contents of the message or the path of that circuit.

To further our understanding of the complexities of distributed systems and the challenges of achieving anonymity in a public network, we propose TorIRC, an internet relay chat (IRC) that allows users to chat anonymously amongst each other. The IRC is backed by an onion routing network designed to preserve anonymity by utilizing the onion routing technique described above. We will design and implement an onion proxy (OP) that exposes an API for the user to connect and post anonymously to the IRC.

The underlying implementation will utilize onion routing at its core to relay messages through the network. We will employ encryption techniques to prevent information leakage and a threat detection system to minimize the chances of attackers compromising the system. We will also handle connection/disconnection protocols to improve robustness of our system as well as improve the overall user experience.

Node Types

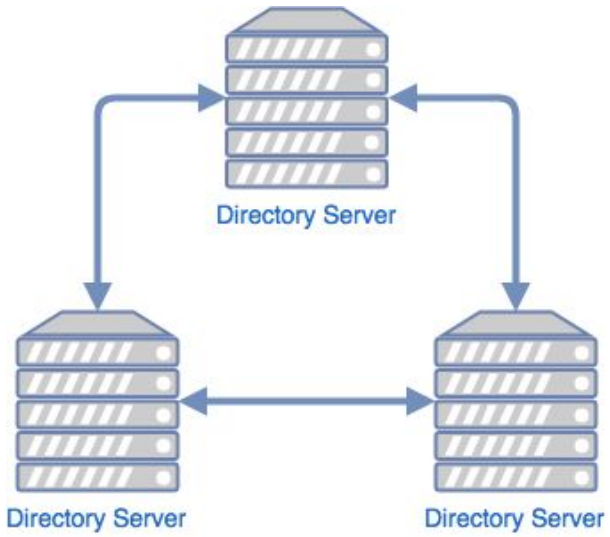
Directory Servers:
The directory servers track the active ORs in the network and their corresponding public keys. They will keep a list of blacklisted IPs of previously determined malicious ORs. OPs will contact a directory server for a set of three ORs to be used in a circuit.

Onion Routers:
The onion routing network is made up of ORs responsible for relaying messages through the network. A circuit is made up of three relay nodes. A relay node within a circuit can act as one of three possible types: a guard node, relay node, or exit node. Each node is unable to determine who the originator of the message or what type of node the previous or next node is. With the exception of the exit node, no other node can determine its own node type within the circuit.

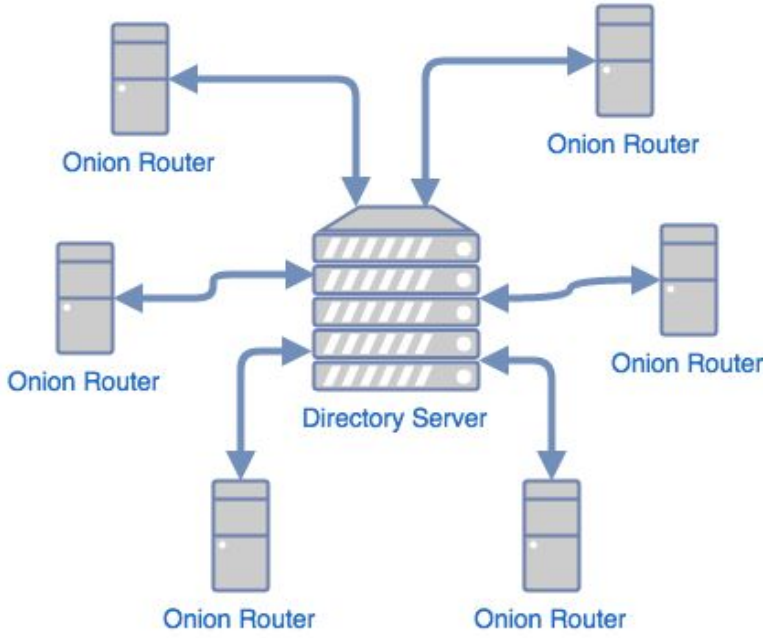
Onion Proxies:
The OP, which is installed on the user's machine, exposes an API to the user for connecting and posting anonymously to TorIRC. During connection, the OP contacts one of the three servers to request a set of three ORs. It begins a symmetric key exchange process with each OR to construct the circuit. When the user posts a message, the message is wrapped in layers of encryption, like an onion, using the symmetric keys and is sent to the entry node.

IRC Server:
The IRC server is where the internet relay chat is hosted. The IRC server exposes an API to the exit nodes to be able to post to the chat.

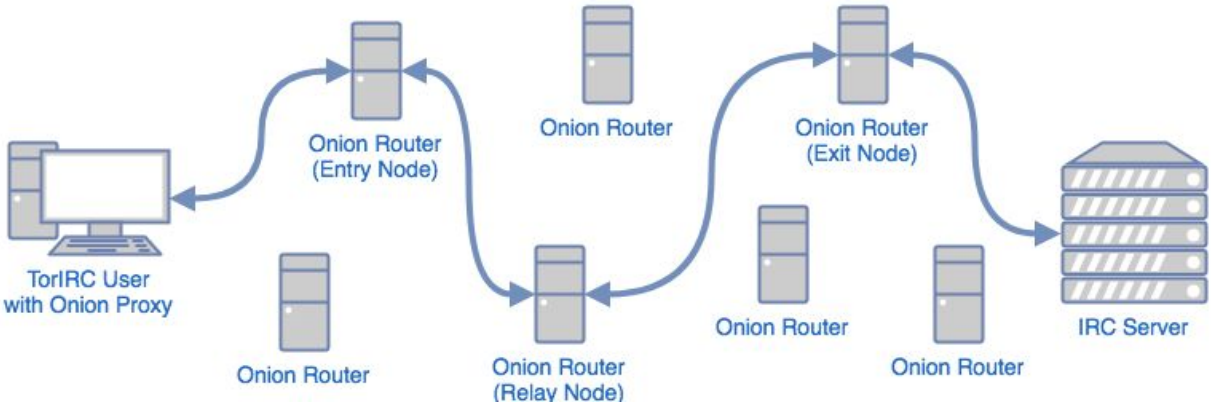
Topology



The figure above shows a fully connected graph between our three directory servers. We will use this topology to synchronize the directory of active and blacklisted ORs.



For each of the three directory servers, there will be a star topology between a directory server and multiple ORs. Each OR will only be connected to a single directory server. ORs will be sending heartbeats to their connected directory server indicating they are still alive.



OPs will be sending encapsulated messages to entry nodes. ORs participating within an active circuit will also be connected to other ORs.

System Architecture

Onion Router Registration:
There will be three directory servers in the network owned by us. ORs wanting to join the relay network must register themselves with a directory server using its Public key and IP address. The directory server only adds the OR to its directory if the IP of the OR does not exist in its blacklist. These directories will be synchronized and redundant, so that they can agree on a common directory.

The directory will contain 2 prime numbers g and p, which will be used to perform the Diffie-Hellman key exchange. These variables will be given to each OR upon registration. When the OR receives these Diffie-Hellman variables, they will compute their half of the key exchange to be used to generated shared keys with different OPs later: g^{OR}.

Circuit Generation:
Whenever a message is requested to be sent from the client, the onion proxy (OP) will make a request to the server asking for the nodes to be used in the circuit. When the directory server responds to the OP with the nodes to be used, the OP must check that it is signed by one of the directory servers. The onion proxy will not use the nodes to construct a relay circuit if the directory is not signed by one of the three trusted servers.

Detection of Malicious Onion Routers:
In our system, we will be focused on detection of routers that are purposely not relaying traffic and therefore leading to lost messages. To protect against these malicious routers and detect compromise in the system (Threat model 2 and 5), the directory server will periodically construct random circuits and send messages through them to check their integrity. If the directory server does not get a message back from the server after sending a message, it will mark each of nodes in the circuit it used as suspicious by incrementing a counter on the node. Once a counter on a node reaches 5, we remove the node from the system and reset counters on all nodes and add the node's IP to the blacklist.

Mitigate Against Exit Node Snooping:
Once the exit node peels away the last layer of encryption, this exposes the payload to the exit node. To prevent exit node snooping of the messages sent by the user, the OP first encrypts the user's message with the IRC server's public key. This ensures that only the IRC server can decrypt and read this message.

Protect Against Traffic Analysis:
Traffic analysis is the process examining messages to deduce patterns (ie. size, timing) of communication, thus potentially revealing the identity of a supposedly anonymous user of TorIRC. To prevent against traffic analysis, all messages/cells sent through the onion routing network will be of fixed size. In addition, each OR will randomly delay for a short amount of

time before sending each message to the next hop. This will greatly increase the difficulty of traffic analysis done on the network.

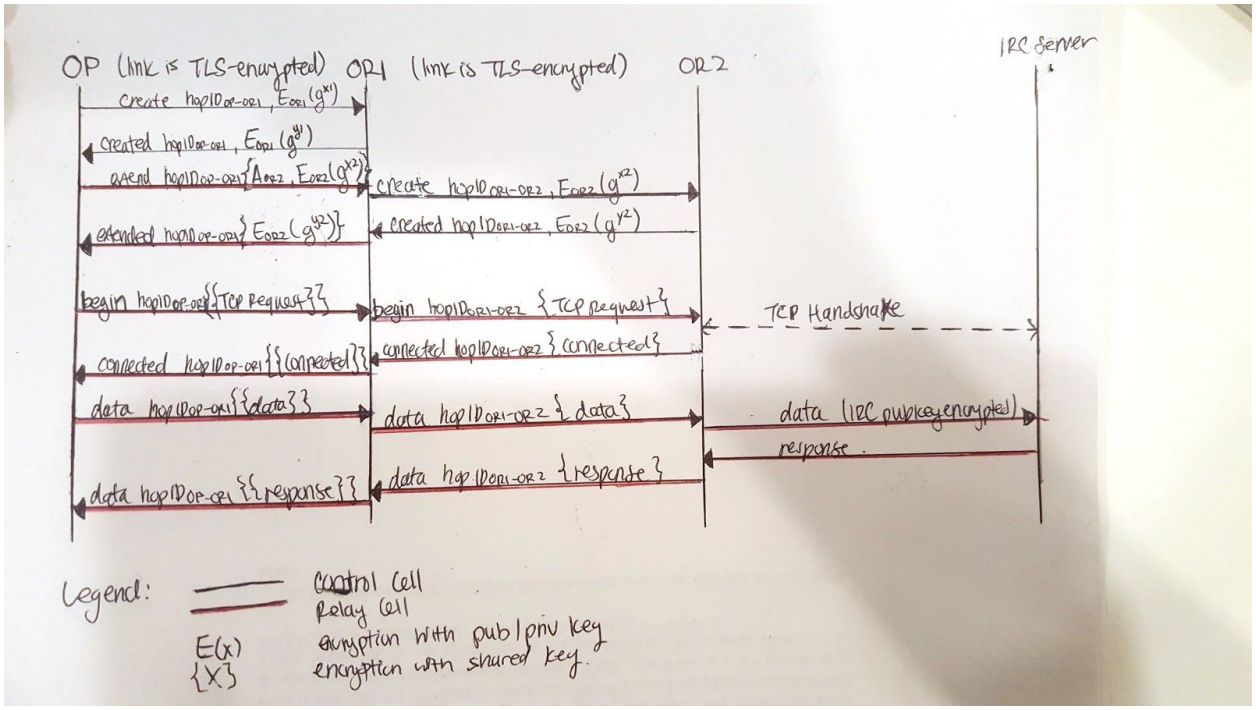


Figure. Circuit Construction (with Diffie-Hellman key exchange and Bidirectional Circuit Relay). See below for more details. Adapted from Dingledine et al, 2004.

Circuit Construction:

When the user calls *connect* the receiving OP resultantly calls *getNodes.* The OP will negotiate a symmetric key with each OR = {OR1, OR2, OR3...} assigned by the directory server to make up the circuit.

OP - OR1 connection:

For the corresponding user address, OP chooses $hopID_{OP\text{-}OR1}$ to represent a new connection to OR1 from OP, where the hopID is not currently used to represent any other connection between OP and OR1. OP will encrypt it's half of the Diffie-Hellman handshake g^x with the public key of OR1, represented by $E_{OR1}(g^{x1})$. $hopID_{OP\text{-}OR1}$ and $E_{OR1}(g^{x1})$ will be sent in a *Control(create)* cell to OR1. When OR1 receives $E_{OR1}(g^{x1})$, it will decrypt the cell with its private key to get g^x in order to create a shared key $K1 = g^{x1y1}$. OR1 will encrypt its half of the Diffie-Hellman handshake g^{y1} with its private key, giving $E_{OR1}(g^{y1})$ and send this and the $hopID_{OP\text{-}OR1}$ in a *Control(created)* cell back to the OP.

When OP receives the *created* call from OR1, it will decrypt it with OR1's public key so OP knows OR1 created the message, and then proceed to use g^{y1} to also create the shared key with OR1, $K1=g^{x1y1}$ which will be the associated with the corresponding user address as the first node in the circuit.

OR1 - OR2 connection:

To extend the circuit to OR2, OP will send a *Relay(extend)* cell encrypted with K1 (shared key OP and OR1) to OR1. This *Relay(extend)* cell will contain the address of OR2 and an OR2 public key

encrypted g^{x2} for OR2, $E_{OR2}(g^{x2})$. When OR1 receives the *Relay(extend)* cell, it will decrypt it with K1 to get the address of K2. Then, it will choose $hopID_{OR1\text{-}OR2}$ to represent a new connection to OR2 from OR1, where the hopID is not currently used to represent any other connection between OR1 and OR2. OR1 associates the $hopID_{OP\text{-}OR1}$ with $hopID_{OR1\text{-}OR2}$ in order to know which address to pass on any incoming Relay cells to marked by the hopID. OR1 will then make a *Control(create)* call to OR2 based on the address given in the *Control(extend)* cell, containing $hopID_{OR1\text{-}OR2}$ and the $E_{OR2}(g^{x2})$. When OR2 responds to OR1 with the *Control(created)* cell, OR1 will wrap the *Control(created)* cell into a *Relay(extended)* cell and pass it back to OR. Now the circuit is extended to OR2 so OP has a shared key with OR2, $K2 = g^{x2y2}$ which will be associated with the corresponding user address as the second node in the circuit

The OP now has established shared keys with each OR in the circuit:
Shared key between OP to OR1 = K1
Shared key between OP to OR2 = K2
Shared key between OP to OR3 = K3
... etc.

Bidirectional Circuit Relay

Once the OP has finished constructing a circuit with the given ORs, the user can call *SendMessage(msg)* to the OP which then the OP will can begin to send relay cells through the circuit via OR1.

In the forward direction, starting from the OP, the nodes in the circuit will incrementally call *Relay(begin)* to request opening a TCP stream, resulting in a TCP handshake with the last OR in the circuit. In response to *Relay(begin),* the reverse direction will incrementally call *Relay(connected)* until it reaches the OP. After this stream is set up, data can now be sent via relay cells. In the forward direction, starting from the OP, the nodes in the circuit will incrementally call *Relay(data)* with the encrypted message from *SendMessage(msg)* sent by the OP. In response to *Relay(data)*, the reverse direction will incrementally call *Relay(data)* with the response until it reaches the OP.

Forward:
The OP will first find the OR1 address, and all the shared keys with each OR in the circuit, associated with the user address it received the *SendMessage(msg)* from. The OP first encrypts the relay cell with its shared key with the largest hop order OR, then the relay cell is further encrypted by its shared key with the next largest hop order OR in the circuit until it encrypts the relay cell with every shared key in the circuit in decreasing hop order. A relay cell will also include the hopID shared between the sender and receiver (OR or OP). When an OR receives the relay cell it looks up the given up hopID to identify the sender which is mapped to the next OR that it should send the relay cell to. Before sending it to the next OR, the OR will first decrypt the relay cell with it's shared key with the OP. This continues until the last OR decrypts the message and identifies it as a call to the IRC server.

Reverse:

The last OR in the circuit will wait for a reply from the IRC server. Once received, it will encrypt the reply with the shared key it had just decrypted the initial message with. Then, it will send the encrypted reply back to the OR it had received the initial message from, together with their shared hopID. This will continue at every OR in the circuit until the message is received by the OP. When the OP receives the encrypted reply, it will decrypt the message with all the shared keys it has with each OR in increasing hop order.

Symmetric Encryption

Using Diffie-Hellman Key Exchange:

1. The OP and OR attain wwo prime numbers *g* and *p* from the directory server
2. Each OP and OR will have the private key *a* and b, respectively.
3. The OP will compute $g^x = g^a \bmod p$ and send g^x to the OR it wants to make a key exchange with.
4. The OR will compute $g^y = g^b \bmod p$ and send g^y to the OP.
5. The OP will then compute $((g^y)^a) \bmod p$.
6. And the OR will then compute $((g^x)^b) \bmod p$.

$((g^x)^b) \bmod p = g^{ab} \bmod p$
$((g^y)^a) \bmod p = g^{ba} \bmod p$
The result is that OP and OR will end up computing the same thing which will be their shared key.

Circuit Renewal/teardown:

Onion proxies will reconstruct new circuits to use every 1 minute. Onion routers will clear hop IDs after 1 minute.

Onion Router Failures:

Directory servers will remove an OR from its directory if it does not receive heartbeats from the OR every two seconds. When the OR reconnects, it will have to register itself again with one of the directory servers. Whenever a message is sent to the IRC server, the IRC server will always respond to the client with an acknowledgement message. If an OP does not receive the an acknowledgement from the IRC server in 5 seconds, it will assume the circuit it used is dead and reconstructs a new circuit.

API Design

In TorIRC, communication exists between OP and OR, between different ORs, and between directory servers and an OP/OR. The path from OP to ORs to the IRC server is called the "circuit."

Communication along the circuit is conducted via TLS connections in order to protect the data along the circuit from being modified by malicious ORs. Fixed-size cells of 512 bytes carry traffic along the circuit. There are two types of cells: *control* cells and *relay* cells.

Control cells ask the receiving node to perform some action related to building and destroying circuits, as well as sending back acknowledgements of completion of those actions.

Relay cells are carried end-to-end through the existing circuit, and they control, extend, and truncate the data stream that flows through the circuit. They also acknowledge successful relays.

In order to extend an existing circuit, a *relay extend* command must be carried all the way to the circuit's current final OR, and that node will interpret the extend command and send a control cell containing a *create* command to a new OR.

The OP exposes the following to the user terminal chat application:

- err <-- Connect(userName)
    - This function is used to get the proxy to establish an end-to-end onion-routed circuit to the chat server. Since the OP is responsible for establishing the circuit, the chat application is agnostic to all details regarding circuit building and teardown. The userName is associated to this user's messages on the chat server, and the user cannot otherwise be identified (i.e. they cannot be identified by IP)
    - NetworkError is returned if the proxy is unable to establish a circuit to the chat server.
- err <-- SendMessage(msg)
    - Once a circuit is established by the OP, the chat client can send messages with content 'msg' to the chat server. Each msg will be associated with the user's userName that was chosen upon calling Connect.
    - NetworkError is returned if the proxy is unable to establish a circuit to the chat server.
- Messages are received in real time through the TCP stream.

OPs (run locally by the IRC user) and the distributed ORs expose the following interface to each other:

- err <-- Control(controlCell)
    - The controlCell is a 512 byte cell containing the circuit ID, command, and payload. The command, which is one of *create*, *created*, and *destroy,* is used to create or acknowledge creation of new circuits between OP/OR or OR/OR, and to destroy those circuits.
    - The err can be InvalidCellError (the controlCell has invalid format) or DisconnectedError (there is a missing acknowledgement somewhere downstream in the circuit)
- err <-- Relay(relayCell)
    - The relayCell is a 512 byte cell containing the circuit ID, relay header, and payload. The relay header contains specifies a relay command: *relay data* (sends data through the stream)*, relay begin* (opens a stream)*, relay end* (closes a stream)*, relay connected* (to send acknowledgement of a successful *relay begin*)*, relay extend, and relay extended* (to extend the circuit by a single OR, and to acknowledge).
    - The err can be InvalidCellError (the relayCell has invalid format) or DisconnectedError (there is a missing acknowledgement somewhere downstream in the circuit)

Directory servers expose this interface to the OP:

- nodeList, err <-- GetNodes()
    - Returns nodeList from a directory server, a partial list of ORs. The list size is determined by the directory servers, and is therefore the length of the circuit that is built by each OP. This list is signed with the private key of one of the directory servers in order to determine the authenticity of the origin. The calling OP assumes that this is the list that is agreed upon by all of the directory servers.
    - The err can be one of DisconnectedError (cannot reach any directory servers), NotEnoughDirectoriesError (cannot reach the threshold number of directory servers), or NotEnoughNodesError (list size is below the threshold number of ORs)

Directory servers expose this interface to ORs:

- err <-- RegisterNode(pubKey)
    - Before an OR can interpret or relay cells, it must register itself with the directory server.
    - DisconnectedError should be returned if no directory servers can be reached.
- err <-- KeepNodeOnline(pubKey)
    - An OR must periodically send a 'heartbeat' to a directory server in order to stay in the router directory. The OR will be automatically removed from the directory by the directory server if it ceases to send heartbeats.
    - DisconnectedError should be returned if no directory servers can be reached.

Directory servers expose this interface to OPs and ORs:

- err <-- MarkNodeOffline(pubKey)
    - When an OR node fails to respond to control or relay cells, the preceding OP/OR must notify the directory server that the node is offline. The directory server then removes that node from its list.
    - The err can be DisconnectedError (cannot reach any directory server).

The IRC server exposes this API to exit OR nodes:

- err <-- RegisterUser(userName)
    - This corresponds to the Connect call made by the user's chat client. A new user called userName is registered on the chat server. This completes the circuit from OP to chat server; the chat server begins to broadcast new messages through the TCP stream that connects the user to the chat server.
    - ConnectionError is returned if the chat server is unreachable.
- err <-- PublishMessage(userName, msg)
    - PublishMessage is used to add a message to the chat server with the corresponding userName and msg. Once the server receives a new message, it will broadcast this message to all connected TCP streams. The message is encrypted with the public key of the chat server.
    - ConnectionError is returned if the chat server is unreachable.

Testing Plan

- Clients will issue chat API calls, and we will be verifying that the server received them and the client can also receive something back from the server.
- When a client issues a sendMessage(msg) API call, we will be verifying that the message is relayed through a subset of the onion routers by printing in each of those routers.
- We will also spin up other clients and make sure that different circuits are used.
- The same client will also issue sendMessage(msg) API call after 1 minute, and we expect to observe a different circuit being used as each circuit should be torn down after 1 minute.
- We will disconnect/reconnect OR and check the directory server's directory gets updated and is synchronized across the three directory servers. We will also verify that the OP reconstructs a new circuit once it learns about a disconnect node along the circuit.

Threat Model:

1. Traffic analysis:
   - Malicious users can observe traffic patterns such as message size, timing and volume to determine the initiator of a message.
     *Handled in *Mitigate Against Traffic Analysis*
2. Selective denial of trustworthy routers:
   - Attackers can DDOS trustworthy routers and make messages to be sent over compromised routers. These comprised routers can be ran by the attacker and used for traffic analysis.
3. Comprising the directory server/fake directory server:
   - Attackers might subvert the directory server to give onion proxies a compromised view of the available nodes leading to construction of a compromised circuit.
4. Exit node snooping:
   - Exit nodes decrypt the final the message sent to the IRC server and can see the contents and sender of the message.
     *Handled in *Protect Against Exit Node Snooping*
5. Lazy Routers:
   - Routers join the network but refuses to relay traffic leading to lost messages.
     *Handled in *Detection of Malicious Onion Routers*

Assumptions:

- There are three trusted directory servers that never go down.
- The clients all have access to the public key of the IRC server which is used to encrypt message sent thereby preventing snooping of the exit nodes.

Assumptions we can not make:

- Relay nodes are always connected.
- Nodes are not malicious.
- Server can not be compromised.

Project Timeline

Deadline	Tasks
March 8	Project Proposal Due (all members)
March 16	Research and understand TOR's cryptography techniques (Stella) Implement Chat Server (Colby) Implement Directory Servers (Daniel Chen)
March 23	Implement TOR relay network (Daniel Choi) Schedule meeting with TA (Stella)
March 30	Implement relay protocol between ORs (Daniel Chen) Handling nodes disconnects (Colby) Implement malicious node detection and recovery (Stella) Implement broken circuit detection and circuit reconstruction (Daniel Choi)
April 6	Testing and write final report (all members)
April 9-20	Demo (all members)

SWOT Analysis

Strength:

- Team members are all familiar and comfortable programming in Golang.
- Good communication and trust built between team members from previous project. All members finished their dedicated tasks on time.
- Team members are capable of learning new material and are willing to work together to tackle challenges.
- Project scope is well defined and lots of online research materials are available on the Tor topic
- Project demo is outlined clearly

Weakness:

- No team member has done any projects related to Tor before
- Tor's cryptography poses a big challenge as members are not familiar with cryptography (ie. RSA and AES)
- Other assignments, exams and projects can hinder the progress

Opportunities:

- Large amounts of material available online on topics related to TOR and cryptography
- Lots of TOR implementations available on Github

Threats:

- Symmetric key cryptography seems to be a complex topic

Azure Deployment

In our demo, we would have 6 onion routers running on individual Azure VMs, one Azure VM for the IRC server, and three azure VMs for each of the directory servers. We will run two chat clients on azure that make API calls to the onion proxy to post messages to the chat. We will show that the messages are relayed through a circuit onion routers and reach the IRC server, and the IRC server can send a response back through the same circuit to the client that sent the request. We will also disconnect one of the nodes in a known relay circuit, and show that the message will now be sent through a new circuit upon reconstruction.

References:

1. Dingledine, Roger, Nick Mathewson, and Paul Syverson. *Tor: The second-generation onion router*. Naval Research Lab Washington DC, 2004.
2. Goldschlag, David, Michael Reed, and Paul Syverson. "Onion routing." *Communications of the ACM* 42.2 (1999): 39-41.
3. Reed, Michael G., Paul F. Syverson, and David M. Goldschlag. "Anonymous connections and onion routing." *IEEE Journal on Selected areas in Communications* 16.4 (1998): 482-494.