GARLIC NOT ONION BREAD

CPSC 416 Project proposal

Sultan Nakyp - c6w8
Trent You - d3n0b
Tina Fan - o1a0
Frances Chong - t3b0b
Jastine Irwan - v4c9

Background and Introduction:

As internet usage increased throughout the world, the need for user and network privacy and security increased. Network traffic sent through traditional IP networks typically only encrypt the contents of the packet itself. For this type of traditional network, the traffic is easily observable from others who watch this network through traffic analysis, as they have the ability to view the source and destination IP addresses of that packet. The destination, as well as the sender of the message, will be clear to all who are snooping on the network, even if the contents of the message will be hidden from potential watchers. This has important implications for the privacy of the users on the internet.

One potential solution towards protection of user privacy is to use an anonymous network, such as TOR, or “The Onion Router”. TOR is used to direct internet traffic, and has an intended purpose to protect the personal privacy of users by obfuscating the source and destination IP address of the message. In TOR, a node only knows about the next node which it is to pass the message to. This is achieved through encryption of the destination of the next node with it’s public key. Thus, at each step, a node will only know about the destination succeeding it.

However, one potential threat to the TOR model is the lack of security on the final node on the path from the sender to destination; anyone snooping on the final node will know of the message’s destination and contents. Furthermore, there are traffic pattern analysis techniques that are able to detect the flow of specific sizes of messages such that they are able to determine the source and destination solely through these patterns.

Our approach is to use a “Garlic” type network that obfuscates the message patterns in the network, such that these analysis techniques are less effective in determining the source and destination of the packets set within our network. Azure Linux VMs will be used to deploy the project.

Approach:

Garlic Routing:
Garlic routing is a modification of onion routing (TOR), with the main difference being message aggregation and use of tunnels instead of node circuits. These two added features help to prevent the reveal of transport details through traffic analysis, such as package deliveries through timing analysis, and package sizes analysis, which is the main concern of the onion routing system.

Tunnels are a collection of nodes that play the role similar to node circuits in the onion routing. There are two types of tunnels, outbound and inbound. When the node wants to send a message to another node, it uses its outbound tunnel to connect to receiver's inbound tunnel and those two tunnels are the path between sender and receiver. Since nodes only know the previous and the next nodes in the path, they will not be able to determine who the sender and the receiver are.

Message aggregation gives us two main benefits. First it increases the message delivery throughput and second it prevents traffic analysis by obfuscating the sizes sent by sender and received by receiver.

Our network will support sending text based messages via TCP between nodes. These nodes will be deployed on Azure VMs. Messages will be encoded in JSON format. We use a 2-way handshake protocol for message delivery. In order for the sender to know that receiver receives the message, the sender has to wait for a confirmation reply from the receiver or a timeout when a failure occurs.

General Lifetime of a Network:

1. Server bootstraps, initializes routerMap.
2. When the first node connects, it adds its router public key and IP address to the server's routerMap.
3. Subsequent nodes join network and build their inbound tunnels and flood tunnel gateway information in the network's gatewayMap.
4. If a node quits a network, server's heartbeat method will detect it and remove the node's entry from the server's routerMap.

Assumptions:

- Our network will be used to send messages (text strings).
- Our server will never go down.
- At least 2 nodes are connected to server at any point of time.
- Create outbound tunnels only when nodes need to send message, delete them afterwards.
- Inbound tunnel is created once the node is alive.
- Each tunnel has its own public/private key pair.
- Maximum length of tunnel is 5.

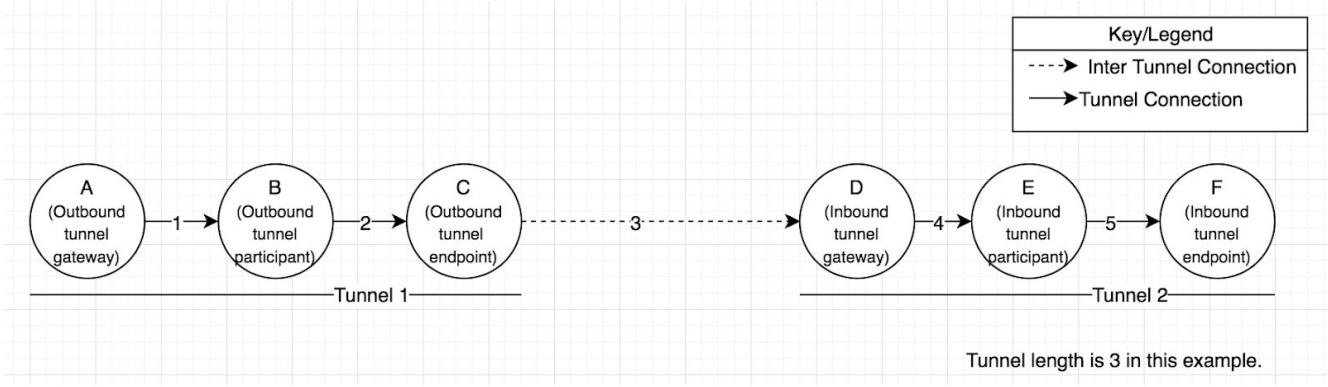


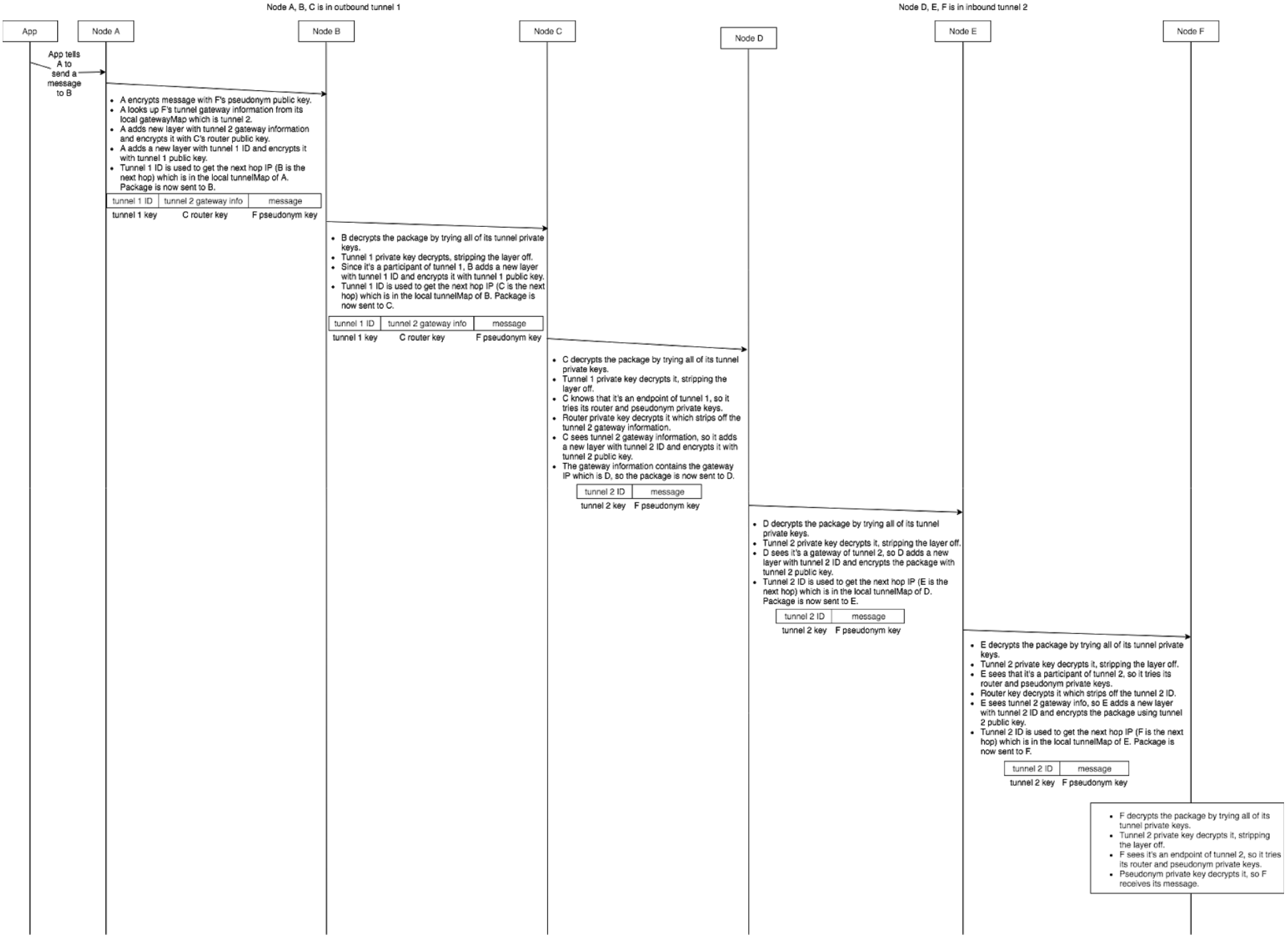
Figure 1.

Tunnels:

Each inbound/outbound tunnel can be either a gateway, participant or an endpoint. Each inbound tunnels has a time to live (TTL) value which is a static value defined inside node.go file. Inbound tunnels have untrusted gateways, which has the tunnel creator as the tunnel endpoint. Outbound tunnels have tunnel creators serving as the tunnel gateway, and the messages are passed out through the outbound tunnel to the destination inbound tunnel. Outbound tunnels are only created when there is a message to send, and expire after the message is sent. In Figure 1, Tunnel 1 refers to the outbound tunnel and Tunnel 2 refers to the inbound tunnel.

- Tunnel gateway: the first router in the tunnel. It pre-packages and encrypts messages into a package and sends it to the tunnel participant. In addition, outbound tunnel gateway is the tunnel creator for outbound tunnel.
- Tunnel participant: all nodes in the tunnel except gateway and endpoint. Peers that are selected by the tunnel creator to be part of the message path.
- Tunnel endpoint: the last router in a tunnel. Outbound tunnel endpoint is responsible for inter-tunnel connection between the outbound tunnel and the inbound tunnel. Inbound tunnel endpoint is the tunnel creator for inbound tunnel.
- Inter-tunnel connection: there could be multiple messages that needs to be sent to different destination(s) in the outbound tunnel endpoint. Those messages need to be repackaged and sent to different inbound tunnel gateways.

Steps on how message will traverse across the network:
Sequence Diagram



Server:
Command line: [*server.go* hearbeat]
Call to server will be used via RPC. Server will contain a routerMap with node's router public key as key and node's ip address as value.

1. err ← Register(nodeIP, routerPubKey)
    - Registers a new node in the map with a router public key as key and nodeIP as value. Returns KeyAlreadyRegistered if the node is already in the server.

2. mapOfAllNodes ← GetAllNodes()
    - Return a map of nodes currently registered in the server and their information.

3. err ← HeartBeat(routerPublicKey)
    - The server also listens for heartbeats from known nodes. A node must send a heartbeat to the server every 2 second after calling Register, otherwise the server will stop storing this node's address/key in its local map. Returns UnknownKeyError if the server does not know a node with this publicKey.

Node/Client:
Command line: [*node.go* pseudonymPublicKey pseudonymPrivateKey]

Figure 4 - Structs for the tunnel.

The file node.go generates a router key pair, and is used as a sender/receiver identify IP addresses in the server's routerMap without giving away the pseudonym key. Pseudonym key pair will be passed in as an argument in the command line to the node. It is used to encrypt and decrypt messages sent

Each node is an inbound tunnel endpoint. If a node needs to send a message, it has to create an outbound tunnel. In this case, the node is now also an outbound tunnel gateway. A node may be part of more than one tunnel. Each node will contain a gatewayMap with pseudonym public key as key and an array of tunnelGatewayInfo struct as value. It also contains a tunnelMap with tunnelID as key and tunnelInfo struct as value.

Tunnel expires when the TTL which is a static integer goes to 0, a new tunnel (gateway, participant and endpoint) with a new ID and public/private key will be created. When an inbound tunnel is created, the information of the inbound gateway will be flooded to all other nodes so that other nodes can update their gateway map.

1. err ← SendMessage(pseudonymKey, message)
    - Sends a message by looking up pseudonymKey in the gatewayMap on the server to find which inbound tunnel to send to.
    - Return TimeOutError if a node in the tunnel get disconnected or message delivery failed.

All other functions will be local to the node.go and will not be exposed to the App.

Tunnel creation:

1. The current node will be the tunnel outbound gateway if it is an outbound tunnel; it will be an inbound tunnel endpoint if it is an inbound tunnel.
2. Call GetAllNodes() on the server.
3. Randomly choose nodes obtained as tunnel participants, the last node picked will be the outbound tunnel endpoint.
4. Contact each chosen node to set their tunelInfo.
5. If it is an inbound tunnel, distribute tunnel gateway information in the network's gatewayMap.

gatewayMap update process for inbound tunnel:

This is slightly different from message delivery protocol.

1. The creator of the inbound tunnel gets the IP addresses and router public keys of all nodes in the network from the server.
2. The creator picks one of those nodes and makes it an outbound tunnel only for the purpose of distributing the inbound tunnel gateway information.
3. The creator encrypts the inbound tunnel gateway information with the receiver nodes' public router keys. Then it adds a layer with values of:
    - 0 for tunnelID
    - IP address of the receiver for tunnelGatewayIP
    - nil for tunnelPubKey

    Finally, it adds another layer that includes information on the tunnel ID and public key.
4. We will send this information/package via the outbound tunnel. At the outbound tunnel endpoint, we will decrypt the tunnel gateway information which contains tunnelID, tunnelGatewayIP and tunnelPubKey. We observed that the tunnelIID is 0 and the tunnelPubKey is nil, therefore we send the message directly to the intended receiver located at tunnelGatewayIP skipping the inbound tunnel.
5. Receiver receives the message and decrypts it with its router key. The receiver sees the sender's inbound tunnel and it updates its local copy of gatewayMap.

Node failures:

In the case of a failed node, there are several scenarios in which the failed node could impact our network:

Node as part of an inbound tunnel:

Inbound tunnels remain in service longer than the temporary outbound tunnels. As such, the receiving node should constantly be monitoring (e.g. using heartbeat) for node failures of nodes that are part of its inbound tunnels. If a node that goes down is part of an inbound tunnel, it will break the tunnel chain. If the tunnel chain breaks, the node will not be able to receive any messages through that tunnel. Then, the receiving node must rebuild that tunnel using the list of available nodes retrieved from the server.

Node as part of an outbound tunnel:

The node that disconnected will not be able to send the package to the next node. In this case, as message fails to reach the receiver, there won't be any reply message and sender will time out. We will receiver an error in the case.

Node that was the the target of the message:

This is an unavoidable node failure and should be checked for through the server before the sender initiates sending a message to that node. If node rejoins from a failure, it will be treated as a new node.

Threat Model Specification:

Threats handled by our network:

Exit Node Vulnerability:
Garlic routing is a modification of the onion routing specification. On an unprotected network, an attacker in control of a single node is able to view packets passing through the node and observe the original source and final destination. An improvement would be the use of onion routing, which encrypts the address of subsequent hops. However, onion routing is vulnerable to attacks on its exit node, as it performs the final decryption before the message is passed to the recipient. An attacker in control of the exit node has access to all the decrypted information of the message before it is passed to the destination node.
With garlic routing, when an attacker is in control of a single node, they will have the information on the IP address of the next hop in the tunnel. However, they will have no idea whether the next hop is another node in the tunnel, or the final destination of the message; the message decryption is performed on the destination node itself.

Brute Force:
Another problem that affects onion routing is the vulnerability to observation of network traffic patterns on a large scale. While the source and destination are hidden even when a non-exit node is attacked, someone observing the entire network has the ability the follow the path of a particular sized (say 5GB) message throughout the network. This method can potentially reveal the sender and destination of a message, depending the amount of resources available to the attacker (how much of the network they can monitor). Garlic routing implements additional protections against this threat, such as the packaging of several messages together to make the signatures of messages much more *difficult* to follow in the network.

Threats our network are vulnerable to:

Brute force:
Attempting to figure out traffic patterns in the network is much more difficult with Garlic routing, but it is still possible by those with the ability to observe a large ISP or an internet exchange point (i.e., the resources to monitor the network on a large scale).

Timing attacks:
Some protocols have predictable message patterns, such as HTTP, which have protocol establishment, replies, and resends in case of failures. It may be possible to pick up on these patterns in certain cases, without techniques such as protocol scrubbing or intentional network delays.

DDOS attacks

Starvation attack:
An attacker could create a large number of unresponsive peers in the network, causing “good” peers to require the need to make much higher levels of requests for lists of peers than needed. This will cause a higher level of difficulty in creating tunnels, and increase the latency throughout the network. Furthermore, these unresponsive peers could perform more malicious acts, such as performing the correct operations, but providing intermittent service such that it would be difficult to distinguish these peers from overloaded or failing peers.

Flooding attack:
An attacker could flood the inbound tunnels of target with an extremely high count of message packages, causing the failure of these tunnels. The Garlic protocol has no way to preventing this type of flooding.

Floodfill DOS attack:
A malicious user introduces a “floodfill” node, which intentionally introduces errors and bad responses to interfere with internode communication.

Tagging attack:
Normally, tagging a message so that it is identifiable down the tunnel should be impossible in Garlic routing, due to the signing of messages. However, if a malicious user controls an inbound gateway as well as another node down the path, the user can correctly identify that the two nodes are part of the same tunnel.

Challenges:

Security will be an issue as we do not want to store all information of a node (eg, private/public keys and tunnel info) inside the server. If the server is attacked, the attacker will know everything about a particular node. Tunnels should be updated and changed periodically.

It might take a lot of work and research to set up a proper/correct way to set up the tunnels, encrypt and decrypt messages on the inbound and outbound tunnels. We might have to change the way on how we implement the tunnels when we encounter problems during our implementation.

Nodes can connect or disconnect at any point in time. This will be a problem if a message is currently on route; package will not be able to reach the desired destination. We need to be able to detect that a node in the current tunnel is no longer alive and the package needs to be resent using a new tunnel. This will be handled by the 2 way handshake.

Another challenge we might have is we need to flood the information about offline nodes to each node in the network, to update their inbound tunnel array, if that node is part of one of their inbound tunnels. If we are reliant on node to node communication, we may have issues if the flooding is reliant on a node that just went down.

SWOT Analysis:

Internals (strengths/weakness):

Strengths:

- All team members have worked with each other before, so we are familiar with each other's work style/habits.
- Team members will meet up almost everyday to work on the project.
- Team members are all very flexible and helpful. (We don’t mind what tasks get assigned to each other).
- Sultan is well informed about Garlic Routing so he will work on tunneling creation.

Weaknesses:

- Only one of us heard about Garlic Routing.
- We spent too much time trying to perfect one part of the project and ran out of time on the other parts (for Project 1).

- We can get easily distracted.
- Our project is based on an existing system that is not well documented so we made many assumptions due to lack of resources.
- All team members only started learning and using Go in this class.

Externals (threats/opportunities)

Threats:

- We ran out of time because we have too much coupling/dependencies.
- Commitment to exams or assignments from other courses may interfere with progress.
- There is no set guideline for the project, so it is easier to get off track.
- Exiting nodes in the network can be very complicated to deal with for our model.
- Garlic Routing is very underutilized so it might be hard to find documentations and examples online.

Opportunities:

- We will learn a lot on anonymous networks, encryption/decryption and failure recovery.
- We will be security experts in Go at the end of this project.

Testing Plan:

In order to make testing easier, we will be creating automated scripts. We will create a test script for tunnel creation. We will also have a test script that check for basic error handling.
We will have an app that does integration test:

- The API will be tested by having an app telling node to send messages to other nodes.
- Two apps will send messages concurrently from node A to B and node B to A.
- Try to send a message to a node that is not online.
- Test with 5 - 10 nodes running.
- We will test the system for node failures force killing nodes and observing that other nodes are still alive and their functionality still works as expected.

Testing App:

Command line: [*app.go* pseudonymPublicKey]
An app that send messages from one node to another. When a node receives a message it will print the message to the stdout for confirmation. Receiver will send a confirmation message back to the sender. Sender will print that confirmation message to stdout. App should also test for case where nodes where disconnected and a timeout occurs. More testing app will be added when testing for concurrency.

Timeline:

Deadline	***Task***
March 9th	Proposal Due
March 12th	Write skeleton code and stubs for all the APIs: - Jastine (*v4c9*) will create the stubs for the server as she has experience with implementing the server in A2. - Trent (*d3n0b*) and Frances (*t3b0b*) will create the stubs for the nodes for, as they did an excellent job implementing miner nodes in Project 1. - Sultan (*c6w8*) and Tina(*o1a0b*) will create the stubs for the tunneling, as they have read further into the details and the functionality for tunnels in Garlic Routing. - The team will help each other with any questions or concerns regarding any areas.
March 14th	Implement server functions: - Sultan and Tina will work together to implement the following APIs in the server: Register, GetAllNodes, HeartBeat.
March 19th	Implement tunnel creation: - Sultan and Tina will continue to work on the tunnel creation and functionality. Implement node/client function: - Trent, Frances and Jastine will work together to work on node functionality. Implement/Test single node-server connection: - The team will all partake in test creation to help familiarize themselves with the code, and provide knowledge transfer on each part of the implementation.
March 23th	Test 2 nodes connection and write a simple app for it - The team will all partake in test creation to help familiarize themselves with the code, and provide knowledge transfer on each part of the implementation.
March 31 - April 6	Testing and Debugging - The team will all partake in testing and debugging together, and will resource help from online and TAs. Complete final report: - The final report will be written throughout the timeline, and the team conduct a final meet up to complete and edit the final report.
TBD	TA Update Meeting - Feedbacks and questions.
April 6	Code and final reports due
April 15 (tentative)	Demo date

References:

https://geti2p.net/it/docs/how/garlic-routing
https://www.torproject.org/about/overview.html.en#whyweneedtor