GARLIC NOT ONION BREAD

CPSC 416 Project Report

Sultan Nakyp - c6w8, Trent You - d3n0b, Tina Fan - o1a0b, Frances Chong - t3b0b, Jastine Irwan - v4c9

I. Abstract

Internet anonymity and security is an important area of focus, especially in countries with censorship and government controlled internet. While onion routing (e.g. Tor) is a popular system for anonymizing traffic on the internet, it suffers from vulnerabilities such as the exit node vulnerability and brute force observation of the network. Our system is a variation on the onion protocol, which utilizes transient inbound and outbound tunnels to send messages between nodes, also known as I2P, or garlic routing. This avoids some of the vulnerabilities present in the onion protocol, while adding additional security features. Finally, the system implements methods of recovery in the case of failures of nodes that make up the inbound and outbound tunnels within the system.

II. Introduction

The need for anonymous communication has become increasingly important for the modern internet. As the internet has become more ubiquitous, the threats to secure communication have become more serious. These include a variety of malicious agents, from individuals to entire governments with the capability of spying on network communication. Users of anonymous networks include residents of countries with internet restrictions, whistleblowers, journalists, as well as individuals who value privacy in general. This makes anonymous networks particularly important as government regulations become stricter in certain countries, and laws like net neutrality are repealed in the United States.

Tor, a popular platform based on the onion routing protocol, is commonly used as an anonymous network. It operates using an “onion” analogy, such that each layer is encrypted using the public key of the next node along the path from sender to receiver. The sender chooses a set of nodes to use and encrypts the message starting with the final node, working backwards and adding additional layers for each node along the path. Thus, each node along the path except the final one knows only the next destination in the path; compromising one would only reveal the next hop’s address. One issue with this implementation is that the final node knows not only the message destination, but also the contents of the message itself. A compromised final node is thus a high security risk inherent in the onion protocol.

Our goal is to create a system based on a variation of the onion protocol known as I2P, (also called garlic routing). Instead of a path, this protocol uses temporary “tunnels” to pass messages between sender and receiver. A tunnel between two nodes is made up of an outbound and inbound tunnel; the outbound tunnel originating from the sender and the inbound tunnel pointing at the receiver. An outbound or inbound tunnel is created from at least one node within the network (excluding itself), but can be of any arbitrary length. Thus, each node serves a dual role within the network; it can be used as a sender or receiver, but it can also serve as a participant in a tunnel, passing along messages for other nodes.

In order to initiate sending a message, the sender creates an outbound tunnel chosen from nodes within the network, then sends the message through that tunnel. The sender will then create an “onion-like” package using multiple layers of encryption, which will be discussed further below. The package is passed from node to node until it reaches the endpoint of the outbound tunnel, at which point it is rerouted to the inbound tunnel of the receiver. The inbound tunnel of the receiver then passes the package down the tunnel until it reaches the receiver itself. One feature of this protocol is that each node has no notion of whether or not the next hop is the final destination; only after the message reaches a node do we know if the message was intended for that node.

Furthermore, the outbound tunnels created for each message are closed after the message is sent, and are replaced \ for the next message. Inbound tunnels are more constant, handling multiple message receives, but are still replaced on a consistent basis.
A single node can send multiple “onion-like” packages by aggregating them to a single “garlic” package. The “garlic” implementation affords additional security over the traditional onion protocol, by reducing the vulnerabilities on the exit node, while obfuscating network traffic by combining multiple messages into a single sent package whenever possible.

Definitions and Acronyms

Outbound Tunnel:

A temporary tunnel created by a message sender using available nodes in the network. Used to send a message to another node. Consists of at least a gateway node (the sender node), an endpoint node, and any number of intermediary tunnel participants. Destroyed after message has been sent.

Inbound Tunnel:

Tunnels created by each node in the network to receive messages. Consists of at least a gateway node, an endpoint node (the receiver node), and any number of intermediary tunnel participants. Each node creates a single inbound tunnel and replaces them after a set amount of time (10 seconds). Once created, inbound tunnel information will be flooded to every node currently in the network.

Tunnel Gateway:

The first node in a tunnel and is the sender of the message (in an outbound tunnel).

If it is an **outbound** tunnel gateway, it pre-packages and encrypts message into a multi-layered “onion” package and passes it on to subsequent tunnel participants. If it is an **inbound** tunnel gateway, it encrypts the received data using the inbound tunnel key, then passes the package onto subsequent tunnel participants.

Tunnel Endpoint:

The last node in a tunnel and is the receiver of a message (in an inbound tunnel).

If it is an **outbound** tunnel endpoint, it repackages the message using the inbound tunnel AES key found in the tunnel gateway info and passes it on to the inbound tunnel gateway. If it is an **inbound** tunnel endpoint, it is the receiver of the message itself, and simply decrypts the message using its own pseudonym private key. If the message contains an AES key, it will store it locally. If the message is an actual message, it will sent a reply if needed.

Tunnel Participant:

An intermediary node within a tunnel. Can be any node within the network and each node can be a participant in multiple tunnels.
Only functions to decrypt the package, determine what the next hop address is, re-encrypt the message using the tunnel AES key, and pass it down the tunnel.

AES Key:

An AES shared between two nodes to facilitate communication between them. Sent by the sender to the receiver as part of establishing an initial communication. Used to encrypt the message itself for subsequent messages received from the sender.

In addition, tunnel participants have tunnel AES keys and tunnel endpoints have their own individual AES keys, to decrypt tunnel information in order to pass messages down the tunnel.

Pseudonym RSA Key:

A key pair used to send an AES key to a certain “pseudonym”, without revealing the IP address of the node that the pseudonym belongs to. Used by the sender of a message to send their shared AES key to the receiver, so that the receiver can decrypt subsequent messages exchanged between the two nodes.

Router RSA Key:

A publicly known key pair used to identify a node without needing to use its pseudonym key. Used for flooding tunnel roles and gateway information to nodes who were chosen to participate in a tunnel. It is also use to register a node to the server.

III. Design

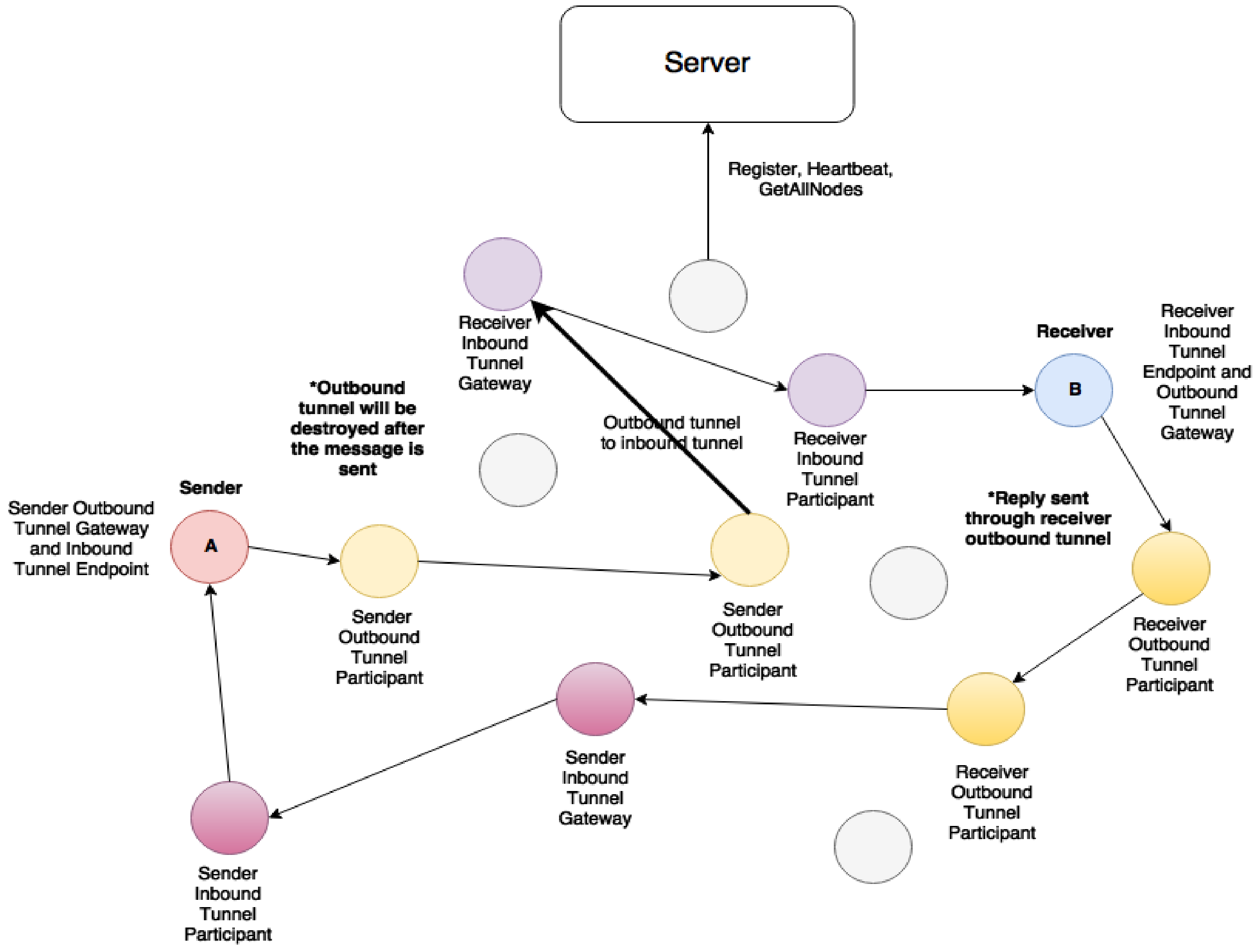

Server
Register, Heartbeat, GetAllNodes
Receiver Inbound Tunnel Gateway
Receiver
B
Receiver Inbound Tunnel Endpoint and Outbound Tunnel Gateway
*Outbound tunnel will be destroyed after the message is sent
Outbound tunnel to inbound tunnel
Receiver Inbound Tunnel Participant
Sender
A
Sender Outbound Tunnel Gateway and Inbound Tunnel Endpoint
*Reply sent through receiver outbound tunnel
Sender Outbound Tunnel Participant
Sender Outbound Tunnel Participant
Receiver Outbound Tunnel Participant
Sender Inbound Tunnel Gateway
Receiver Outbound Tunnel Participant
Sender Inbound Tunnel Participant

Figure 1. Message Sending Through Tunnels Composed of Nodes

Our system is composed of only a single type of node, and a centralized server that handles registering new nodes and informing current nodes of the state of the network. In addition, tunnels are conceptual objects made up of at least one node and form a tunnel that forwards messages between sender and receiver.
Conceptually, each node is capable of adopting three roles: **sender**, **receiver**, and **tunnel participant**. Within the tunnel participant role, a node could also be a **gateway**, a **normal participant**, or an **endpoint**.

Furthermore, each node is capable of occupying several roles at the same time, when participating in multiple tunnels. Each node can also play multiple roles in a tunnel. At any time, each node is capable of sending a message to any other node.

Tunnels

Tunnels are an abstract concept in our system, created from nodes in our network to form a “linked-list” of nodes in order to propagate messages between a sender and receiver. Tunnels are composed of three types of nodes, a Gateway node as the first node in the tunnel (and first node to receive messages), a Participant node that propagates messages to the next hop ip in the tunnel, and an Endpoint node that either passes the message to an inbound tunnel or opens the message itself.

Tunnels are composed of at least two nodes, a Gateway node and an Endpoint node. In the case of an outbound tunnel, the sender occupies the role of the Gateway node and in the case of an inbound tunnel, the receiver occupies of the role of the Endpoint node.

Tunnel “Onion-like” Package Format:

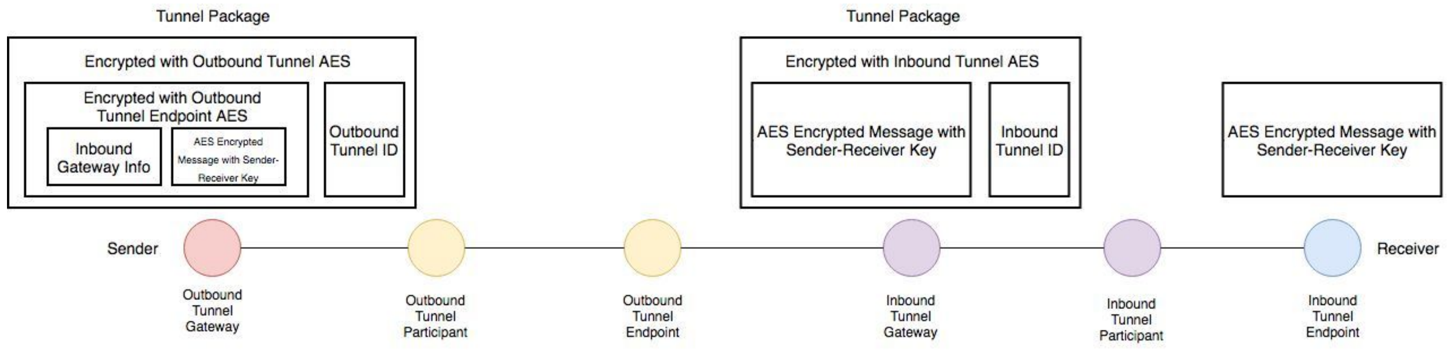

Tunnel Package
Encrypted with Outbound Tunnel AES
Encrypted with Outbound Tunnel Endpoint AES
Inbound Gateway Info
AES Encrypted Message with Sender-Receiver Key
Outbound Tunnel ID
Tunnel Package
Encrypted with Inbound Tunnel AES
AES Encrypted Message with Sender-Receiver Key
Inbound Tunnel ID
AES Encrypted Message with Sender-Receiver Key
Sender
Receiver
Outbound Tunnel Gateway
Outbound Tunnel Participant
Outbound Tunnel Endpoint
Inbound Tunnel Gateway
Inbound Tunnel Participant
Inbound Tunnel Endpoint

Figure 2. Tunnel “Onion-like” Package

To package a message to propagate through both the outbound and inbound tunnels of sender and receiver, we must include multiple layers of encryption that are stripped off as the message passes between the tunnels to the receiver.

The message itself is encrypted with the AES key shared between the sender and the receiver.

An outbound tunnel endpoint also has a unique AES key that we encrypt the information about the destination inbound tunnel with, so that only the endpoint can decrypt the information on the inbound tunnels to send to.

The information on the destination inbound tunnel, (i.e. its IP address), is combined with the encrypted message itself and the entire package is encrypted with the outbound tunnel endpoint AES key mentioned above.

Each tunnel is assigned a single AES key used to decrypt and re-encrypt messages that are passed through the tunnel. This key is distributed between *all* the tunnel participants (including Gateway and Endpoint nodes).

Finally, the tunnel ID information (an id that uniquely identifies a tunnel that a node is part of) is packaged with the package of information mentioned above (encrypted with the outbound tunnel endpoint AES key), and encrypted with the tunnel AES key shared between all nodes in the outbound tunnel.

This entire package is sent to the outbound tunnel of the sender, and parts of the encryption are either stripped off or added at various points of the tunnel process.

Outbound Tunnel:
At each node in the outbound tunnel, the node attempts the decrypt the package using its tunnel AES key. If successful, it examines the tunnel ID contained within the package, and identifies the correct tunnel the message belongs to. The node re-encrypts the package (packs the message up again), and sends it to the correct next hop in the tunnel.

When the package reaches the outbound tunnel endpoint and after it decrypts the package using the AES key, it further decrypts the inbound tunnel information using its unique endpoint AES key.
It discovers the address of the inbound tunnel gateway and the AES key associated with that inbound tunnel. Then, it prepares a second layer of encryption for the message after stripping off the outbound tunnel and outbound tunnel endpoint information.
This second layer of encryption encapsulates the tunnel ID of the inbound tunnel, combined with the encrypted message itself. This entire package is encrypted with the inbound tunnel AES key, and sent to the inbound tunnel gateway.

Inbound Tunnel:
The inbound tunnel gateway node acts similarly to any other participant node; it decrypts the incoming messages from outbound tunnels using its tunnel AES key, identifies the tunnel ID that the message belongs to, then re-encrypts the message and sends it on its way to the next hop.

This behavior is the same for all inbound tunnel participants, which simply decrypt, identify, re-encrypt and pass the message on to the next hop.

This behavior only changes for the inbound tunnel **endpoint**, which is the destination of the message. There are two ways that an endpoint can handle a message sent to it.

a. If the message is able to be decrypted by the Pseudonym Key of the node, it knows it is an AES key message from another node attempting to initiate communication with it.
b. If an AES key is present and manages to decrypt the message, it is an actual message from another node, and the node decodes the message using the shared AES key.

Tunnel Creation:

Inbound tunnel: A node will create a new inbound tunnel every 10 seconds. The node creator will be an inbound tunnel endpoint. The tunnel creator will ask the server for a current list of connected nodes in the network, then choose nodes randomly as tunnel participants. This information will be flooded to the network using the tunnel gateway flooding protocol described below.

Outbound tunnel: A node will create a new outbound tunnel each time it wants to send a new message, and will delete it after the message sending is complete. This sending node acts as the tunnel outbound gateway and the last node picked will be the outbound tunnel endpoint. The chosen node will be contacted of its role directly. No flooding is necessary.

Tunnel Gateway Flooding:

As our system creates inbound tunnels on a frequent basis (every 10 seconds), we must flood this Tunnel Gateway information package to every single node in the network, so that nodes have the correct inbound tunnel information to send a message to a particular pseudonym. This package includes information such as the tunnel gateway IP address and the inbound tunnel AES keys. This key is used to encrypt the message package before passing it through the inbound tunnel.

Tunnel Gateway Information Flooding Protocol:

1. The creator picks one of the nodes in the network and makes it an outbound tunnel only for the purpose of distributing the created inbound tunnel gateway information.
2. The creator encrypts the inbound tunnel gateway information with the receiver nodes' public router keys. Then it adds a layer with values of:
    - - 2 for tunnelID
    - IP address of the receiver for tunnelGatewayIP
    - nil for tunnelPubKey
3. Finally, it adds another layer that includes information of the tunnel ID which is -2 which is not encrypted
4. We will send this information/package via the outbound tunnel. At the outbound tunnel endpoint, we will strip the outer layer and decrypt the package, which contains tunnelID, tunnelGatewayIP and tunnelPubKey and the encrypted message which contains the tunnel gateway information that we want to flood. We observe that the tunnelID is -2 and the tunnelPubKey is nil, therefore we send the encrypted message directly to the intended receiver located at tunnelGatewayIP, and skipping the inbound tunnel of the receiver.
5. Receiver receives the message, decrypts it with its router key and updates its local map.

Sender Role:

The sender role is for any node that wants to send a message to another node in the network. Nodes are identified using their pseudonym keys, and mapped to information on their inbound tunnels. A node sends a message to a **pseudonym**, rather than a specific IP address.

To initiate sending the message, our node first retrieves a list of all nodes in the network, then constructs an outbound tunnel from a randomly chosen list of nodes in the network, with at least one node in the list.

Next, the node checks whether a shared AES key exists and if it doesn't yet exist, the node creates a new AES key. It encrypts the AES key using RSA and sends it through the outbound tunnel to the receiver it wants to send message to.

Once both nodes have the shared AES key, the sender encrypts the message with the shared AES key. Next, the encrypted message is combined with the address information for the inbound gateway of the receiver, which is encrypted using the outbound tunnel endpoint AES key. Finally, the entire package is combined with the outbound tunnel id and encrypted with the outbound tunnel AES key, completing the “onion” package, and sent to the first outbound tunnel participant.

Receiver Role:

The receiver role is the endpoint of all of the inbound tunnels it owns. This node role only has a single function, which is to attempt to decrypt the final contents of the message using the shared AES keys it has stored. As it knows it is the endpoint of the inbound tunnel, and therefore the receiver of a message, it will go through all of its shared AES in an attempt to decrypt the message it receives. Once decrypted, it reads and prints out the message received.

Tunnel Participant Role:

All nodes are eligible to be tunnel participants, and are chosen at random by nodes creating tunnels. Tunnel creators obtain a list of all the nodes in the network, then build their tunnels from randomly chosen nodes.

Tunnel participants are meant to pass on messages sent to it, to the next hop in the tunnel. Since participants might be a part of multiple tunnels, whenever a participant receives a message, it attempts to decrypt the message using all of the AES keys it holds. The AES key that decrypts it will clue the participant to the tunnel the message belongs to, and allows the participant to pass on the message on to the correct next hop; after the participants discovers the correct tunnel to which the message belongs, it re-encrypts the message and passes the message on.

Server

The server is responsible for registering all new nodes, monitoring their heartbeats, generating node ids/tunnel ids, and providing a list of current nodes in the network when nodes request this information.

For our system, we assume the server will never go down.

IV. Implementation

Node Initialization

When a node starts up, it generates a Pseudonym Key (RSA) and a Router Key (RSA). Next, it registers its node ID with the server. Finally, it sets up a heartbeat with the server and configures both its TCP and RPC connections. It then prints its Pseudonym public Key, node ID, and its RPC IP address to the console. This is required by the client to send messages to other nodes. Next, it makes sure that it has the latest gateway information all current nodes in the network. Lastly, it creates and floods its own inbound tunnel information to all other nodes in the network. It will now begin the process of recreating inbound tunnels and flooding them to all other nodes in the network every 10 seconds.

Message Send Flow

Communication Initialization Phase:

This phase occurs before the first message is sent between two nodes. In order to establish communication, the sender prepares an onion package containing an AES key meant to be shared between the two nodes. This is encrypted using the public pseudonym key of the receiver (RSA encryption) and packaged in the format used to send a message through a tunnel (detailed in the Tunnel in the Design section). The sender creates an outbound tunnel, and this package is sent through that tunnel. The package propagates down the outbound tunnel until it reaches the outbound tunnel endpoint, then deposits it to the inbound tunnel of the receiver, which eventually propagates it to the receiver. The receiver then decrypts the message using its pseudonym key, and retrieves the AES key stored in the message. It stores the AES key to decrypt the anticipated message to be sent from the same node that sent the AES key. After this, the initial communication is established and the future message from the sender will be decrypted using this shared AES key.

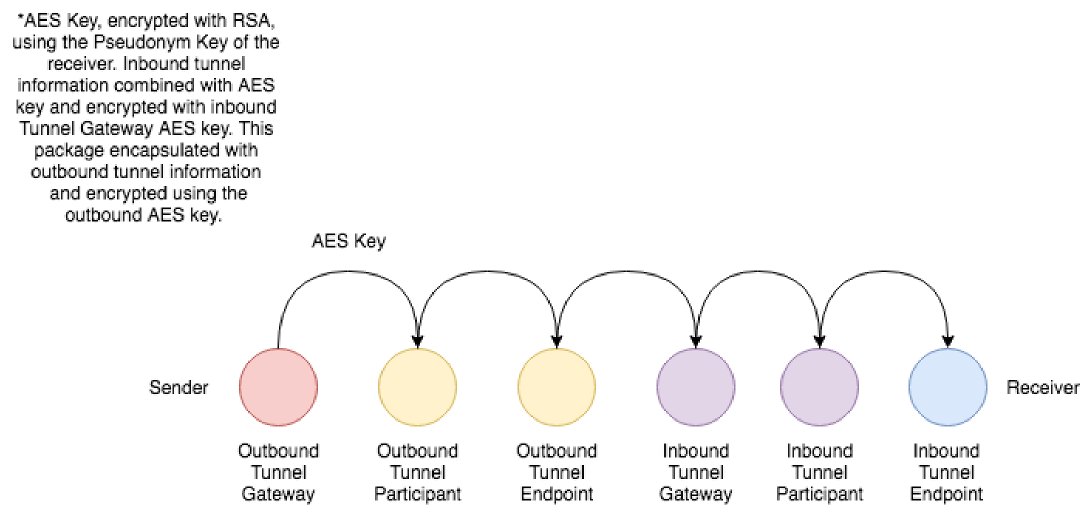

*AES Key, encrypted with RSA, using the Pseudonym Key of the receiver. Inbound tunnel information combined with AES key and encrypted with inbound Tunnel Gateway AES key. This package encapsulated with outbound tunnel information and encrypted using the outbound AES key.
AES Key
Sender
Receiver
Outbound Tunnel Gateway
Outbound Tunnel Participant
Outbound Tunnel Endpoint
Inbound Tunnel Gateway
Inbound Tunnel Participant
Inbound Tunnel Endpoint

Figure 3. AES Key Propagation

Message Send Phase:

This phase occurs after the AES key has been successfully sent to the receiver. The sender now uses the AES key it sent to the receiver to encrypt the message itself. Next, the sender packages the AES encrypted message similar to how the AES key was packaged in the Communication Initiation Phase, ie onion package. It sends the created onion package through the same outbound tunnel used in the Communication Initialization Phase. The package is passed through the outbound tunnel and is deposited into the inbound tunnel, which propagates the message to the receiver. The receiver, using the AES key it received earlier, decrypts the message and prints out the result.

Message Confirmation Phase:

Once the encrypted message is received by the receiver, it readies a confirmation reply to the sender. The original receiver creates a new outbound tunnel, then packages the reply using the standard format to send a message through a tunnel. The package is sent through the outbound tunnel of the original receiver, to be received by the inbound tunnel of the original sender. Once the reply is received by the original sender, the message is considered send and done.

For more visual information refer to *Figure 5 (last page)* for more information on how shared AES keys and messages is packaged into an “onion-like” layer and our message passing protocol.

Garlic Message:

A node can send multiple messages to two different nodes by aggregating the two onion packages to a single garlic package. The tunnel outbound endpoint is responsible for pelling the garlic package and rerouting each “onion” package to the inbound gateway of each intended receiver.

Failure Handling

Message Send Failure: This failure could be due to failing/exiting node, congestion in the network or encryption/key error. If a wrong pseudonym key is used, the receiver will not be able to decrypt the shared AES key message and hence will never be able to decrypt any messages from the sender. If some other node that is part of the tunnel, outbound or inbound, failed or exited, client will receive a timeout error.

Failing Node(fail stop): As a node keeps on updating its inbound tunnel every 10 seconds by calling the server to obtain a list of active nodes, our system will be able to recover from failing node eventually. Similarly, an outbound tunnel of a node will only be created with the current active nodes in the network every time a message is sent and deleted right after. Any recently joined nodes will be considered a new node.

V. Evaluation and Testing

Manual /End-to-end Testing:

To evaluate our system, we tested our code manually using client apps and debugging messages. In our tests, we made sure to cover a range of different features and scenarios. Features we tested included heartbeat RPC calls, tunnel creation, message sending, message encryptions (both RSA and AES), marshalling, and concurrency. We also simulated failure cases and tested failure handling, including failing and joining nodes. We repeated the tests for single and multiple nodes, and on different environments, including our local machines and Azure VMs. We also logged and tracked debug messages to visualize and test our system control flow.

1. Using *client.go*: this is used to send one message from one node to another. This requires the address of the sender, the node id and the pseudonym public key of the receiver.
2. Using *aggClient.go*: this is used to send messages from one node to two other nodes. Messages could be similar or different. This requires the address of the sender and two pairs of {node_id, pseudonym public key} of the receivers.

Both *client.go* and *aggClient.go* contact the node using RPC protocols via the node lib.

Unit Testing:

Automated scripts were created to test key generation, conversion of public keys to string and vice versa, and encryption/decryption of multi-layered encrypted messages using both AES keys and RSA keys.

VI. Limitations/Tradeoffs

For our system, we made a number of assumptions so that our team could focus on the security and failure handling aspects of the project. The limitations for our systems included:

1. Server will never die
2. The length of the inbound and outbound tunnels is set to a constant, which we chose to be 3
3. Created tunnels are not deleted but are overwritten instead as we did not manage to figure out the bug in our delete tunnel flooding message which causes our message passing protocol to fail.
4. A node cannot send multiple messages simultaneously, unless it is an aggregated message, which can be sent to multiple nodes (this is limited to 2 nodes in our system). It has to wait until the aggregated message is done sending before it can send another message.
    - For example, node 1 can send message A to receiver 1 and message B to receiver 2. Message A and message B need not be different. While these messages are being sent, no other messages can be send by this sender until the previous messages are done sending or terminated.
5. Theoretically, it is possible for our system to send messages from one node to a list of other nodes in the network. However due to time constraint, stress test has not been conducted for this feature.
6. We used a counter to keep track of a number of reply messages received but we did not differentiate which receiver the reply is coming from.

VII. GoVector and ShiViz

We added support for GoVector and ShiViz for our system to generate a diagram that provides a visual of how the data/control flow of our implementation of Garlic Routing. In Figure 4, it displays how the server and nodes interact. Initially, the nodes will send a message to the server, indicating that it would want to connect to the server. Once it is connected, the server will send a response back to the node indicating that it is now connected to the server.

A limitation we had was that we ran out of time, so we couldn’t provide that control flow between nodes when a message is sent.

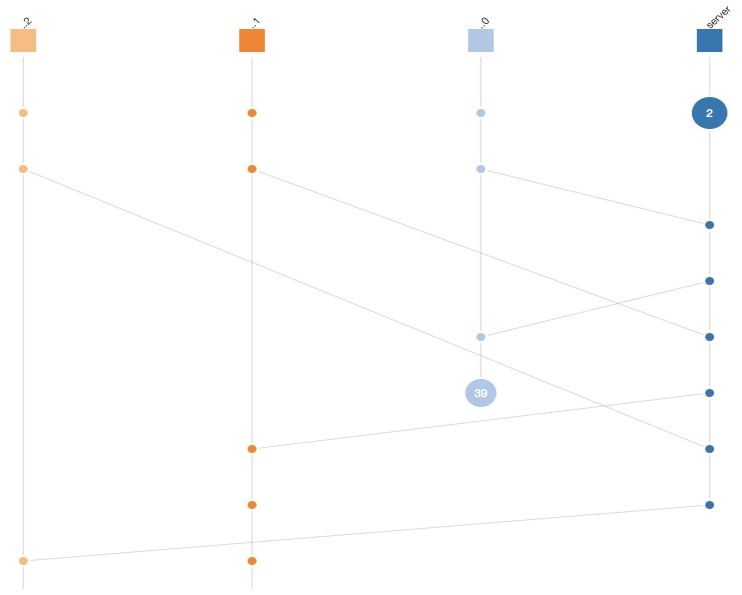


Figure 4. Shiviz diagram for communication of server to nodes.

VIII. <u>Discussion</u>

For most of the project, our team made steady progress and stayed on track with the proposed timeline. The team spent the first week and a half conducting research, discussing design and writing skeleton code, which helped immensely to prevent the creation of bugs and problems from the implementation process. Furthermore, the team met up 3 to 4 times every week to code together; each meeting lasted at least 3 hours. During the team meetings, teammates peer-coded and provided code reviews. This allowed everyone to stay updated with any issues and new additions to the project, and allowed a smooth integration of the different parts of the system.

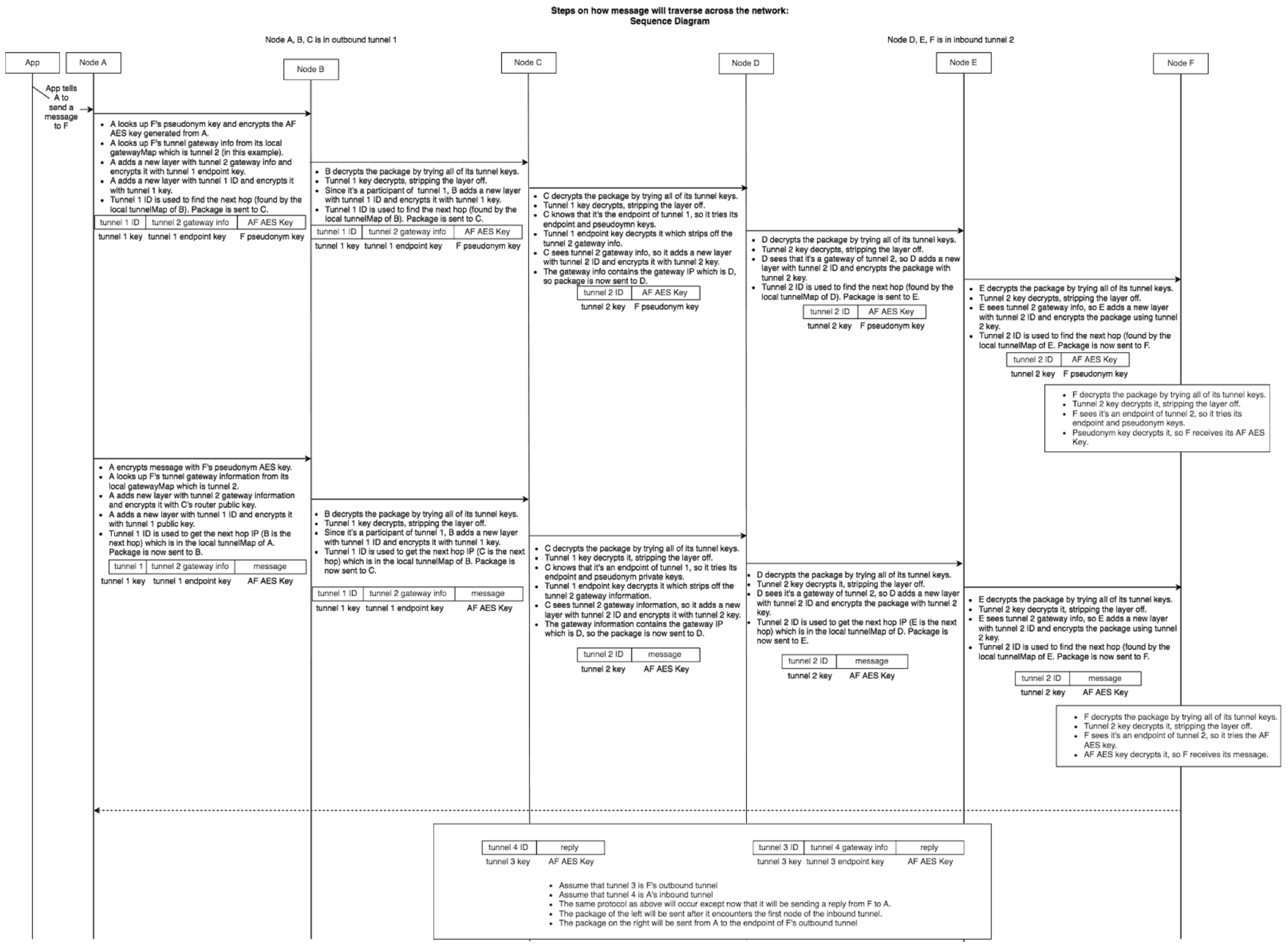
One issue we encountered, which set us off course was the assumption we made about RSA key encryptions. We did not check the size limit of RSA key encryptions, which was 256 bytes; thus, when we tried to double-encrypt our message structs, we were unable to. The double-encryption is critical to our system, as we intended to have multiple layers of encryption using RSA key for our “onion” package. To work around this problem, the TAs and us came up with two solutions: split the message structs to be encrypted into chunks, or look for another encryption method which allowed for bigger sizes. After some extensive research, our team decided to utilize a shared AES key in our message communication. The encryption issue required a lot of modification in the code and the design of our message passing protocol, causing us to fall behind our intended schedule. The encryption issue delayed our timeline, which prevented us from performing any form of stress testing of “garlic package” on our system.

IX. <u>Allocation of work</u>

Our team completed most of the work towards the project during team meetings. Sultan came up with the project idea, and did most of the research and knowledge transfer for the team on Garlic Routing. We worked together on the project design and proposal. For the implementation, we followed the designated timeline and work allocation we suggested in the proposal, and made any adjustments along the way. Jastine wrote the server code, and worked on implementing the node functionalities such as rpc and tcp connections, message passing protocol alongside Frances and Trent. Sultan worked with Tina on the tunnel implementations and flooding for the system. Sultan also worked on message aggregation/ “garlic” packages. Tina and Frances also worked on the extra credit features. For error handling, debugging and testing, the team worked together during the meetings throughout the progress of the project. The final report was written together as a group and edited by each member.

X. <u>References</u>

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


Steps on how message will traverse across the network:
Sequence Diagram
Node A, B, C is in outbound tunnel 1
Node D, E, F is in inbound tunnel 2
App
Node A
Node B
Node C
Node D
Node E
Node F
App tells A to send a message to F

- A looks up F's pseudonym key and encrypts the AF AES key generated from A.
- A looks up F's tunnel gateway info from its local gatewayMap which is tunnel 2 (in this example).
- A adds a new layer with tunnel 2 gateway info and encrypts it with tunnel 1 endpoint key.
- A adds a new layer with tunnel 1 ID and encrypts it with tunnel 1 key.
- Tunnel 1 ID is used to find the next hop (found by the local tunnelMap of B). Package is sent to C.

tunnel 1 ID	tunnel 2 gateway info	AF AES Key
tunnel 1 key	tunnel 1 endpoint key	F pseudonym key

- B decrypts the package by trying all of its tunnel keys.
- Tunnel 1 key decrypts, stripping the layer off.
- Since it's a participant of tunnel 1, B adds a new layer with tunnel 1 ID and encrypts it with tunnel 1 key.
- Tunnel 1 ID is used to find the next hop (found by the local tunnelMap of B). Package is sent to C.

tunnel 1 ID	tunnel 2 gateway info	AF AES Key
tunnel 1 key	tunnel 1 endpoint key	F pseudonym key

- C decrypts the package by trying all of its tunnel keys.
- Tunnel 1 key decrypts, stripping the layer off.
- C knows that it's the endpoint of tunnel 1, so it tries its endpoint and pseudoymn keys.
- Tunnel 1 endpoint key decrypts it which strips off the tunnel 2 gateway info.
- C sees tunnel 2 gateway info, so it adds a new layer with tunnel 2 ID and encrypts it with tunnel 2 key.
- The gateway info contains the gateway IP which is D, so package is now sent to D.

tunnel 2 ID	AF AES Key
tunnel 2 key	F pseudonym key

- D decrypts the package by trying all of its tunnel keys.
- Tunnel 2 key decrypts, stripping the layer off.
- D sees that it's a gateway of tunnel 2, so D adds a new layer with tunnel 2 ID and encrypts the package with tunnel 2 key.
- Tunnel 2 ID is used to find the next hop (found by the local tunnelMap of D). Package is sent to E.

tunnel 2 ID	AF AES Key
tunnel 2 key	F pseudonym key

- E decrypts the package by trying all of its tunnel keys.
- Tunnel 2 key decrypts, stripping the layer off.
- E sees tunnel 2 gateway info, so E adds a new layer with tunnel 2 ID and encrypts the package using tunnel 2 key.
- Tunnel 2 ID is used to find the next hop (found by the local tunnelMap of E. Package is now sent to F.

tunnel 2 ID	AF AES Key
tunnel 2 key	F pseudonym key

- F decrypts the package by trying all of its tunnel keys.
- Tunnel 2 key decrypts it, stripping the layer off.
- F sees it's an endpoint of tunnel 2, so it tries its endpoint and pseudonym keys.
- Pseudonym key decrypts it, so F receives its AF AES Key.

- A encrypts message with F's pseudonym AES key.
- A looks up F's tunnel gateway information from its local gatewayMap which is tunnel 2.
- A adds new layer with tunnel 2 gateway information and encrypts it with C's router public key.
- A adds a new layer with tunnel 1 ID and encrypts it with tunnel 1 public key.
- Tunnel 1 ID is used to get the next hop IP (B is the next hop) which is in the local tunnelMap of A. Package is now sent to B.

tunnel 1	tunnel 2 gateway info	message
tunnel 1 key	tunnel 1 endpoint key	AF AES Key

- B decrypts the package by trying all of its tunnel keys.
- Tunnel 1 key decrypts, stripping the layer off.
- Since it's a participant of tunnel 1, B adds a new layer with tunnel 1 ID and encrypts it with tunnel 1 key.
- Tunnel 1 ID is used to get the next hop IP (C is the next hop) which is in the local tunnelMap of B. Package is now sent to C.

tunnel 1 ID	tunnel 2 gateway info	message
tunnel 1 key	tunnel 1 endpoint key	AF AES Key

- C decrypts the package by trying all of its tunnel keys.
- Tunnel 1 key decrypts it, stripping the layer off.
- C knows that it's an endpoint of tunnel 1, so it tries its endpoint and pseudonym private keys.
- Tunnel 1 endpoint key decrypts it which strips off the tunnel 2 gateway information.
- C sees tunnel 2 gateway information, so it adds a new layer with tunnel 2 ID and encrypts it with tunnel 2 key.
- The gateway information contains the gateway IP which is D, so the package is now sent to D.

tunnel 2 ID	message
tunnel 2 key	AF AES Key

- D decrypts the package by trying all of its tunnel keys.
- Tunnel 2 key decrypts it, stripping the layer off.
- D sees it's a gateway of tunnel 2, so D adds a new layer with tunnel 2 ID and encrypts the package with tunnel 2 key.
- Tunnel 2 ID is used to get the next hop IP (E is the next hop) which is in the local tunnelMap of D. Package is now sent to E.

tunnel 2 ID	message
tunnel 2 key	AF AES Key

- E decrypts the package by trying all of its tunnel keys.
- Tunnel 2 key decrypts it, stripping the layer off.
- E sees tunnel 2 gateway info, so E adds a new layer with tunnel 2 ID and encrypts the package using tunnel 2 key.
- Tunnel 2 ID is used to find the next hop (found by the local tunnelMap of E. Package is now sent to F.

tunnel 2 ID	message
tunnel 2 key	AF AES Key

- F decrypts the package by trying all of its tunnel keys.
- Tunnel 2 key decrypts it, stripping the layer off.
- F sees it's an endpoint of tunnel 2, so it tries the AF AES key.
- AF AES key decrypts it, so F receives its message.

tunnel 4 ID	reply
tunnel 3 key	AF AES Key

tunnel 3 ID	tunnel 4 gateway info	reply
tunnel 3 key	tunnel 3 endpoint key	AF AES Key

- Assume that tunnel 3 is F's outbound tunnel
- Assume that tunnel 4 is A's inbound tunnel
- The same protocol as above will occur except now that it will be sending a reply from F to A.
- The package of the left will be sent after it encounters the first node of the inbound tunnel.
- The package on the right will be sent from A to the endpoint of F's outbound tunnel

Figure 5. Sequence Diagram of Message Passing Protocol