CPSC 416 Project 2 Report

A Prototype Anonymous Tor Network

Muzhi Ou (q4d0b) Zi Wen(Joyal) Li (a9h0b) Wei Chen (i5g5) Minxing Wang (v3d0b)

Introduction

The Tor anonymity network preserves users' privacy and keeps their online activity away from the prying eyes of advertisers, stalkers, hackers and even governments. Despite the attention that Tor has received worldwide, the technical questions surrounding it remain underexplored. Does Tor grant its users 100% anonymity? What are the major components of the Tor network and their roles? Are there any weaknesses and pitfalls of Tor? Aiming to gain an in-depth understanding of Tor and the degree of privacy protection it claims, we design and implement a basic version of Tor from scratch.

We start with a structural overview of our Tor model and its components. Then, the concept of onion routing is explained across components that apply this technique. The underlying technical structure and cryptography of Tor, and the access to the network, its relays, and exit nodes are elaborated on afterwards. Furthermore, we highlight the difficulties in integrating these modules into a working distributed system. In the end, we conclude the report by discussing the results of experiments and tests conducted on our final product and revisiting the unanswered questions.

High level design

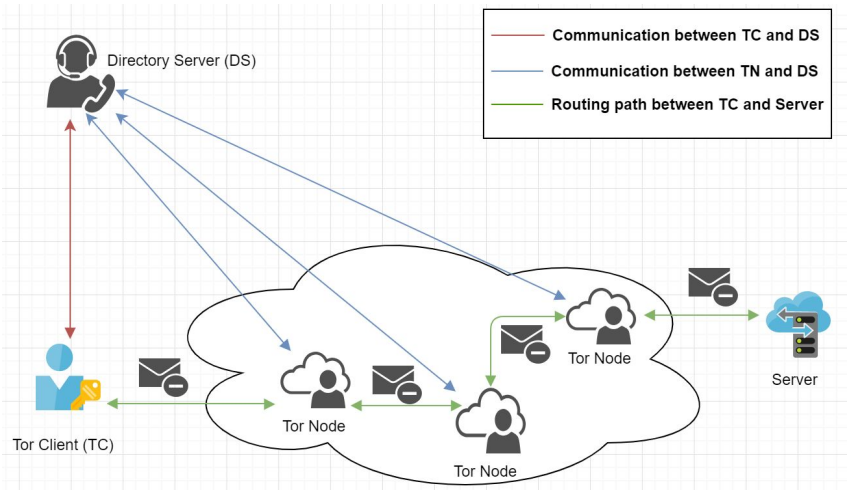
There are four basic types of nodes in our system.

- Tor Client (TC)
  End user that uses Tor's privacy-preserving service

- Directory Server (DS)
  Coordinator that provides TC with access to the Tor network and registers TNs that want to join

- Tor Node (TN)
  Packet router that hides TC from the rest of the world

- Destination Server
  Destination server that stores data



Destination Server

In our project, the destination server is a simple server that provides look up services from a simple in-memory key-value store. It expects to receive the key as a request from the client and provides the value associated with that key as response. If the requested key does not have an associated value in the key value store, the destination server returns an empty message. The destination server is mainly developed for demo purposes for this project. We assume that the clients of our tor network only access our destination server.

Directory Server (DS)

Directory Server handles a few important tasks in our tor network system:

1. Act as the entry point for new Tor node to join the network. New Tor nodes that want to join the network need to contact the directory server first.
2. Store information about all the existing Tor Nodes (TNs) that have joined the network.
3. Monitor all the Tor nodes in the network to keep updated information about which Tor nodes are still live in the network and which ones have died or left the network.
4. Select a number of TNs at random for Tor client to establish a circuit through the Tor network.

Note: a **circuit** is established by a few TNs that connects the Tor Client (TC) and the server. For example: TC ↔ T1 ↔ T2 ↔ T3 ↔ Server is an example of a circuit, where each pair of consecutive nodes are connected via TCP connections.

Tor Client (TC)

Tor clients (TCs) act as the users. Human users of our Tor network will use a Tor client to communicate with their desired destination server. Tor clients send requests containing keys (of the in-memory key-value store) to the destination server through the Tor network and expect to receive the associated values as the response. As described above tor clients will first contact the DS server for a set of tor nodes available for use. It will create the circuit using the Tor nodes provided by the DS server in arbitrary order.

After a circuit is setup, it encrypts the messages received from the Tor client into a layered data structure like an onion and sends the message into the Tor network through the circuit. The messages bounce through a few Tor nodes in the network and will eventually reach the server the original request was designated to. The server response goes through the same set of Tor nodes it came from and goes through the same multi-layer of encryption and reaches the Tor client. The Tor client decrypts the message layer by layer to retrieve the actual response from server.

Tor Node (TN)

Tor Nodes are the individual nodes that make up the circuit. Within one circuit, a Tor node only connects its immediate previous node in the circuit and its immediate following node in the circuit. Any individual tor node only knows the location/IP of its previous node and following node. It does not know any other Tor nodes in the circuit, where the original data came from, where is its final destination or what is the original data in the packet.

A Tor node only needs and is only given these 2 pieces of information (preceding node IP and following node IP) to 1. forward packet from preceding to the next node in the circuit, 2. forward packet back from following node to prior node. With this setup, the Tor nodes that build up a complete circuit can forward messages from Tor client to destination server, and message back from destination server to Tor client anonymously.

Exit Tor Node (Exit TN)

An exit node is just a regular tor node except that it is the very last node in the circuit. After this Tor node peels off the last layer off our onion packet, it gets the original data from the client and an IP to the destination server. The exit node will perform the final job of sending the client data to destination server. Notice that the content of client request message is still not visible to the exit node, because it is encrypted using destination server's public key.

Difficulties

Encrypting large data (onion messages) with asymmetric keys

We discovered that a asymmetrical key of 2048 bits can only encrypt payloads of up to around 220 bytes. However, the original client request payload encrypted using server's public is usually at least 800 bytes. We first tried to compress the message with gzip but this was not a good solution since we cannot guarantee the message will be compressed down to lower than 220 bytes after compression. Also, the size of the onion increases linearly with the number of Tor nodes we use, which means as we use more Tor nodes to set up the circuit, the compression will not help.

Finally we decided that we should break up the onion into 200 byte pieces and then encrypt each piece. Below is a step by step description of what we did:

Sender:

1. Marshal the onion message into an array of bytes
2. Break up the array of bytes into 200-byte pieces
3. For each of the 200-byte piece, encrypt it using the public key
4. After the encryption, we marshall the pieces back into a single array of bytes

5. Send the final array of bytes

Receiver

1. Receive an array of bytes
2. Unmarshal into an array of byte arrays
3. Decrypt every piece with the private key
4. Combine all the pieces into a single byte array
5. Unmarshal this byte array to reveal the original onion message

Sending data as strings vs raw bytes

We started our project by sending all TCP payloads as strings. We would marshall the original payload into a JSON string and concatenate a new line character at the end of the string. The benefit for doing this is that on the receiving end, we can just read the bytes until we reach the new line character. However, soon we realized that the encryption after the JSON marshalling could mess up our string casting. Sending casted string bytes and encryption don't work well together, because the new line character is represented as 0x00 in hex. There was no way to ensure that the encrypted bytes didn't have any 0x00 before the "true" new line character.

We decided to send message between nodes using raw bytes. Our initial approach was as follows:

1. from the sending node, write the byte array to the tcp connection.
2. On the receiving node, initialize a result byte array to store the entire payload received and a temporary byte buffer of size 1024 (we called it chunks) to keep reading data from TCP read
3. We keep reading from the TCP connection using the buffer and concatenate the read data to the result byte array until the size read from the tcp connection was less than the chunk size (1024).

This was based on the assumption that the tcp will keep receiving the full chunks i.e. 1024 bytes until the end of the byte stream (the last chunk can be any size between 0 to 1023 bytes, at this point we assume there's no more bytes coming). However, it turns out that it is possible for TCP connection to receive a chunk with size less than 1024 in the middle of the byte. For example if your entire payload size was 2049, we assumed it always arrives as chunks [1024 bytes, 1024 bytes, 1 bytes]. However, depending on the traffic the chunks could be anything in the form [x bytes, y bytes, ..., z bytes] as long as x + y + … + z = 2049.

To solve this problem, we developed a protocol explained below.

Our protocol to send payload size first in TCP communication

The third approach we took was a 2-stage protocol for sending data payload. For each message, we first send a fixed-length 4-byte integer indicating the size of the message we are sending, then we send the actual message to the TCP connection as a byte array as usual.

On the receiving side, we always read the 4-byte integer first to get the expected message size. Note that even for the 4-byte integer, we cannot just create a 4-byte length buffer to directly read from the TCP connection, because even the 4-byte integer can arrive in incomplete chunks. So we first read from the connection byte by byte until we have the first 4 bytes in hand. Now we proceed to read payload chunks from the TCP connection until the total bytes received equals to the expected size.

This approach was what we settled in the end for node communications. Our newly designed protocol works well for data

Providing system-wide ShiViz

To isolate GoVector logging from the rest of the system and to make the GoVec logging easily reusable, we inserted a middle layer to pack & unpack ShiViz bytes between the application layer (each node's own logic) and the networking layer (the functions that actually write bits onto the wire). This design allows the application layer to be unaware of ShiViz logging and the networking layer to only focus on sending and receiving bytes (it doesn't care what's the content in the bytes).

Changes

We have made one small change to our protocol details. Initially when the client receives a list of Tor nodes from the directory server, we take that list and create a circuit to which we send our message along. The original plan was to first create an initial message containing no payload and only the symmetric key. So the circuit will be setup and each tor node will receive its symmetric key for that route. After this we start sending actual onion payload into the circuit.

We have realized that the initial message is unnecessary because we can just create the circuit as the Tor nodes are forwarding their first payload onion. For the first onion payload, we just need to include the symmetric key as well so that the Tor node knows how to encrypt response data coming back.

Evaluation

We have spun up a total of 13 VMs to evaluate our system: 9 for Tor nodes, 1 for destination server, 1 for directory server, and 2 for Tor clients.

We have tested our system with shorter and longer circuits. We first have all the tor nodes (9 of them), and the directory and destination servers running. We configure the client to use only 3 Tor nodes to set up the circuit. The client sends a request string of length 4 (Most of our request strings are less than 10. But it can easily be configured to be longer). The initial unpeeled onion

(with all layers of encryption) is about 15 Kbytes in size, and the client was able to get the response back in about 1 second. We then manually shut down and later restart a few nodes, as long as there are still more than 3 nodes in the system, the client can send a request to the Tor network and get a response back in around 1 second. This experiment shows that our system can handle failures and “new joins” of nodes smoothly.

Now we pressure test our system. We have all the 9 Tor nodes on and configure the client to set up a circuit that has all the 9 nodes in it. We still send a request string of length 4. This results in an initial unpeeled onion (with all layers of encryption) of about 13 MegaBytes and took about 3 minutes to get a response back from the Tor network. Though a long time and a huge payload size, the system is still able to process the request correctly. This experiment shows that our Tor system can handle huge pressure payload data and that our implementation of the 2-stage protocol to transmit payload over a TCP connection functions correctly!

Demo Plan

Demonstrate normal operation of your system (no failures/joins) with at least 3 nodes.

- We will use a total of 13 nodes for the demo with the following breakdowns:
    - 1 directory server.
    - 2 tor clients.
    - 1 data server.
    - 9 tor nodes (we will start 3 nodes first).

- We will start the directory server and data server first, then start 3 tor nodes first to join the tor network and set the MaxNumNodes of the tor client to be 3.
- We will use 2 tor clients to send requests at the same time to demonstrate that our Tor nodes can handle multiple requests at the same time.
- We will then start all remaining 6 tor nodes.
- We will use 1 tor client to send requests to obtain the value from the server. For example:
    - go run client/client.go config/client.json GOOGL, we should receive a response of Google, and we can validate from the data server side that the ip of the request is not the same as the Tor client. We can try a few different key values such as AMD (Advanced Micro Devices Inc), AKAM (Akamai Technologies Inc) to get longer response strings.

Demonstrate system can survive at least 3 node failures

- Since we assume that our directory server and data server will never fail, our demonstration will mainly focus on the failure of tor nodes.

- We will take down 6 tor nodes first, wait for a few seconds and have the client to send a request again and make sure none of the 6 tor nodes are used in the circuit.
- We will further take down 2 tor nodes and have 1 tor node remaining and send a request again and make sure we are only using the 1 remaining tor node in the circuit.

Demonstrate system can join and utilize at least 3 new nodes

- We will restart up 3 tor nodes from the ones that we took down and have the client send a few requests and make sure some of the 3 new tor nodes are being used in a circuit.

Design Q/A

- We are happy to answer any questions from our amazing TAs and Ivan!

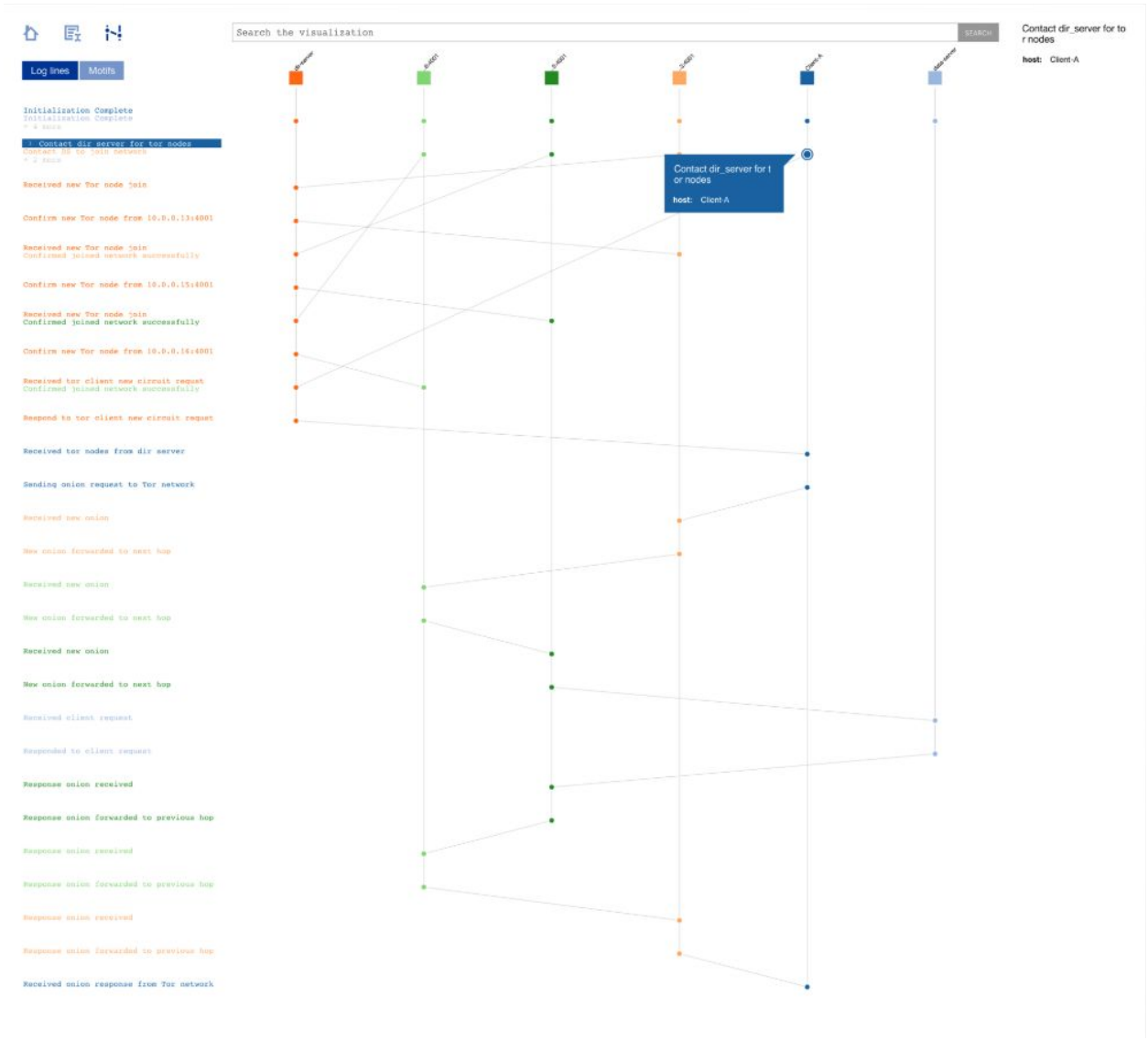
ShiViz

Below is a ShiViz diagram generated from our Tor network. For simplicity and not to make this diagram hard to read, we only included 3 Tor nodes, 1 Directory Server, 1 Data Server and 1 Tor Client for this ShiViz demo.

The 1st vertical line is the Directory Server, the 2nd to the 4th vertical lines are the 3 Tor nodes, the 5th vertical line is a Tor client, and the last vertical line is the Destination Server.

In the top half of the graph, we can see that the all 3 Tor nodes are making requests to the directory server to join the network, and the directory server responded back to each of them.

The client then makes a request to the Directory server to get a set of Tor nodes to form a circuit (this is the highlighted second event on the 5th vertical line). After the Directory Server responded to the client, the bottom half of the ShiViz diagram gives a nice illustration of how the client first send the onion message to the Tor node on the 4th vertical line, then forwarded to the Tor node on the 2nd vertical line, then forwarded to the Tor node on the 3rd vertical line, eventually reaching the Destination Server (the last vertical line), and then the response follows the same path to get back to client.



Conclusion

Our system works as desired and all tests return expected results. That is, Tor client is able to receive requested data without exposing its identity to the majority of Tor network and destination server, other than the entry Tor node and Directory Server. Besides, Tor client remains anonymous in the event of packet interception and/or loss. Furthermore, our one-time circuit and time-out mechanism form a robust defense against malicious surveillance and Tor node failure, as well as enable our system to recover from them quickly.

As part of future enhancements, we would like to add new features such as load balancing, malicious node detection and Tor node whitelisting/blacklisting, which will make the system more effective and secure.