Thor: The Harvard Onion Router

Dimitrije Pavlov and Cynthia Chen

CS 262 Final Project

Abstract

We introduce The Harvard Onion Router (Thor), a proof-of-concept model of Tor implemented as a distributed system. We replicate Tor functionality using a peer-to-peer distributed system of onion routers, a client that performs encrypted communication with the routers, and a directory server managing router information. The primary goal is to enable a client to exchange data with a server without revealing the client's IP address to the server or to any third parties that might be observing the network. To accomplish this anonymity of communication, we create a series of intermediate nodes that relay traffic, using encryption to prevent the nodes from seeing the data being forwarded and to prevent the source IP from being associated with the destination IP at any point during the communication.

1 Introduction

Anonymous communication systems enable a wide variety of practical applications, including secure instant messaging, incognito web browsing, and political activism and journalism in sensitive areas. Such systems originated with the development of Mix-Net [1], which introduced a technique based on public-key cryptography allowing for anonymous user communication in an electronic mail system without reliance on a universally trusted authority. The encrypted communication is achieved through a path composed of "mixes", where each mix in turn decrypts, delays, and re-orders messages before relaying them onward. Since then, several other analogous anonymous communications systems have been developed, such as Babel [2] and Mix-minion [3]; however, these systems suffer from the trade-off between latency and anonymity. These systems prioritize anonymity at the expense of latency and lag, rendering them inadequate for complex tasks like internet browsing.

Tor, also known as The Onion Router, is a software protocol that was developed in the 1990s by US Naval Research Laboratory employees Paul Syverson, Michael Reed, and David Goldschlag with the goal of protecting the identity of intelligence agents. Since then, a second-generation version of Tor was developed with latency low enough to enable users to anonymously browse and communicate over the Internet [4]. Tor prevents any third party from being able to associate the source IP (i.e. the IP address of a user) with the destination IP (i.e. the IP address of a service the user is communicating with), making it difficult to trace a user's browsing activity. While Tor has been infamously associated with illicit activities, Tor has found irreplaceable usage among whistleblowers, journalists, activists, users from Internet-censoring countries, and people who are simply concerned with their privacy. Overall, Tor has become a pillar of secure online communication.

In Tor, internet traffic is first encrypted and then routed through a Tor circuit consisting of three different types of intermediate nodes ("onion routers"): entry nodes, middle nodes, and exit nodes. The entry node knows the source IP, the exit node knows the destination IP, and the middle node knows the addresses of the entry and exit nodes. At no point in the network are the source IP and destination IP both visible in clear text, which provides strong anonymity guarantees. The traffic is encrypted in layers (hence the "onion" name), which ensures forward secrecy– if any node gets compromised, the user's anonymity is still guaranteed. The encryption applied at each node is based on public-key cryptography and Diffie-Hellman key exchange, allowing the traffic to be decrypted only by the next node in the chain.

The primary motivation for our work stems from the distributed model of Tor. In particular, we are interested in the encrypted peer-to-peer communication that occurs between a client and the circuit of onion routers, and replicating a proof-of-concept model of these interactions. Our objectives are as follows:

(1) Implement Thor, a proof-of-concept model of Tor.

(2) Use a peer-to-peer distributed system to enable encrypted communication between the client and respective onion routers.

(3) Apply our system to a real-world application – connecting to ChatGPT in Italy, where it is currently blocked (as of May 5, 2023).

Section 2 outlines the design of our system. Section 3 provides implementation details. Section 4 provides a brief evaluation and testing of Thor, along with an application. Section 5 discusses some of the lessons we learned during this project. Section 6 concludes the paper.

2 System Design

Section 2.1 outlines the system architecture. Section 2.2 describes the Thor wire protocol. Section 2.3 discusses the authentication and encryption in Thor.

2.1 Overview of System Architecture

We first describe an overview of our system architecture (see Figure 1) and the various entities involved in setting up an anonymous connection between a client and a website.

The directory server (DS) is the first point of communication for the client. It has a well-known address and a well-known long-term identity key (see Section 3 for details on the cryptography that we use). The client trusts the directory server to maintain a dynamic list of onion routers (OR) that participate in the Thor network. The client initiates the communication by sending a request to the DS, requesting the list of all ORs. The DS responds with a list of ORs' IP addresses and ORs' long-term identity keys. The communication from the DS to the client is signed using the well-known DS identity key, so the client can verify that the DS is legitimate.

The client arbitrarily chooses three ORs to establish an OR circuit. Encrypted communication exists between the client and its OR circuit via a layered messaging system. Every message sent from the client through the circuit is forwarded along the path OR 1 – OR 2 – OR 3 (see Figure 1). The last OR in the circuit establishes a TCP connection to a given <hostname, port> pair. Once the connection is established, the circuit is ready to relay data from the client to the destination. In principle, Thor supports any TCP application; in practice, the HTTP(S) protocol is most commonly used.

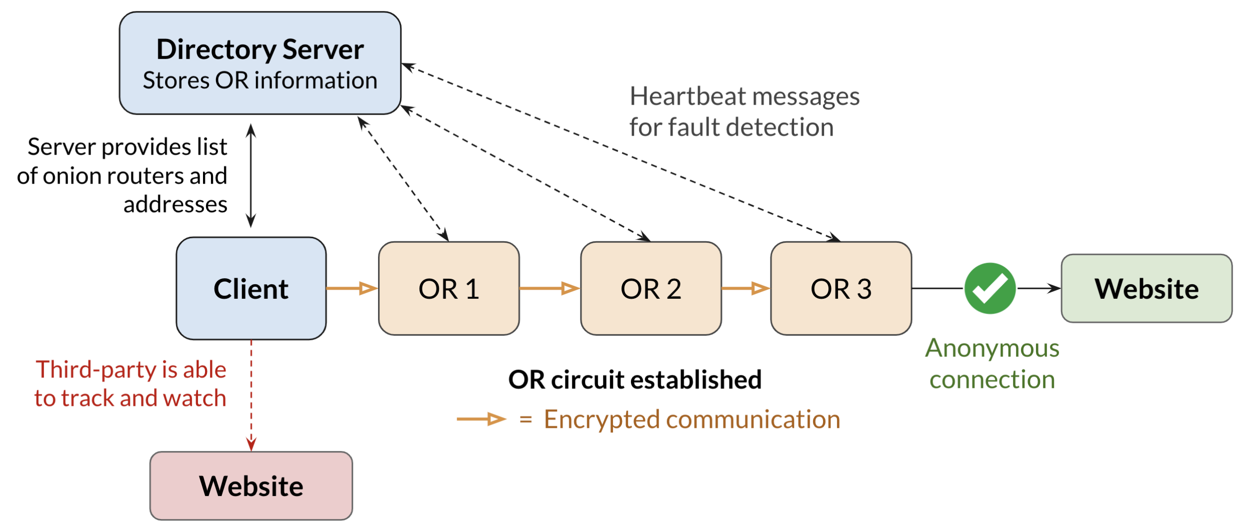
Directory Server
Stores OR information
Heartbeat messages for fault detection
Server provides list of onion routers and addresses
Client
OR 1
OR 2
OR 3
Website
Anonymous connection
Third-party is able to track and watch
OR circuit established
= Encrypted communication
Website

Figure 1: An overview of the Thor system architecture, including the client, directory server, and the circuit of three onion routers.

For an OR to join the Thor network, it establishes communication with the well-known DS and sends a request to join. The request includes a challenge (a randomly generated nonce), and the OR's long-term identity key p_k. The DS responds by signing the OR's challenge (proving that it holds the private key for the well-known DS public key), and by sending a new challenge, which the OR has to sign in order to prove that it holds the private key s_k that corresponds to p_k. Our join protocol ensures that both the OR and the DS authenticate each other. After successful authentication, the server includes the OR in its dynamic list of ORs. In principle, the DS could choose to set arbitrary criteria for joining the Thor network. For instance, it might be beneficial that the DS administrator verifies through other channels that the OR is operated by a non-malicious user. In our current proof-of-concept implementation, the DS allows anybody to join the network as an OR.

Once an OR joins the network, it maintains heartbeat messages with the DS. This way, the DS is able to detect fail-stop failures of any ORs and update its list of ORs when this happens. An individual OR itself is not fault-tolerant – in theory, they are entities run by volunteers. However, the whole network is fail-stop fault-tolerant – if an OR fails, the network will simply re-route traffic through other ORs. For n ORs, the network can, in principle, tolerate $n - 1$ failures (a circuit can only have 1 OR). However, a small number of ORs weakens the security guarantees of the network. We assume that some of the ORs might be compromised by passive attackers. Communication is secure as long as there is at least 1 OR in the circuit that is not compromised. If the number of ORs becomes really small, the chance of all ORs in a circuit being compromised increases.

In our current implementation, the DS is a single point of failure. However, DS can easily be made fail-stop fault-tolerant by a simple replication scheme; we leave this for future work.

2.2 Messages and Communication

We define our custom wire protocol for all messages exchanged between Thor participants. Here, we discuss the three types of control cells—`Create`, `Created`, and `Destroy`—and the relay cells `Extend`, `Extended`, and `Begin`.

A `Create` cell is sent by the client to an OR when they want to establish a circuit. This cell includes the material for the negotiation of a shared secret key (see Section 2.3 for details). The OR responds back with a `Created` cell, which helps the client authenticate the OR against the information received from the DS. The `Destroy` cell is used by a client when it wants to destroy the entire circuit, and it can also be used in the case of an error (for security, the Thor network preemptively tears down a circuit in case of an error). When an OR receives a `Destroy` cell, it destroys the state it associates with the circuit and also forwards a `Destroy` cell to the next OR to do the same, and so on, recursively destroying the entire circuit.

The relay cells `Extend` and `Extended` are used by the client when establishing the onion router circuit and adding another OR to the circuit. A client sends `Extend` to the current last OR in a circuit when it wants to add another OR after it. An `Extend` cell includes the IP address of the new OR (which the client obtains from the DS), material for key negotiation with the next OR, and a digest. The digest, which is the hash of the whole packet, is a signal that indicates who the message is meant for. ORs that receive a relay cell will check the digest after decrypting the cell ("peeling off an onion layer"). If the digest matches, the message is intended for the OR, and it interprets it and acts accordingly. Otherwise, the message is passed along the circuit. The same happens when the message is passed back to the client, except that each OR encrypts the message ("adds an onion layer"). For an `Extend` cell where the digest correctly matches, the OR will re-package the key negotiation material in a `Create` cell and forward this to the next OR based on the IP address. The `Extended` cell is essentially a response back to the client from the last OR in the circuit and it contains essentially the same information as `Created`, but encrypted with the session key negotiated earlier. As such, the workflow used by the client to establish a new OR is `Extend` → `Create` → `Created` → `Extended`, a process that is visualized in Figure 2. Once the circuit is established, the `Begin` relay cell is sent by the client to OR3 to establish a TCP connection to the <hostname, port> pair.

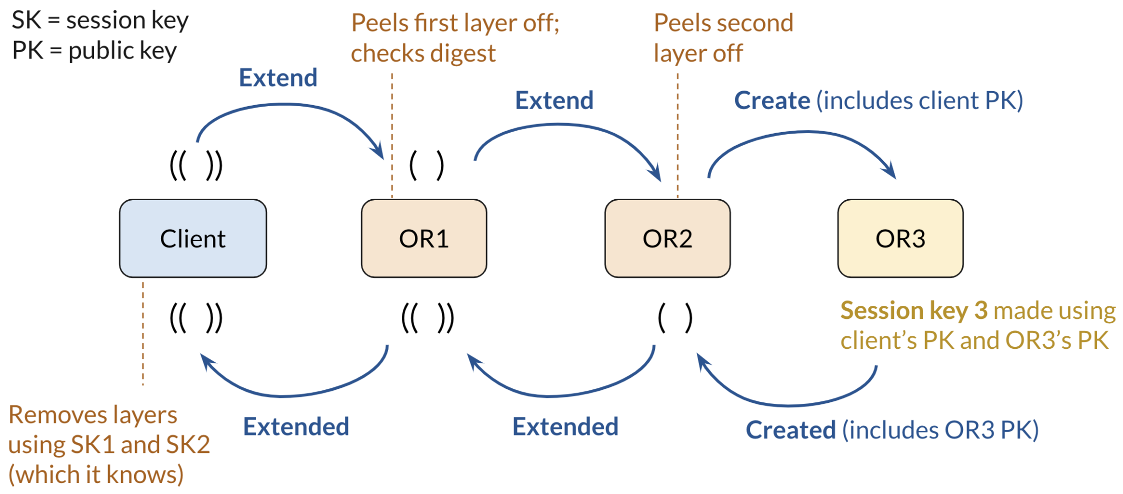
SK = session key
PK = public key
Peels first layer off; checks digest
Peels second layer off
Extend
Extend
Create (includes client PK)
(( ))
( )
Client
OR1
OR2
OR3
(( ))
(( ))
( )
Session key 3 made using client's PK and OR3's PK
Removes layers using SK1 and SK2 (which it knows)
Extended
Extended
Created (includes OR3 PK)

Figure 2: A visual depiction of the layered encrypted messaging that occurs between the client and the first two onion routers when establishing the third onion router in a circuit.

Figure 2 depicts the example of establishing the third onion router in a circuit, and the layered messaging that occurs between the client and the first two onion routers to accomplish this. As shown, the client first sends an `Extend` message with two layers to OR1, which peels the first layer off and checks the digest, realizing the message is not intended for itself. OR1 then passes the `Extend` message (with only one layer now) to OR2, which peels the remaining layer off, checks the digest, and realizes the `Extend` is meant for itself. It then passes a `Create` cell which includes the client's public key to OR3, where session key 3 is established using the public keys of the client and OR3. A similar train of response communication occurs from OR3 back to the client via `Created` and `Extended` cell types, and the client finally removes the two-layered message using the secret keys of OR1 and OR2 which it knows.

2.3 Authentication and Encryption

Thor uses a mix of asymmetric and symmetric cryptography. Each DS has a long-term private/public identity keypair s_k^{DS}, and p_k^{DS}; these are Curve25519 keys. The public key p_k^{DS} is well-known and form the root for our chain of trust. When the DS sends the list of ORs to the client, it signs the list using s_k^{DS} and th Ed25519 signature scheme. The client is able to verify the signature using p_k^{DS}. This way, trust is extended to p_k^{OR}, which are public Curve25519 keys belonging to ORs that the client received from the DS.

When forming a circuit, the client generates an ephemeral private/public keypair s_k^{c}, and p_k^{c} and send the public key p_k^{c} to an OR. Upon receiving the message, the OR generates its own ephemeral private/public keypair $s_k^{\mathrm{OR,\,e}}$, and $p_k^{\mathrm{OR,\,e}}$. The OR uses p_k^{c} and $s_k^{\mathrm{OR,\,e}}$ generate a session key s_{s}. Mathematically, the session key is a BLAKE2b hash of the Curve25519 shared secret, that is stretched to 32 bytes to use for symmetric encryption. The OR sends back $p_k^{\mathrm{OR,\,e}}$ so that the client can compute s_{s} as well, a BLAKE2b hash of s_{s} for key agreement verification[1], and a signature over the entire message made using s_k^{OR} (the long-term OR identity key). The established session key s_{s} is subsequently used to symmetrically encrypt all relay messages using the XSalsa20 streaming cipher for encryption and Poly1305 MAC for authentication. This way of creating a circuit helps us establish a few properties:

- Because all encryption keys are ephemeral, we achieve perfect forward secrecy – if the session key for a particular session is compromised, previous and future sessions are not compromised.
- Because of the Curve25519 key exchange, even if a third party eavesdropped on the key exchange, they would not be able to derive the session key.
- Using the long-term OR public identity key received from the DS, the client is able to verify (using the signature) that the circuit was in fact formed with an OR that was verified by the DS. The private key is needed to create the signature, but only the public key is needed to verify the signature, so the client can easily verify the signature.

A careful reader might notice that these properties are also guaranteed by Transport Layer Security (TLS) protocol. This is correct, but the TLS guarantees these properties only for point-to-point communication. The Thor protocol is relaying messages with layered encryption across multiple ORs where an encryption layer is removed at each OR. This is not something that is guaranteed by TLS, so we have to implement it at the application level.

[1]The hash of the session key helps confirm that the client and OR agreed on the same session key without having to send the actual session key over the network.

3 Implementation

The Thor system is implemented in Python using its associated libraries. Our codebase, which includes implementations of the client, onion routers, directory server, encrypted communication protocol, an example application, and unit tests is available online [5].

In the Thor communication protocol, peer-to-peer communication is implemented through cells, which essentially act like messages. Every cell consists of a header and a payload (cell body). The cell header includes information including the version of the protocol, the type of cell, a circuit ID that specifies which circuit the cell refers to, and the body length. Unlike the original Tor paper, which implements each cell in a fixed 512-byte size, the body length of cells in the Thor protocol is variable. Though this makes implementation easier, a potential disadvantage of this is that an outside attacker could potentially tell the type of message it is by observing the length of the cell. These attacks, known as traffic correlation attacks, are weaker if lengths are different, but if there are enough users it becomes harder to carry out.

The cell body contains varying information depending on what command is being executed. Mimicking the original Tor protocol, we implement two main types of messages: control cells and relay cells. Control cells are interpreted by the node that receives them, while relay cells carry end-to-end stream data. We implement three types of control cells: `Create`, `Created`, and `Destroy`. We also include several types of Relay cells: `RelayData`, `RelayBegin`, `RelayEnd`, `RelayTeardown`, `RelayConnected`, `RelayExtend`, and `RelayExtended`. Together, all of these cell types form the basis for the creation of an OR circuit and also the communication that ensues between nodes after a circuit is established.

Since there can be multiple circuits going through two ORs, we implement a circuit ID to keep track of which circuit is being used. We randomly generate circuit IDs as 16-byte integers so there is a very low chance of collision. Each router knows the same circuit by a different circuit ID, and there is a circuit state dictionary where the key is the circuit ID and the value is the general state of the circuit. The state of a circuit state is the information about a circuit that an OR would need to know, such as the two IP addresses for incoming/outgoing connections.

We also implement a directory server, a trusted entity that keeps track of all ORs. The interactions between the directory server and the client are implemented in a traditional distributed fashion with serialized messages sent between them. Finally, we implement a few additional cell types (on top of the control/relay cell types) that are used in establishing the heartbeat messaging between the directory server and the onion routers and requests from onion routers to join.

Encryption is implemented using the PyNaCl [6], a Python binding to the libsodium library [7]. We have several inputs to encryption, including a nonce, session key, and the data/message to be encrypted, which is normally the contents of a cell. The inputs to decryption are the same, and the nonce has to be the same as the one used for encryption. For security reasons, the same nonce must never be reused with the same secret key. In our system, we generate random nonces, but this is critically included as part of the message. Note that in the Tor implementation, a counter nonce is used; however, keeping track of this counter in a distributed fashion and guaranteeing accuracy is difficult for the scope of our project, so we opted for a randomly generated nonce. We send the nonce as part of the encrypted message so that the next entity can decrypt the message. Every OR in the chain of communication peels off one nonce/layer of encryption.

4 Evaluation and Applications

Section 4.1 describes the testing we performed for Thor. Section 4.2 describes an application of Thor to evading censorship.

4.1 Tests

We implement unit tests for the Thor protocol, which are included in the code repository [5]. Critically, we make sure that the implementation of the Thor wire protocol can correctly serialize and deserialize messages sent over the wire, that computation of the digest is implemented correctly, and that adding and removing onion layers works as intended.

We also perform testing of our client and server implementations, by making sure they can successfully establish a circuit and relay traffic successfully. Both our client and server implementation log detailed output.

Finally, we test our heartbeat implementation and find that the DS is able to successfully detect an OR crash. Subsequent OR lists sent to the client do not include ORs that failed.

4.2 Accessing ChatGPT from Italy

On March 31, 2023, Italy blocked access to ChatGPT for users located in Italy over data privacy concerns[2] [8]. We use this opportunity to demonstrate how Thor can be used to evade Internet censorship based on geographical location (note that this is just one of many applications of Thor). We also emphasize for transparency that Italy's block of ChatGPT is different from Internet Service Provider (ISP)-level blocks typically work in countries with Internet censorship. In the case of Italy and ChatGPT, OpenAI customized their webpage and API to serve different data to users located in Italy. In countries with heavier Internet censorship, ISPs typically perform IP packet inspection and drop packets destined for blocked websites. Thor is an applicable method of evasion in both cases, since it prevents association of source IP and destination IP. The difference is that, in case of Italy, we hide the source IP from OpenAI; in countries with heavier Internet censorship, we would typically hide the destination IP from an ISP.

To demonstrate that Thor works, we set up the following remote machines using AWS EC2:

- A client located in Milan, Italy.
- An OR located in London, UK.
- An OR located in Mumbai, India.
- An OR located in Sydney, Australia.
- A DS located in North Virginia, USA.

Trying to access chat.openai.com directly from Italy returns an error explaining that ChatGPT is restricted in Italy. If we use the Thor network, the client produces the following log output:

[2]The access to ChatGPT was restored in Italy on April 28, 2023. At the time of our project demo on April 26, 2023, ChatGPT was still banned, so we will proceed in the paper as if ChatGPT were still banned.

```
Connecting to DS at 35.175.188.11 in US
Received 3 OR addresses from DS
DS signature verified

Extending the circuit to OR 1 at 13.211.81.243 in AU
OR signature verified
Established a session key with OR 1

Extending the circuit to OR 2 at 18.170.42.48 in GB
OR signature verified
Established a session key with OR 2

Extending the circuit to OR 3 at 3.109.158.220 in IN
OR signature verified
Established a session key with OR 3

Opening a TCP connection to api.openai.com:443 through the circuit
Successfully connected to api.openai.com:443
```

We can see that our traffic is successfully relayed from Italy to Australia, then to UK, then to India, and finaly to OpenAI. Note that the order of ORs was randomly chosen by the client. Here is the log output of the OR in the UK (the intermediate OR):

```
Received a request from 13.211.81.243 [AU]: Create
Established a session key, sending Created to 13.211.81.243 [AU]

Received a request from 13.211.81.243 [AU]: Relay
Removed an onion layer, connecting to next OR at 3.109.158.220 [IN]

Received a request from 3.109.158.220 [IN]: Created
Added an onion layer, sending Relay to 13.211.81.243 [AU]

Received a request from 13.211.81.243 [AU]: Relay
Removed an onion layer, forwarding Relay to 3.109.158.220 [IN]

Received a request from 3.109.158.220 [IN]: Relay
Added an onion layer, forwarding Relay to 13.211.81.243 [AU]

Received a request from 13.211.81.243 [AU]: Relay
Removed an onion layer, forwarding Relay to 3.109.158.220 [IN]

Received a request from 3.109.158.220 [IN]: Relay
Added an onion layer, forwarding Relay to 13.211.81.243 [AU]
```

```
Received a request from 13.211.81.243 [AU]: Destroy
Destroyed the circuit, forwarding Destroy to 3.109.158.220 [IN]
```

This illustrates that this OR doesn't know who the client is, and doesn't know what final destination the client is tryint to connect to; it only knows the addresses of the neighboring two ORs in the circuit.

5 Lessons Learned

One thing that our system is lacking is more robust fault tolerance / resilience to bad participants. The current is assumption is that all participants will behave according to the Thor protocol, and that nobody will send malformed or malicious messages. In the real world, this assumption is obviously false. We were able to see first-hand how much harder implementing the protocol becomes if we drop the assumption that participants are nice. We have to handle every single edge case and possible client input. Even worse, because Thor is distributed, ORs can be malicious as well, so we can't assume that ORs behave nicely either. This just makes things a lot harder. In practice, we assume that a proper implementation would take much much longer because of this.

Another thing that was surprisingly hard on the OR side was worrying about race conditions. We have multiple incoming connections, and we have mutliple outgoing connections, and all of these can do anything at any point to the shared state. We tried to prevent the system from ever entering an inconsistent state to the best of our ability, but we admit the possibility of errors. In practice, we assume that the server state would probably be redesigned to account for a multi-threaded server, in a way that would make synchronization easier.

Worrying about the uniqueness of circuit ID or nonces in encryption was also really hard. The real Tor protocol makes use of distributed counters for both encryption circuit IDs and encryption nonces. While this is a smart solution that allows the messages to be more compact and always of fixed size, we thought this would be outside the scope of our project. We recalled the lesson from Prof. Waldo about UUIDs, and we just randomly generated these quantities. Of course, this does not guard against malicious users, but is the next best thing to minimize distributed state.

Implementing heartbeats between DS and ORs was not something that initially occurred to us. The idea came when we were doing some testing and one of the ORs actually crashed (the SSH connection to an AWS instance crashed and the OR process got terminated). This is when we realized that having the DS send stale lists of ORs to clients is probably a bad idea, so we came up with a heartbeat protocol.

The latency of Thor is not great. To measure it, we set up a simple HTTP server in the US and time how much it takes to send a request and receive a response. We measure that a connection from Italy to the US takes about 0.5 seconds, pretty consistently. With Thor and ORs in the UK, India, and Australia, the latency is not only much higher, but it's also variable – we measured between 2.7 and 3.9 seconds. This is much higher, and could very well be prohibitive for some applications. Unfortunately, getting around this is hard, and this is also something that the real Tor protocol suffers from. The traffic is intentionally bounced randomly, and any attempt to optimize for the path would open the possibility of traffic correlation attacks. Our thoughts are that the latency is probably too high to casually browse the web, but is more than acceptable for use cases such as evading censorship, whistleblowing, and activism.

6 Conclusion

In this project, we present The Harvard Onion Router (Thor), a distributed systems implementation of a proof-of-concept model of the original Tor system. Thor is able to successfully set up an anonymous connection between a client and a destination host through an encrypted circuit of onion routers. Some future directions for this project include improving latency and performance by compressing the amount of information we send as well as improving security guarantees by eliminating the variable-length messages. Furthermore, though we implement fault detection for onion routers that go down, future work could work on making our system fully fault-tolerant, as the directory server is currently a single point of failure.

Working on this project was very rewarding for us, as it gave us a glimpse into modern anonymous communication systems and the elegant model of encryption layers employed by Tor. Along the way, we made several design decisions relating to how we wanted various entities to communicate with one another, drawing upon the knowledge we learned from other design projects in CS 262.

References

[1] D. L. Chaum. Untraceable electronic mail, return addresses, and digital pseudonyms. *Communications of the ACM*, 24(2):84–90, 1981.

[2] C. Gulcu and G. Tsudik. Mixing e-mail with babel. In *Proceedings of Internet Society Symposium on Network and Distributed Systems Security*, pages 2–16. IEEE, 1996.

[3] G. Danezis, R. Dingledine, and N. Mathewson. Mixminion: Design of a type iii anonymous remailer protocol. In *2003 Symposium on Security and Privacy, 2003.*, pages 2–15. IEEE, 2003.

[4] R. Dingledine, N. Mathewson, and P. Syverson. Tor: The second-generation onion router. Technical report, Naval Research Lab Washington DC, 2004.

[5] Thor. Retrieved from https://github.com/cynthia9chen/thor262 on May 5, 2023.

[6] PyNaCl. Retrieved from https://pynacl.readthedocs.io/en/latest/ on May 5, 2023.

[7] libsodium. Retrieved from https://doc.libsodium.org/ on May 5, 2023.

[8] ChatGPT banned in Italy over privacy concerns. Retrieved from https://www.bbc.com/news/technology-65139406 on May 5, 2023.