Tor@CORE Final Report

Advanced Real-World Data Networks Course Project

Zhuoyu Ji
Carnegie Mellon University
zhuoyuj@andrew.cmu.edu

Rong Jin
Carnegie Mellon University
rongj@andrew.cmu.edu

Haoyang Wang
Carnegie Mellon University
haoyang4@andrew.cmu.edu

Peiyang Yu
Carnegie Mellon University
peiyangy@andrew.cmu.edu

ABSTRACT

Our team is interested in Tor, which is a network service that uses the onion routing protocol to provide secure and anonymous communication. The team decides to build a naive Tor network model inside the CORE emulator and then test its performance. During project development, the team studied several existing custom implementations of Tor including TinyTor and Torpy, which provide valuable takeaways in core logic and design details. With those help, the team successfully implemented three major components of Tor network: the directory authorities, the Tor clients, and the Tor relays. Though each component involves some simplifications, together they could achieve the basic functionality of the real world Tor network and model its behavior inside CORE. Then, the team tested the performance by comparing it to direct connection and adjusting the relay number. As a result, the team justified that Tor functionality is slow and a 3-Node circuit is optimal for practical usage. In general, the team focuses on reproducing the Tor network architecture and its performance instead of security. Therefore, most simplifications are related to security concerns. If the team has more knowledge in security and cryptography, it may further improve the project to make the custom Tor model and its testing scenario more realistic.

1 BACKGROUND

Tor, also known as The Onion Routing, was first developed by The United States Naval Research Laboratory in the mid-1990s. Its initial usage was to project U.S. intelligence communications online. But nowadays, Tor has been widely used by the public, aiming to protect the personal privacy of its users, as well as their freedom and ability to conduct confidential communication, by keeping their Internet activities untrackable. Due to this feature, Tor has been an iconic tool in some illegal fields online, such as the "Dark Web".

To realize the primary goal of Tor, which is to provide secure and anonymous communication, Tor network uses a circuit as a communication link, which consists of a guard relay, an exit relay, and multiple middle relays. Those relays connect to form the circuit. Every time a client tries to communicate with a server, instead of a direct connection, it uses this circuit to transmit messages.

Besides, the original message (plaintext) was encrypted multiple times. At each relay, it will decrypt/encrypt the received message and send it to the next relay. This process is known as peeling off or wrapping up messages, with the analogy to an onion. This special structure gives Tor anonymity, as each node only knows its previous and next nodes' information; it also has no idea where it is in the entire circuit. So, even if one of the nodes has been attacked, the attacker has no way to get full information, and the message encrypted for unknown times is also very difficult to decode.

However, like any technology, Tor is not 100 % secure, and attackers can still compromise Tor's security. In 2014, a research team from Carnegie Mellon University gained control of enough servers in the Tor network to observe the relays on both ends of the Tor circuit and compare the traffic timing, volume, and other unique characteristics to identify which other Tor relays were part of which circuits. By putting the entire circuit together, the researchers were able to see the IP address of the user on the first relay and the final destination of their web traffic on the last relay, allowing them to match users to their online activity[2]

2 PROJECT OVERVIEW

Our team members only knew briefly about the concept of onion routing before this project. Thanks to the analogy to an onion, the main concept of onion routing is intuitive and pretty easy to understand. Our team is interested in Tor because it develops such a straightforward concept into a very sophisticated and widely used system in the real world. After going through related research in Tor, we realized that: as Tor is a tool for secure and anonymous network communication, security in the real world environment is the main focus of its actual design and those related researches. However, this is not what we find most interesting. Instead, we want to explore the functional implementation of Tor and focus on its performance without considering too much about security or realistic restrictions. Based on those motivations, we came up with the original plan of this project: build a reliable Tor testbed inside CORE that functions like real world Tor as much as possible, and then analyze its performance. As to specific performance metrics, we want to investigate how efficient our Tor is and how changing parameters like relay setup and relay selection method would affect its performance.

To start this project, we need to go beyond the basic knowledge to deeply understand the Tor implementation so that we could build our own version. Therefore, we first tried to learn from the official implementation and corresponding documentation. Nevertheless, the official resources are mostly vague and they could hardly help us since the developers put major efforts into making Tor reliable and secure to use. So, it's pretty difficult for us to filter out the core logic we need. Thus, we gave up very early in this direction and shifted to examine existing implementations as those will be introduced below.

At the same time, we were also building our Tor model inside CORE part by part. After the most basic parts were done, we tried to incorporate those existing implementations we have learned. But, it turned out to be harder than we thought. Firstly, those existing works are all partial implementations of the entire system and they try to interact with the real world Tor; secondly, most existing works have some unfinished part or internal issues (limits and bugs). Thus, we could not easily use them in our model. However, learning from those works is still very meaningful as we learned some valuable design details. And that knowledge helped us a lot in improving our Tor model.

Overall, we were too optimistic at the beginning regarding our original plan. Building Tor inside CORE from scratch is more difficult than we thought and the resource we could really make use of was very limited. As a result, our Tor model is more simplified than we wish it could be. But we believe that we have implemented the major functionality of Tor. And we will go over what we have accomplished in later sections.

3 RELATED WORKS

During the project, we did some research online on the existing community's attempts to implement Tor network. Based on the open-source implementations we found, we focused on two. One was TinyTor[3], and the other was Torpy[4]. Here we give an overview of both implementations, discuss what we gained from them and how they gave insights into our project.

TinyTor is used to communicate with onion services in the Tor network. It acts as a client and tries to be as small as possible. With its aim of being as small as possible, all its implementations are in a single Python file. The components include the directory authority, the onion router used in a circuit, the hardcoded consensus provider, the relay cell that is the format of packets going through circuits, the circuit formation, and encryption-related specifications.

We found TinyTor as a good "quick start" for us to think of implementing our own Tor network in the CORE environment. It has almost everything on the client-side defined, and it provides detailed function descriptions. However, it is not very scalable. It puts everything in a single file, which results in high coupling and should not be our practice. Along with this, it is not finished yet, and we can't know whether the logic of some key aspects, such as the directory authority and the circuit formation, would work or not.

Compared to TinyTor, Torpy is more fully-fledged and gains a wider adoption based on the number of stars the repository has. Torpy is also a Tor client implemented with Python. Torpy can be used to communicate with real world hosts or hidden services through the Tor network. All the actual implementation of Torpy is in a subdirectory with great modularity. We can clearly see different components, such as the cell, the circuit, the directory authority, the key agreement, etc.

The modularity of Torpy is what we look for when sourcing the community version. Meanwhile, the more detailed and robust implementation can be a guide to our own implementations.

Combining our investigations into these two open-source projects together, we had several thoughts. The first one was that these two attempts were great additional resources that helped us break down Tor's components and learn the Tor network better. The official Tor services and all the components related to the protocol are written in C, while these two projects are written in Python. Since Python is the go-to language for our project, they are good resources to get us started. The second was that the architecture of directory authority and Tor client are the largest takeaways. As both of them are intended to talk to real world onion services, they are more on the client side. On the client-side, the communication between the directory authority and Tor client is important, and we can take their implementations as reference. However, something that we would not take as reference includes their cell and encryption-related implementations. Cells have many types, and Torpy's implementation of serializing and deserializing cells depend on subclasses, which is a bit complicated. Sending cells requires the encryption and decryption of cells, which involves the security matter. As the main focus of our project is on the network architecture of Tor, we want to simplify the security side. So, we will not take their security design and cryptography setup into account.

4 RESULTS

Now, let's introduce our implementation of Tor inside CORE. We will break it down into different components.

Directory Server

Directory servers, also known as directory authorities, are a few servers that track the state of the entire Tor network. They handle Tor client's requests of knowing what relays are available and form a consensus of all the available relays while keeping listening from relays and updating the consensus. They provide redundancy for better services and distributed trust to prevent the Tor network from being entirely compromised.

In our Tor testbed, we have installed two directory servers as directory authorities; due to the limited scale of our simulation we have not included fallback mirror directory servers [1], which serve as backups or alternatives of authorities. Our directory server uses multiprocessing to handle synchronous requests from clients and synchronous heartbeats from relay nodes. We maintain metadata for those relay nodes with a read-write lock to provide synchronization and avoid race conditions. The metadata of each relay node is formed from the node's heartbeat message. The metadata includes information on the timestamp, the IP address, the port number, the Caesar Cipher key, the node type, and the node status. Clients can fetch a list of metadata by pinging directory servers.

Our design of directory servers tries to be as close to real world one as possible, but with some simplifications. First, our directory authorities do not communicate with each other, while real world directory authorities form a distributed system. Real world directory authorities involve the communication between authoritative directories and their mirror servers. Second, our directory authorities do not form a consensus on relay nodes' information. Since our directory authorities do not talk to each other, they will not combine all the available relay information into a single consensus for the client. Each directory authority will be in charge of maintaining the information of some relay nodes, and the client needs to query all of them to get a full picture of the state of the network. Third, real world directory authorities are involved in the process of client-relay key exchange. As we have simplified the cryptography part, our directory servers do not have such a feature. Instead, our directory servers just make the Caesar Cipher key handy for the clients.

Client

Clients are the interfaces for users to connect to servers using Tor network. They take users' input, find a circuit, encrypt the message to form the onion, and then transmit the encrypted message to the guard relay.

Our implementation of the client can be divided into three parts. First, before transmitting the message to the destination server, the client needs to build the circuit. To do so, the client sends a request to known directory servers and receives the information about relays, including guard relay, middle relay, and exit relay. Note that the information also includes all symmetric keys used to encrypt the plaintext. Second, the client randomly selects three nodes, one from each type, to create a circuit to transmit the message. The 3-Node circuit is the minimal and standard structure for Tor as adding

middle relays won't increase security and anonymity. Thus, our default client supports only 3 nodes. (Later for testing, we write special clients to support more relays.) Then, the client encrypts the message using Caesar Cipher three times by applying the keys of the relays that it selects. During those encryptions, information of the next node should also be added as plaintext in each layer, since only the client has knowledge of the entire circuit and each relay needs to be instructed on where to send its message. Finally, the client sends the encrypted message to the guard relay using a python socket which sets up a TCP connection. After that, the relays will take turns peeling off the encryption, parsing out information of its next node, and transmitting the message to that next node.

We tried to implement our client similar to real world Tor clients. But we still simplified some parts of the implementation. First, the encryption algorithms and key exchange methods were not the same. In the real world, Tor network uses the Diffie-Hellman algorithm to exchange the public keys, while we retrieve the key information from directory authorities directly. Besides, the current Tor network mostly uses AES as the encryption algorithm. In our implementation, we selected Caesar Cipher as our encryption algorithm as it's solid as encryption yet easy to check manually in debugging. Second, the circuit selection method has been simplified. There are much more restrictions for circuit selection in the real world, for example, bandwidth and subnet locality need to be taken into consideration. However, in our model, we randomly select three relay nodes, one from each type, to form a circuit. Third, our packet headers are simpler than real world Tor protocol headers. Our header field of each layer only contains information of the next relay, whereas the header field of real world Tor is much more complicated. Besides, we also ignore the restriction that Tor packets need to be of fixed size. We are comfortable making those simplifications since most of them, such as fixed packet size, are for security concerns in the real world.

Relay

Relays are intermediary nodes between the client and the server to securely transmit the request on behalf of the client to the server or the response on behalf of the server back to the client. Relays are building blocks of all potential paths or circuits in Tor. Relays advocate themselves to directory servers, as mentioned previously, so that directory servers are able to keep a global state of all relays currently active, which may later be queried by clients. In the real world, relays are run by thousands of (currently more than 6000) individuals or organizations as volunteers.

As mentioned in previous sections, our testbed in CORE contains 2 directory authorities running independently. In the real world, a list of 10 directory authorities around the world is published and each node, either a client or a relay, is assumed to hold all information about those authorities before any further queries. We held similar assumptions in our simulation, namely we hard-coded the addresses and ports of each directory authority into the relay implementation. There are two routines in relay implementation, each running in an independent process. The first routine is responsible for contacting the directory authority. The relay will attempt to connect to each directory authority as it starts to run. When connected, the relay will use its own IP address to generate a random key as the encryption/decryption key in Caesar Cipher for future communications. The relay type (guard, exit, or middle), as well as the opening port, is provided as command line arguments to the relay. The relay will periodically send metadata that describes itself to the directory server in the form of short heartbeat messages, which includes fields such as timestamp, IP address, open port, key, type, and status (ON or OFF). The second routine of relay is responsible for receiving the message from the previous hop in the path, decoding the message and extracting relevant fields. Each relay will be able to know its next hop address and port number once it decrypts the message using its own key, and will wait for reply from the other direction of the circuit and send it back.

Our design of relay is also simplified from a real world fully protocol-compliant design. In the real world, the security and cryptographic features of the relay are much more involved. For instance, each relay is expected to keep multiple public/private key pairs, including either long-term, medium-term or short-term 1024-bit RSA keys, Curve25519 key, and Ed25519 keys. These keys are all of different purposes. We do not implement these features due to the shift of our major focus from security and cryptographic features of Tor. Besides, the real word metadata sent to directory authorities are much more complicated, including fields like bandwidth. Those information are given to DAs so that client can make better relay selection in the

realistic network environment.

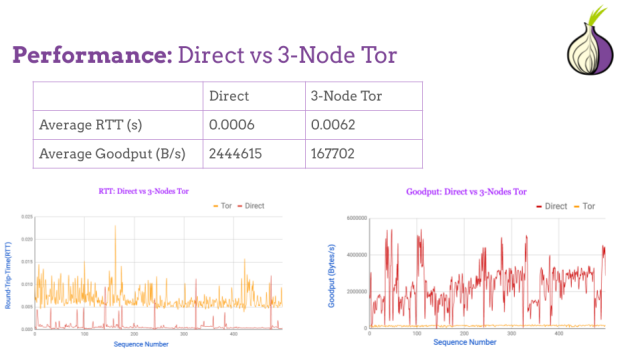
Test

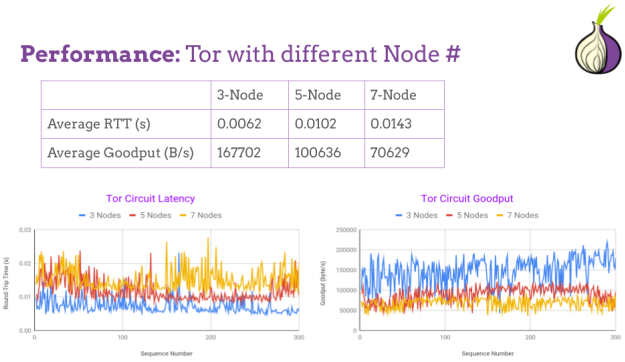
Having every component ready, we are going to test the performance of our Tor model. As mentioned earlier, we want to see how efficient our Tor model is. So, the metrics we choose are round trip time and goodput of a message, where round trip time is the time interval between the client sending a request and receiving its corresponding response, and goodput is the amount of application data transferred in unit time.

To make a comparison, we select direct connection as the first control group, where client and server just communicate directly. To collect data, we create log files from the client side for our parser to analyze. A timestamp is written to the log right before making the initial connection and right after parsing out the response. The application data length is also written once after parsing out the response. For simplicity, we use a basic echo server. And we write scripts just for testing, where fixed 990 bytes of data are repetitively sent as payload. Note that when collecting data, we ignore the time the client needs to communicate with a directory authority and select its Tor circuit. The reason is that this process happens only once both in our model and in the real world (for most cases). Besides, this delay also depends on how busy the DA is. So, we don't count this one-time delay for our client to prepare the Tor circuit. However, the delay for the client to process the request (creating an onion) is considered as it's a routine for each request. We put this process after setting up a connection with the guard node; so, the place to log the beginning timestamp is reasonable.

The results between direct connection and standard 3-node Tor network are shown here. As you can see, the 3-Node Tor circuit is about 10 times slower than direct connection. This is expected. Although our encryption algorithm is a simple O(N) Caesar Cipher, encryptions and decryptions still happen multiple times, and there are multiple TCP connections set up in order to form the circuit.

Next, we would like to see how changing our Tor network configurations may affect its performance. Since our circuit selection is too naive to have any space for modification and circuit selection delay is not considered when collecting data, we only compare the performance for different numbers of relays in the circuit. We pick 3, 5, and 7 relays to test. Each case has its own test





script but the client requesting behavior is the same. Results are shown here. As you can see, adding two nodes in middle roughly causes an extra 4 ms delay in terms of round trip time. That's about of the standard 3-node round trip time. Thus, the extra delay is decent. Moreover, if we test with a very large number of relays, we expect the metrics to be proportional to the relay number since each relay has similar behavior and should generate an equivalent amount of delay.

Those above are our main test results. In the real world, Tor is much slower than our implementation since the key exchange and encryption parts take much more time. And there are other processes to ensure security.

5 CONCLUSION

We have explored Tor as broadly and deeply as we could throughout the project. Tor has both superior capability design-wise and great complexity implementation-wise, which truly impressed us. To be more precise, technical-wise, Tor is an ever-evolving and ever-growing protocol family with numbers of actively maintained specifications and proposals describing its key components

and functionalities. Further, considering its spillover social impact, just like any other large-scale complex systems or projects, Tor network is never a simple, self-contained, "small and beautiful" construct to target a specific need or solve a specific problem but rather a huge community-driven distributed network with different organizations or individuals running its key components and services around the globe, with ambitions to improve the privacy and security on the Internet. We treat our journey in this project as a great opportunity to taste the experience of approaching an unfamiliar family of multidisciplinary technical concepts, building our knowledge base from scratch, learning through official specifications and existing implementations, balancing the trade-off between feasibility and protocol-compliance, and so on and so forth. In our original plan and proposal, we assumed that Tor, as a project that mainly tackles the concerns among various aspects of the Computer Networks, would mainly raise challenges for us on understanding how network protocols work and how infrastructures operate to support those protocols. We have analyzed the knowledge and skills required to implement Tor, as we conducted our research into the technical overview, online documentation, and tutorials, as well as official and non-official implementations of Tor. It turned out that Tor involved much more demanding technology and concepts in domains other than Computer Networks as well, especially Security and Cryptography. We realized that Security and Cryptography are indispensable and key components of Tor, as expertise in those areas is mainly responsible for fulfilling the mission of Tor, namely bringing extra anonymity and security into the Internet. In other words, other design choices of Tor concerning the network topology, including path selection, circuit building, etc., are tightly integrated with and accommodating the security mechanisms adopted in Tor, including key exchanges and layers of different encryption algorithms on a hop-to-hop or an end-to-end basis. Unfortunately, Security and Cryptography are not the main focus of Computer Networking and our team lacks the background in these areas. Given the tools and knowledge that we currently possess, we found it more realistic to grasp Tor's core idea and design philosophy regarding network topology, i.e. how connections are made among clients, servers, relays, and directory servers to achieve different features and functionality described by the specifications. Thus, CORE as a powerful network emulator became handy to us, as it has already enabled us to conveniently design and test our whole network topology. While investigating those aspects of Tor, we mainly focused on the motivations, reasoning, and trade-offs behind Tor's design decisions. Those explorations turned out to be quite inspiring and all of us have acquired experience in designing and developing a rather complicated system. Admitting the adaptations to the original plan as well as the simplifications we made to the design and implementation due to realistic challenges and limitations, we still see opportunities and potentials to dig deeper and attempt more in this project, given the efforts that we have already made. With the introduction of knowledge and skills in the fields of Security and Cryptography, we may develop further understanding of the security side of Tor specification and implementation and modify our current implementations to incorporate those security features. We could also attempt to scale our existing topology to include more clients, servers, relays, and directory servers, meanwhile bringing in more system parameters to better simulate the real world scenario. We believe that by conducting this project we've had a great experience in touching real world network systems, and we hope that such experiences would be useful in our future learning paths.

6 COMMENT

This is a public git repository with the final works of this project: *Tor@CORE*.

During development, the team had some trouble with shared teamwork synchronization. As a result, the original repository is messy with different versions of code spreading everywhere. So, to clearly demonstrate what we've accomplished, we decided to rearrange everything and publish this public repository.

Please refer to its readme page for more details.

REFERENCES

[1] [n. d.]. Fallbackdirectorymirrors · wiki · Legacy / Trac. https://gitlab.torproject.org/legacy/trac/-/wikis/doc/FallbackDirectoryMirrors
[2] 2021. Is tor safe? learn how secure tor is. https://protonvpn.com/blog/is-tor-safe/
[3] Marten4n6. [n. d.]. MARTEN4N6/tinytor: A Tiny Tor client implementation (in pure python). https://github.com/Marten4n6/TinyTor
[4] Torpyorg. [n. d.]. Torpyorg/Torpy: Pure Python Tor Client Implementation. https://github.com/torpyorg/torpy