Report: TOR File Sharing

Seongjun Lee: j2v8, Louis Mau: l5l0b, Jovan: w0c9, Andrew Chang: x7y8, Simon Plath: z4n8

Abstract

By utilizing onion routing and the anonymity that it brings to its users on the network, the program achieves a working file transfer protocol sharing similarities with BitTorrent. By connecting to routing nodes, information is sent from nodes with a file to a client who wants a file. Chunks are encrypted and then passed through the network to achieve anonymity and security.

1. Introduction

In terms of heavily classified or confidential files, it makes sense that a file sharing network between a group of those with the correct clearance would be encrypted; namely to maintain the clandestine nature of the data. Combining the TOR model with a file sharing system similar to that of BitTorrent, we have managed to produce a file sharing system that uses onion routing to make sure any intermediary stops the data has to make are adequately sealed off from snooping.

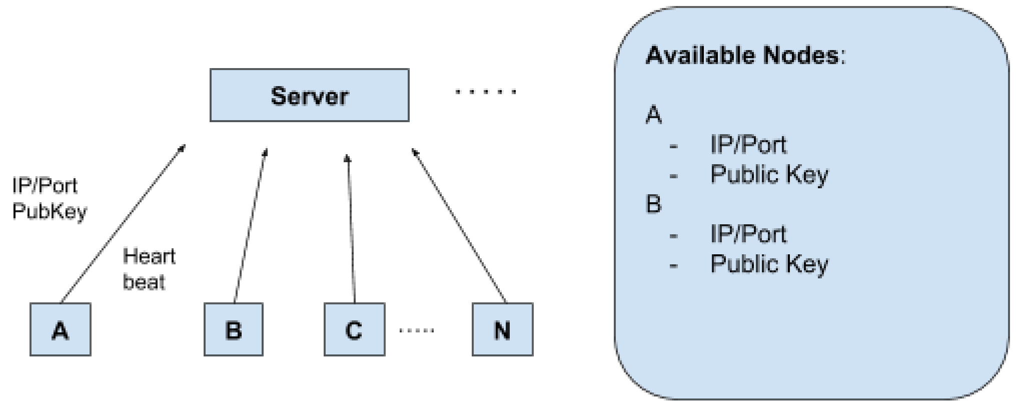
TOR is an anonymity system implemented using onion routing, providing clients a method to conceal their network identity or IP address in the network. At a basic level onion routing consists of a number of nodes (machines) forming an onion network. Data, sent in the form of a multi-layered encrypted structure, is sent from the client application along a route in the network to the destination machine or server. At each node in the network route, a layer of the encrypted data is decrypted using the corresponding private key held by that node, before it is sent forward. This means packet content and absolute source and destination remain hidden from outside eavesdroppers, as well as nodes along the route as they only know about nodes immediately before them and after them in the route.

As with most distributed systems file sharing using a TOR network possesses many challenges. Node and server failures at different times need to be accounted for and dealt with appropriately to prevent the system from crashing or hanging certain times. Unique challenges to implementing a TOR network include dealing with nodes failing in the route selected when downloading and transferring files, as well as dealing with data limits when encrypting files. Through our implementation we feel we have adequately addressed these problems providing a robust TOR file sharing program. Furthermore, we have provided flexibility to the system by allowing the number of nodes in routing to be predetermined allowing a tradeoff between security and speed, with a higher number of intermediary nodes providing more security but slower transfer speeds.

In the following sections we will first describe the design of the system (section 2), followed by the implementation of the system (section 3). Then an evaluation of the system (section 4), and finally concluding with the some of the limitations imposed on the system, as well as possible improvements (section 5 and 6).

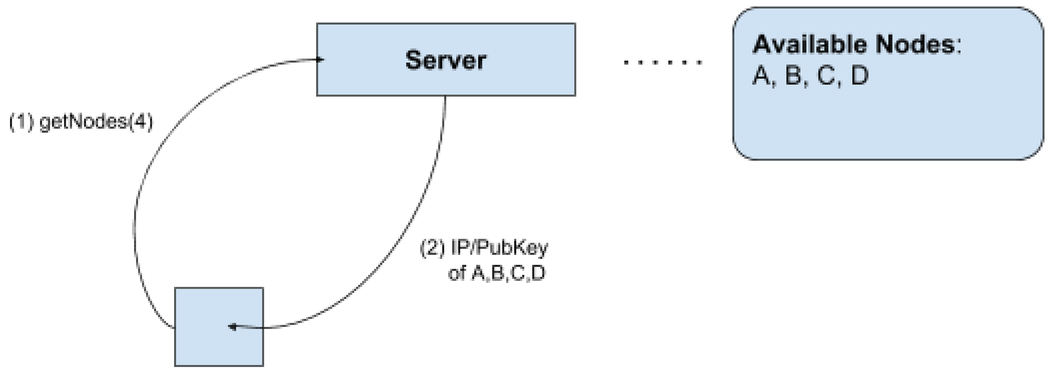
2. System Design

1) Connection with Server



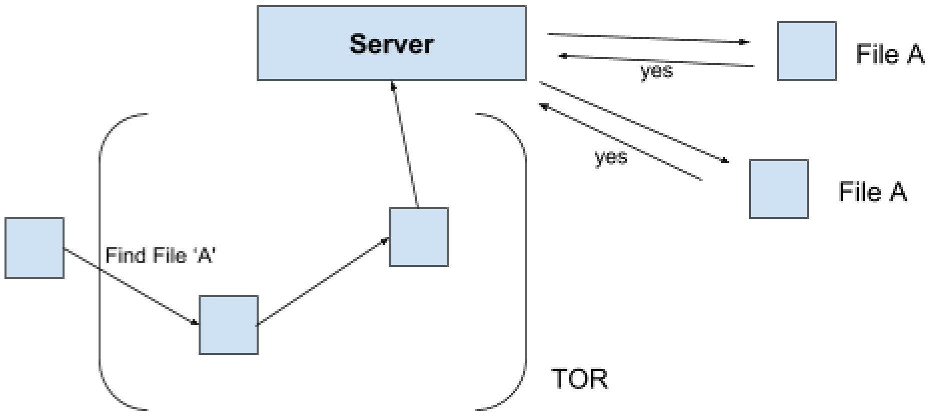
When a new node joins the TOR network it registers with a server, passing on its IP/Port and public key. The server stores this metadata and keeps track of nodes which have registered and are currently connected to it. Additionally, a heartbeat is setup from each node to the server and is used to determine disconnects on the client. The servers maintains a list of available nodes to participate in the TOR network which is subsequently updated upon disconnects.

2) Getting Nodes for Routing



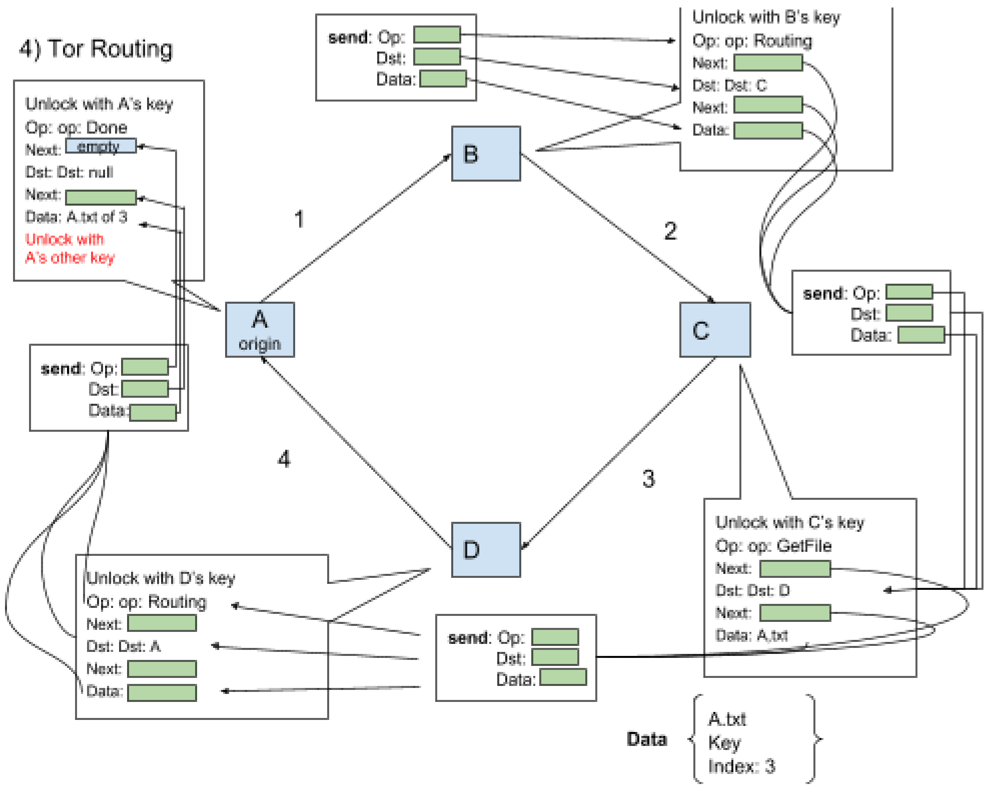
Before a client (node) requests a file chunk, they must receive a list of nodes which will form the route in the TOR network to the destination node or server. Upon requesting a list of nodes from the server, the server responds with a list of available nodes and their corresponding IP addresses and public keys. This is then used to encrypt the data and transmit it to its destination in the TOR network.

3) *Search for Files (TOR)*



If we search for a file, we will use the TOR network to ask the server which of the nodes have the file. The server will respond back with all the nodes that have the file. Server also provides detail of files if there are more than one file with same name but different content.

4) *Tor Routing*



The system uses the TOR network to get the chunks of the file. In the diagram, A is the node requesting the file, C is the one with the file, while B and D are used for routing. Since we are getting small chunks of the file from each nodes, we have to make multiple getChunk requests to many different nodes that have the file. Finally, node A needs to gather and combine all the chunks back into a single file. The layer of encryption guarantees anonymity of the transfer.

3. Implementation

Application:
The application is the interface for the client and provides a way for the user to interact and communicate with the node through the node API which acts as a communication bridge for the node and downloader. It displays what is happening on the surface through print statements and reads/scans the user's inputs when needed, such as at the menu with a bunch of options.

Server:
The servers store a list of available nodes (as well as servers) and their corresponding IP addresses and public keys. The servers accept connections from nodes and maintain a map of active nodes, as well as handling search and heartbeat functions. To maintain synchronization across servers a sync operation was implemented which runs repeatedly after a specified time delay to maintain concurrency. Additionally, each server has a mutex implemented for map access in order to eliminate concurrent map accesses.

Node:
When a node starts up it first connects to the server retrieving the list of available servers and begins sending heartbeats at predefined intervals to ensure the server is alive. Once running the each node waits and listens for connections from other nodes in the network. As part of the TOR network each node handles incoming RPC calls, decrypting the data with its own private key and making an outgoing RPC call to the next node in the route, passing the inner layer of encrypted data along, as specified by onion routing.

Downloader:
Upon the request of a client or application to get or download a file chunk the downloader is responsible for encrypting the data and route in a layered message and sending the data along the preset route. That is, the downloader creates the layered encrypted message representing the onion structure with data at the centre.

File Manager:
The File Manager for each node is responsible for the actual searching, reading and writing of the files and file chunks into the set directory or a directory specified by the client through the application. It is responsible for reading and breaking up the requested file into smaller chunks as well as combining (writing) them back into a single file at the receiving node. Finally it

checks that the file hash is the same as the original before finishing and cleaning up by removing the file if it isn't the same.

Cryptography:
Initially RSA encryption alone was intended to encrypt and decrypt data passing through routes in the system. This was problematic since data limit size for RSA encryption was found to be only 245 bytes, allowing only small units of data to be transferred. To solve this issue a hybrid system using AES and RSA encryption for data was implemented. In this method a symmetric AES key which is able to encrypt much larger data sizes is first generated and used to encrypt the data. This AES key is then encrypted using a RSA public key, sufficient for AES key encryption. When the data is decrypted, the encrypted AES key is first decrypted using the RSA private key, upon which it can then be used to decrypt the corresponding data.

Node Failures:
During a file transfer if a node fails, to retain consistency for the 3 node minimum routing, a known node will be used as the 3rd node. So the data will be passed through a node twice. And if one more node fails the node will encrypt itself and go ahead with that. If a node is not being used and owns a file then it will just fall off the map and not get put as a known node with the file.

To handle server failure, we have each node maintain a list of available servers. When a node first connects to a server it retrieves and stores this list, and upon reconnection the node queries different servers in the list, connecting with the first server which is alive. This requires the assumption that the first server it contacts to retrieve the list of servers is always running.

Testing:
Testing consisted of unit testing, integration testing and manual testing. Unit testing was used for some encryption/decryption and file manager functions. In many cases we found this form of testing impractical, as most methods involved rpc calls or network related functions. Our main form of testing involved manually tests where a server and multiple applications were run and tested for correct behavior under different operations and commands.

4. Evaluation

The primary way to evaluate our system is through the use of system logs, to observe the sequence of operations and behavior displayed by the system in response to different commands. Server and applications both log to the console or application on which they are running. Evaluation may also occur by verifying that files are properly downloaded and operations such as search returns the correct results.

5. Limitations/Assumptions

Although server replication was implemented, nodes must be able to connect to the first server they try, and therefore we assume this server is always running (never fails). There must also be a starting directory named “data” for every node/client, as this is what it’s set to in the file manager and where the file manager will look in first unless the directory is changed through the application. Furthermore, we assume every node possessing a file has the complete version of the file. When clients are downloading, they will only contact nodes with the complete version of the file and clients know what file they are looking for in the network.

6. Improvements:

One possible improvement would be to add a GUI for clients participating in the network, in an effort to improve the user experience and usability as the current user interface consists of a console window and command line. Another improvement would be to allow the ability for seeders to add new files to be shared or downloaded on the network.

References:

1. (eDonkey File-Sharing Network)
https://pdfs.semanticscholar.org/242a/8ad865c8b2d40caafecd4a88a4af99b75624.pdf

2. (Peer-to-peer networking with BitTorrent)
http://web.cs.ucla.edu/classes/cs217/05BitTorrent.pdf