Decentralized Systems Engineering project: Private Message Exchange and File Sharing using Onion Routing for Peerster

Artem Shevchenko, Riccardo Succa, Aleksandr Tukallo

December 2018

1 Introduction

One of the main problem of the Peerster is that all messages are going around network in unprotected form: any node on the route between origin and destination can easily find out who is communicating with whom or who is downloading a certain file. Our project will introduce encryption and anonymity using onion routing. The motivation to choose this problem was that now anonymity and personal data protection is highly important so we can do something as a project and get some knowledge in this area.

Moreover, we also want to solve the problem of sharing big files, that is now impossible because of the size limitations of the metahash. A merkle-tree solution will solve this problem, giving us a system that that allows to download big files in an anonymous way.

2 Related work

The idea of onion routing is quite simple [1]: messages are encapsulated in layers of encryption, so that each subsequent node of the path is able to decrypt only its layer. Consequently, an attacker cannot infer the source or destination of a packet: he can only see that a packet is coming from a node and going to another node, but the two can be at any step of the tor chain.

In order to create a chain, it is necessary to know all nodes who can provide such service. Let's see how it is done currently. Tor uses the concept of Directory Authority (DA) servers. The DA servers are the trusted providers of a consensus that contains the complete information about each known Tor relay and is periodically updated. When it's time to update the list, a majority of the directory authorities must agree on the accuracy of the new list by cryptographically signing the proposed consensus. Once this process is complete, clients are able to download the updated list of relays. If a seizure attempt is able to take down significant number of DAs, the network will be in an unstable state and the integrity of any updates to the consensus cannot be guaranteed. This is a disadvantage of such approach. We will solve this problem using blockchain which will store all nodes participating in onion routing and their public keys. The example of usage of blockchain as a base for the public key infrastructure (PKI) is described in [2]. Nodes will be able to publish their public keys and find public keys of the random onion route on the blockchain. Our implementation of blockchain for PKI will be based on the ideas of the blockchain that was implemented for filename to hash mapping with some changes. However, we decided that it is better to have another blockchain for this purpose and not mix them.

Moreover, as was mentioned above, we are adding the possibility to share big files via Merkle trees. The data structure used was initially described in [3]. Why to use Merkle tree at all? What are the other ways to share a big file? The naive approach is to avoid using hashes at all and just split the file into chunks, give each chunk a natural number and then let the downloader request chunks one by one. Compared to such an approach Merkle tree gives a great advantage – for every chunk hash is provided, which guarantees data - correctness. Moreover, the topmost metahash (called root hash) uniquely identifies the file and wrong file cannot be sent instead of correct one.

3 System Goals & Functionalities & Architecture

We decided to implement a simpler version of the Tor protocol, adding some new features to the architecture. The following 3 main steps explain how we divided the work and how the parts were combined in order to make a single complete project.

3.1 Blockchain

This part is done by Artem Shevchenko. Goal: provide nodes in the network with the list of nodes that are participating in onion routing and their public keys. Functions: allowing publishing transactions with nodename to public key mapping on the blockchain and keeping such mapping unique; allowing getting a random path for onion routing containing 3 nodes except for sender and receiver. The idea of blockchain will be implemented close to what was done in the last homework but will have some changes. The separate blockchain will be implemented.

We will make it slower to produce blocks and will make it mine all the time, not only when we have transactions. By doing this we can protect the data from changes under the assumption of non-malicious majority.

New types of messages and the structure of block:

```
type TxOnionPeer struct {
    NodeName     string
    PublicKey    []byte
    HopLimit     uint32
}

type BlockOnionPublish struct {
    Block     BlockOnion
    HopLimit  uint32
}

type BlockRequest struct {
    Origin       string
    HopLimit     uint32
    BlockHash    [32]byte
}

type BlockReply struct {
    Origin       string
    Destination  string
    HopLimit     uint32
    BlockHash    [32]byte //hash of requested block
    Block        BlockOnion
}

type BlockOnion struct {
    MinerName     string
    PrevHash      [32]byte
    Nonce         [32]byte
    Transactions  []TxOnionPeer
}
```

When a new node joins the network, it generates public/private keys pair, broadcasts the transaction and starts to mine on top of genesis block (in our case it will be imaginary block with hash [000...000]). However, we want to relax the requirement from homework that all nodes must see all blocks (or, that all nodes must start simultaneously). We introduce a way to download the history of blocks from other nodes. When a node receives a block with a parent that it had never seen before, it will not discard such block. Instead, the block will be added to a temporary storage and broadcasted further. Meanwhile, the node will broadcast periodic requests (every 10 seconds) for the parent block to all neighbours. When a node receives a request for a block, it either sends back a response with an appropriate block (if it can) or broadcasts the request further. The block response is sent in point-to-point manner to requesting node. When the requested parent block arrives, everything is repeated again until the block whose parent exists in the blockchain is received. After that the chain is moved from the temporary storage to the main one and is treated as a normal chain of the blockchain.

To provide the node with the information for routing, the querying function was created. This function returns the public key of destination node and names and public keys of three intermediate nodes.

3.2 Onion routing protocol

This part is done by Riccardo Succa and implements the private messaging and file downloading through onion encryption. The onion protocol chooses by default 3 nodes to construct a path from the sender to the receiver (the same way as Tor does). The available nodes are requested from the blockchain. The 4 public keys associated to the nodes, i.e. the 3 randomly chosen plus the destination node, are used to create the layers of encryption. Consequently, a new message is defined:

```
type OnionMessage struct {
        Cipher           []byte
        AESKeyEncrypted  []byte
        Destination      string
        LastNode         bool
        HopLimit         uint32
}
```

The algorithm works as follow. We generate an AES key and encrypt the message we want to send (it can be a PrivateMessage for peer to peer messaging or DataRequest/DataReply for file downloading). Then, this AES key is in turn encrypted with the RSA key of the destination node, which has been received from the blockchain. Both the cipher and the encrypted AES key are part of a new OnionMessage and the OnionMessage itself is set inside a GossipPacket. The procedure is repeated 3 more times, each time with the public key of a new node of the path.

When a node receive a OnionMessage packet, it first checks if it is the destination node. In this case, it decrypts the encrypted AES key with its private RSA key, and with this one it can decrypt the cipher, basically removing his layer of encryption. Moreover, if it is the final destination node (LastNode boolean set to true), the decrypted cipher is indeed the original message that the sender wants to send, and it can be handled as normal. Otherwise, if the node removes his layer of encryption but it is not the final destination (LastNode set to false), it means that it is an intermediary node of the path: in this case it simply forwards the message to the subsequent path's node, written inside the destination string.

Onion encryption is used to send private messages or download files. With private messages, an attacker can neither read the message (thanks to the first layer of encryption) nor understand who is talking to whom (thanks to onion encryption). The same is applied to file downloading: the content of the file is encrypted and we can't understand who is downloading it. Two new special buttons for sending with onion encryption are added to the WebClient.

3.3 Merkle tree filsharing

This part is done by Aleksandr Tukallo. In default Peerster implementation only one metafile was present, that bounded the maximum possible file-size n with $\frac{n}{c} \cdot h \leq c \Leftrightarrow n \leq \frac{c^2}{h}$, where h is hash-size, c is chunk-size. With offered chunk-size and hash-size choices (8096 and 32 bytes respectively) the maximum file-size to share was limited with $2Mb$, which is undoubtedly a serious limitation.

The Merkle tree was introduced to solve the problem of file-size limitation. Merkle tree is a hash tree implemented in the following way: the file is being divided into chunks; for every chunk hash is calculated; all the hashes are concatenated; they're once again being divided into chunks; their metahashes are being calculated, etc. Procedure terminates, when the number of chunks obtained after concatenation is 1. The last chunk obtained is called *root chunk* with respective *root hash*.

Now file-sharing works in the following way. Every shared file in uniquely identified with it's *root hash*. When Peerster gets *DataRequest* with *root hash*, it replies in a usual way with *root chunk* body. The only difference introduced to *GossipPacket* is that now *DataReply* has a field *Height*:

```
type DataReply struct {
        Origin        string
        Destination   string
        HopLimit      uint32
        HashValue     []byte
        Data          []byte
        Height        uint32
}
```

When Peerster gets *DataReply*, it splits reply-body into hashes and requests these chunks later (downloading of the tree is done with BFS). If height of received data is 0, then data is not splitted into chunks, but just saved, waiting for other file-chunks to be downloaded to reconstruct the file.

Analysis: with tree height k file of size $n \leq \frac{c^{k+1}}{h^k}$ can be shared. With offered chunk and hash size file of $2Tb$ can be shared with $k = 4$, which is probably more than enough for most of the cases.

For the efficiency of data-replying all the tree is stored in memory, RAM. Though an important optimization to reduce RAM-usage is introduced. One may store all the levels of the tree in memory. It will lead to shared file size being bounded not with the tree height, but just with RAM size, which is often very small compared to persistent memory size. To solve this problem all the levels of the tree, except the lowest one are stored in the memory. The lowest level chunks are stored in separate files on disk and are read only when demanded. Such an optimization leaves a set of present-chunks-hashes in memory, though it reduces the tree-size in RAM hugely, because the lowest layer is the biggest one. Every next layer is $\frac{c}{h}$ times smaller, that is 256 with offered sizes choice. So, exact mathematics is not given here (it's too complicated for such an obvious result), but such an optimization allows the user to share the file of size up to approximately $256 * RAM$ bytes, which is with $16Gb$ RAM exactly $2Tb$ and, once again, good enough for most of the purposes.

Using Merkle tree together with Onion encryption we can share and transfer large files in an anonymous way. Though onion encryption leads to an expected drawback – big files sharing is significantly slower with encryption, than without it.

References

[1] R. Dingledine, N. Mathewson, and P. Syverson. TOR: The second generation onion router. In Proceedings of the Usenix Security Symposium, 2004.

[2] M. Ali, J. Nelson, R. Shea, and M. J. Freedman. Blockstack: A Global Naming and Storage System Secured by Blockchains. In 2016 USENIX Annual Technical Conference (USENIX ATC 16), pages 181–194, Denver, CO, June 2016. USENIX Association.

[3] Merkle, R. C. (1988). "A Digital Signature Based on a Conventional Encryption Function". Advances in Cryptology — CRYPTO '87. Lecture Notes in Computer Science. 293. p. 369.