OnionRPC: Final Report

Sherman Zhu, Joel Broek, Peter Lee, Victor Zhu, Jack Yuan, Yu Tian

> Github Repository

Introduction and Background

While conventional internet protocols incorporate encryption which obfuscates the contents of messages between clients and servers, it is still possible for malicious actors to determine other information about the communications, such as which computers are communicating with one another, and the volume of communication between them. When individuals or organizations need to conceal this information as well, additional protocols are required.

Onion routing is a family of protocols which facilitate anonymous communication over the internet. Messages in an onion network are passed through a sequence of nodes in the onion network, called a circuit, and repeatedly encrypted so that no router knows the source and destination of the message, or the full sequence of nodes in the circuit, only the preceding and following nodes in the chain.

Our project consists of a simplified version of onion routing, which supports jsonRPC (1.0) between applications.

Solution

https://www.figma.com/file/kP9OXD9I8nZgCYLmx5RpY4/

We proposed to implement a simplified onion network that would support jsonRPC calls to an jsonRPC server from a golang client.

The system would have three components: a client-side/server-side library for sending/receiving RPC calls over the onion network, a coordinator server which orchestrates the onion network (keeps track of and monitors nodes), and a number of onion routers ("nodes"), which come in three varieties: guard nodes, which are the first nodes in a circuit and interact directly with the client, exit nodes, which are the final nodes in a circuit and interact directly with the RPC server, and relay nodes, which serve as intermediate steps in the circuit between the guard and exit nodes.

Assumptions

1. Coord and mock rpc server will never fail.
2. Guard, relay, and exit nodes will not fail during the joining protocol, but may fail at any time thereafter.
3. Coord metadata (e.g. IP address & ports) is public and known by every server & client.
4. Communications between guard, relay, exit, and client might fail.
5. The system will not need to scale to more than 1000 nodes

Guarantees

1. The network will continue to function so long as there remains one available node of each type (guard, relay, exit).
2. Guard nodes and relay nodes will not have access to request payload from client, response payload from mock rpc server, or mock rpc server IP.
3. Relay and exit nodes will not have access to client IP after client has negotiated shared secrets with relay and exit node. (All images have hyperlinks to figma diagram)

Client Interface

After client is started, client initiate an jsonRPC by calling client.RpcCall() method which will take 4 arguments, server address, rpc service method string (Server.Add, Server.Subtract, Server.Multiply, Server.Division), argument struct, and a pointer to the response struct.

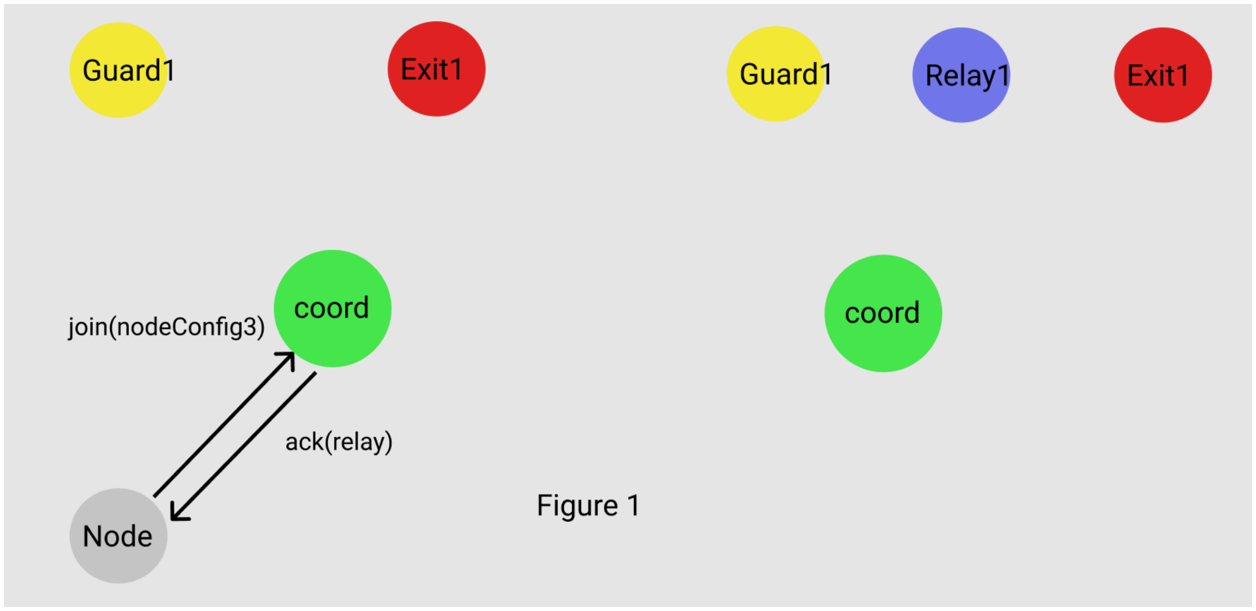
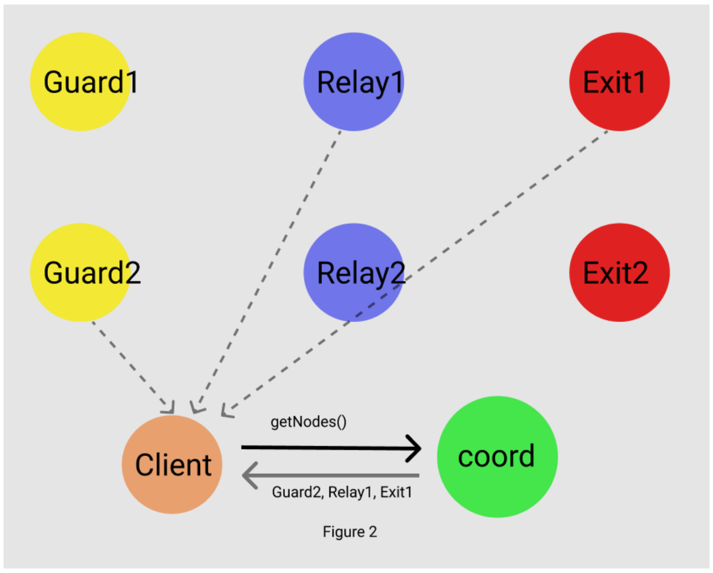
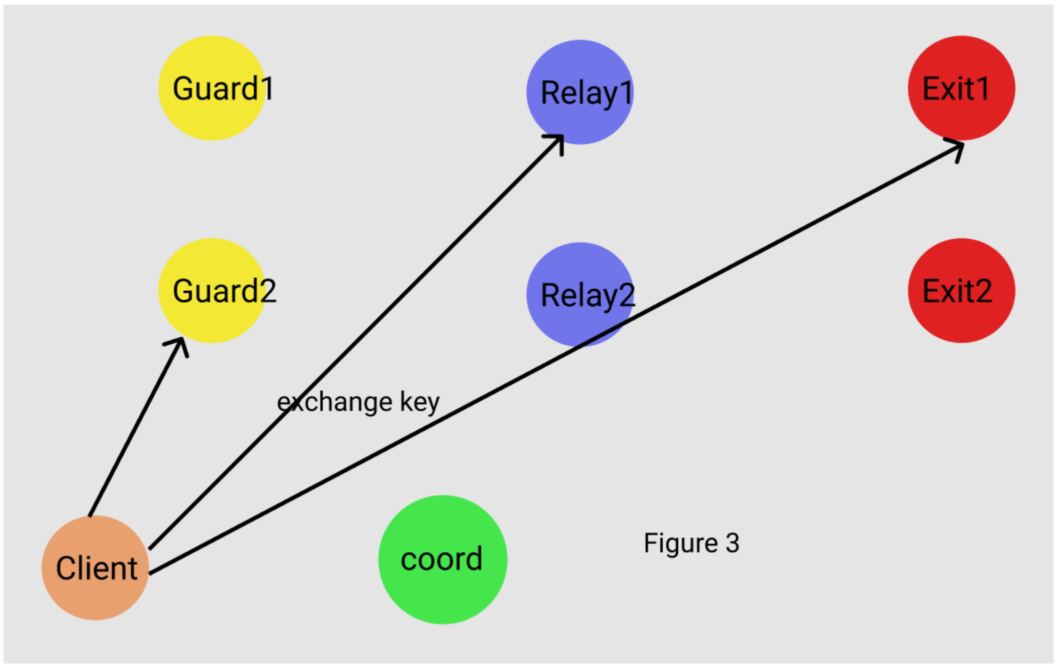


Figure 1

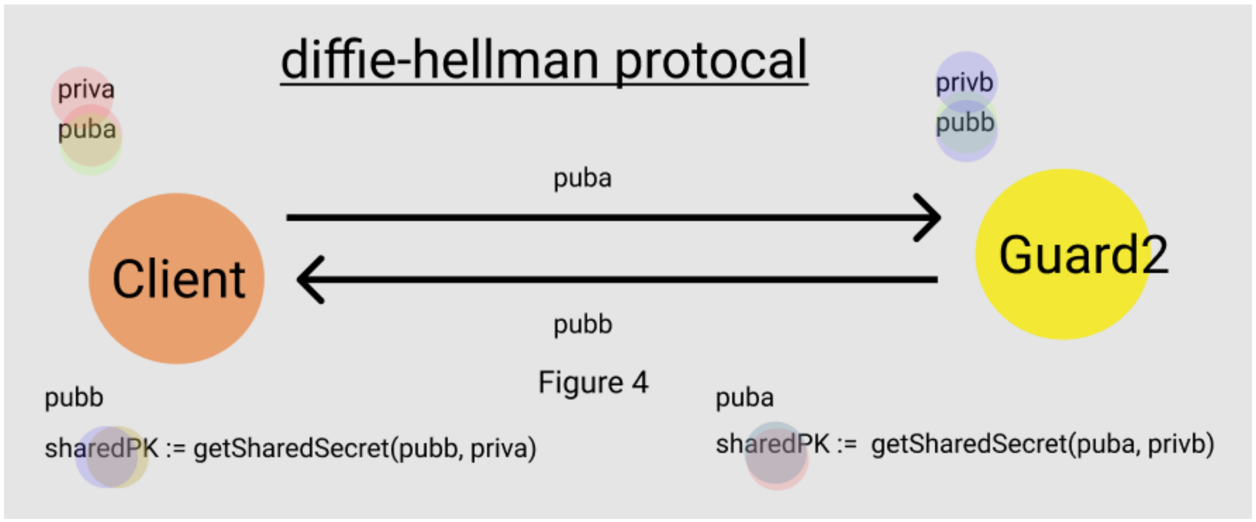
When a relay/guard/exit node joins the system, it sends a join request to the coord server which in turn assigns the node a role (relay/exit/guard) based on the needs of the system at that time. As nodes join the system, the coord assigns roles in order to construct a minimally functioning onion network (2 nodes for each node type). Once that is accomplished, the coord assigns roles in such a way as to maximize the robustness of the network ($\frac{\#guard}{\#relay} = \frac{\#exit}{\#relay} = predefined\ constant$). Upon receiving a role assignment from the coordinator, the joining node starts the corresponding protocol (relay/exit/guard).



When the client needs to establish a connection to the RPC server and the client does not have working relay/guard/exit node metadata stored in memory, the client will request relay/guard/exit node metadata from the coord server.



Then the client will negotiate shared secrets with relay/guard/exit nodes using diffie-hellman protocal. If the negotiation process fails at any step, the client will request a new set of relay/guard/exit node metadata from the coord server up to 100 times and renegotiate shared secrets with the new set of nodes.


diffie-hellman protocal
priva
puba
privb
pubb
puba
Client
Guard2
pubb
pubb
sharedPK := getSharedSecret(pubb, priva)
puba
sharedPK := getSharedSecret(puba, privb)
Figure 4

First, both client and onion node generate a private key (`priva` and `privb`), then client and guard will generate public keys `puba` and `pubb`. Then client and onion node will exchange public keys and use private keys along with received public keys to generate shared private keys.

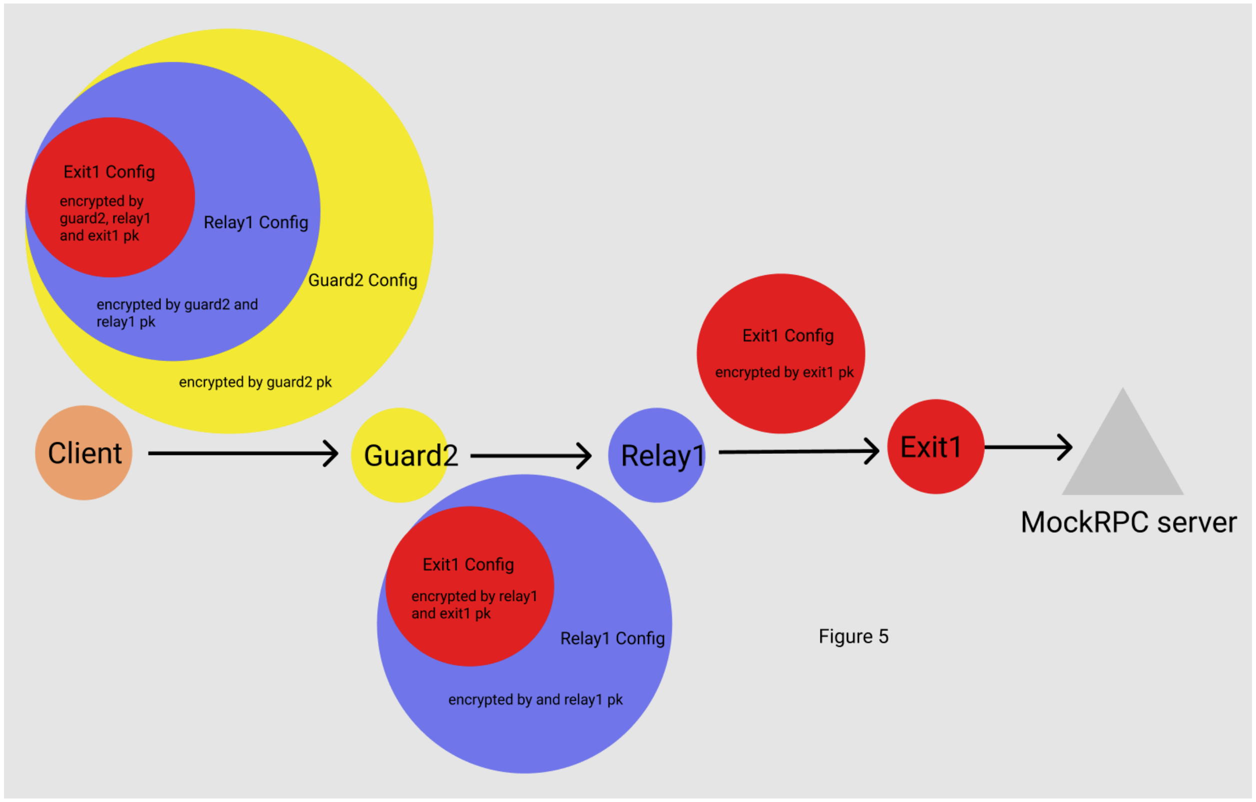
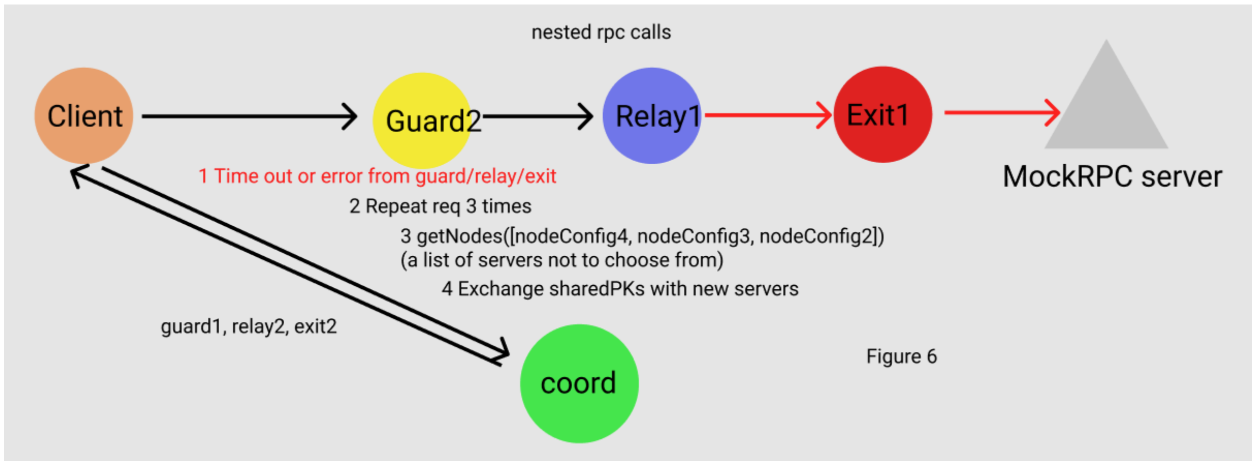
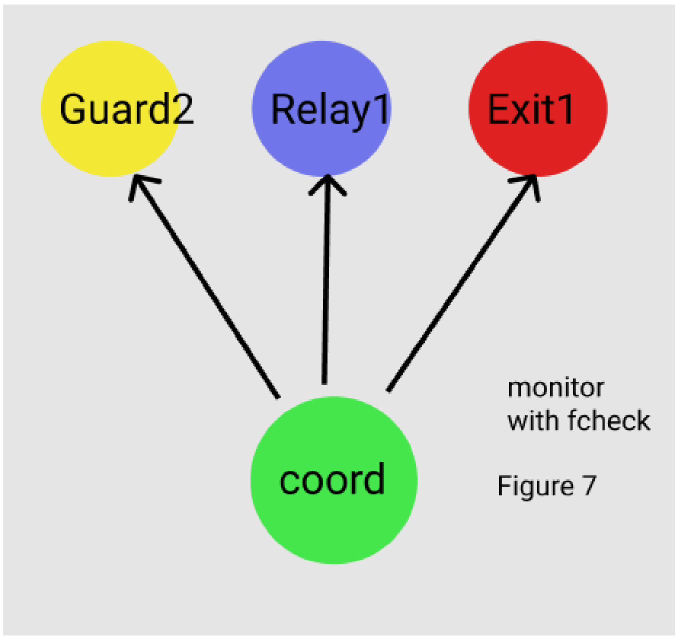


Figure 5

After the client has gathered all the necessary information to establish a connection to the RPC server, the client will encrypt the RPC payload/server metadata in onion layers as shown in the diagram, and forward the request to the guard node. Each node will decrypt the message using its shared secret key to access the next node server config. After the exit node receives the response payload from the mock RPC server, each node will encrypt the payload using its shared secret key it acquired early by means of the Diffie-Hellman protocol. Finally, the client will use gathered shared secrets to decrypt the message.


nested rpc calls
Client
Guard2
Relay1
Exit1
MockRPC server
1 Time out or error from guard/relay/exit
2 Repeat req 3 times
3 getNodes([nodeConfig4, nodeConfig3, nodeConfig2])
(a list of servers not to choose from)
4 Exchange sharedPKs with new servers
guard1, relay2, exit2
coord
Figure 6

The client operates on a timeout. A client resends an RPC up to three times on any error or timeout. After three iterations for a given RPC, the client sends the current server chain config to coord and requests a new server chain config. Coord does not send back any server configs in the failed chain. Then the client negotiates shared secrets with new servers.



The coordinator continuously monitors nodes using fcheck. When coord detects a server failure, the coord marks it as failed and coord does not include it in future circuits sent to the client unless it rejoins.

Limitations:

OnionRPC will only support jsonRPC 1.0 implemented by golang standard library.

Evaluation and Results

To test our project, we created golang clients which requested onion circuit configs from coord, negotiated shared secrets with guard/relay/exit nodes, and sent RPC calls to the RPC server through the onion circuit. We added some hardcoded delay to each onion node and created some failure cases to ensure the system properly handles failures. Such failures included:

- One client sending RPC request with a terminated guard node
- One client sending RPC request with a terminated relay node
- One client sending RPC request with a terminated exit node
- 2 clients sending RPC requests with constantly terminated/joining guard/relay/exit nodes on localhost

Our mock RPC server offers four remote procedures: add, subtract, multiply, and divide. The client will constantly send add/subtract/multiply/divide requests in a loop. All nodes will log intermediate payloads before/after encryption/decryption. If the system works correctly, clients should eventually receive a correct number from the RPC server, the guard/relay node log should not show the returned value from the RPC server, guard node log will not show the addresses of the exit node or mockRPC server, and exit node log will not show the addresses of the guard node or client.

Additionally, we used tracing and ShiViz to track the interactions between nodes.

Demo Methodology

All server/coord/node/client will be deployed through ssh using goland remote execution devtool. The tracing server will be started through a remote terminal. Tracing server. coord, and mockRPC server will run on valdes.students.cs.ubc.ca. Onion nodes will run on pender.students.cs.ubc.ca, thetis.students.cs.ubc.ca, and anvil.students.cs.ubc.ca. Clients will run on remote.students.cs.ubc.ca and pender.students.cs.ubc.ca. Onion nodes will be terminated through goland GUI. The tracing log will be found on the remote server under the demo folder.

Demo Outline

1. **Normal Operation (15 mins)**
    a. To demonstrate normal operation of our system, we will first initialize the onion network, by doing the following:
        i. Starting the tracing server
        ii. Starting the coord
        iii. Starting the rpc server
        iv. Starting the 3 nodes
    b. Next, we will start a client, which sends RPC calls in a loop. We will then show that the system is behaving correctly by doing the following:

    c. Lastly, we will terminate all servers, upload the log onto ShiViz web UI, and then show the system is behaving correctly (guarantees 2 and 3) by doing the following.
        i. Showing that the coord assigns the role correctly

    ii. Showing that the client receives an onion chain from coord
    iii. Showing that the client exchanges public keys with guard/relay/exit nodes
    iv. Showing that the value received by the client is the same as the value sent by the server.
    v. Showing that the guard/relay nodes do not show the value sent.
    vi. Showing that the guard node does not show the exit node and server addresses
    vii. Showing that the exit node does not show the guard node and client addresses

2. **Node Failures (10 mins)**
    a. Repeat a and b in demo 1
    b. To demonstrate our system can survive at least 3 node failures, we will first start 3 new servers, and then, we will manually terminate the first guard, relay, and exit nodes.
    c. We will then send terminate all servers. To show that the system is working correctly, we will explain the tracing and terminal output.
    d. Optionally, we can run an existing test on localhost to show that the system can survive constantly failing nodes.
3. **New Nodes (10 mins)**
    a. Repeat a and b in demo 1
    b. To demonstrate our system can join and utilize at least 3 new nodes, we will then join 3 new onion nodes. Then we will terminate/add nodes one at a time until the client starts to send requests through new nodes.
    c. We will then terminate all servers. To show that the system is working correctly, we will explain the tracing and terminal output.
4. **Design Q/A (10 mins)**
    a. We will spend the remaining 10 minutes answering any questions about our system.

Conclusion

We have implemented a simplified onion network that supports jsonRPC calls to a jsonRPC server from a client. The system has three components: a client-side/server-side library for sending/receiving RPC calls over the onion network, a coordinator server which orchestrates the onion network, and a number of nodes. There are three varieties of nodes: guard nodes, which are the first nodes in a circuit and interact directly with the client, exit nodes, which are the final nodes in a circuit and interact directly with the RPC server, and relay nodes, which serve as an intermediate step in the circuit between the guard and exit node. The system is robust to several node failures, as well as being capable of supporting multiple clients.

Reference

diffie-hellman protocal: https://asecuritysite.com/encryption/goecdh
Onion routing: https://www.youtube.com/watch?v=I5FRYpPwpJ0