Noctua: A secure messaging platform

Lucas Givens

October 5, 2022

1 Intro

Due to the advent of the internet, the issue of privacy has become ever more prevalent. Large companies use the lack of technological literacy most citizens posses to their advantage by convincing them to opt into giving up information they didn't even know could be used against them. Google, amazon, apple, Facebook, and even some big box retailers like Walmart collect vast amounts of data that they sell off or abuse. This abuse comes in many forms but one of the most common is using your data to target you with ads or media to increase your use of their services. Every year the amount of data these companies expose for profit grows. Multiple types of technologies have risen to help bring some control back to the average citizen. The Onion Routing project (Tor), a internet networking system, uses algorithms and information hiding techniques to deliver information to websites across the internet without them knowing where the data came from. This keeps websites from tracking data about users. Multiple blockchain technologies like Ethereum have also taken advantage of cryptographic technologies to validate transactions anonymously. Allowing users to pay for items or make exchanges of data securely without the personal information of either party being known, skipping the bank all together. Both of theses systems work using a similar principle called decentralization, which allows for large computer networks to work securely with each other to accomplish a goal. These examples are also open source as well, allowing anyone to connect and run a computer to help grow or use the network.

With Facebook, Google, Microsoft, Apple, and Amazon being responsible for almost all of the online communication channels we use today, it is only a matter of time before they start pulling data from private chats. This would be a huge breach of privacy but does already happen when the governments request information from these organizations. This shows that they have the capability to access this data as well as the motive. Noctua seeks to solve this problem before it starts, implementing peer-to-peer messaging in a secure and decentralized way. By combining the Tor network and blockchain technologies like Ethereum, it can securely move data between devices on the internet without a centralized middle man. This peer to peer structure allows for you to be in complete control of your data, with no entity to step in the middle. Only people you know and have given permission to can message you using proven cryptographic techniques that encrypt your data with personal keys that you can change. Data, stored locally, gives you the agency to completely delete if you feel its necessary, giving you complete control.

2 Literature Review

To adequately prepare for the coming solutions discussed in this paper it is necessary to go over several primary technologies. These being Blockchains and Tor which will be used in the implementation of the proposed decentralized messaging application.

Blockchains are a subsection of cryptographic techniques that chain data together using hashes that prevent unauthorized changes being made to the chain [10]. In the context of this paper blockchains are most often used to share information between large networks of anonymous computers. This allows for a distributed database with high data integrity that only allows for the original owners of the data to change it [10]. This can simulate peer to peer data transfers but is often **too slow** for large amounts of data to be sent or stored.

Tor is a method not for storing information but for sending data anonymously on the internet. The Tor network takes data and wraps it in several layers of encryption, protecting your data from malicious attackers. The data is then sent to a network of computers that reroute your data around between tor servers called routers [6]. No router on the network knows both the sender address and the destination address [6]. This makes tor effective at hiding user information. The coming paragraphs, will go into detail on tor and the specific blockchain that will be used for this paper.

Blockchains used for distributed computing are described in the Ethereum yellow paper[10] as a "simple application on a decentralized, but singleton, compute resource". The Ethereum blockchain takes this idea and uses it to create a generalized block chain that works as one distributed computational resource. Resources are managed via transactions that are signed off on using cryptographic techniques that verify their authenticity []. The method that the Ethereum network uses is known as a Blockchain [10]. The Ethereum blockchain is a ledger of information with each entry being tied to an account known as a "wallet". Each time this ledger is updated all computers on the network verify the authenticity of the update with proof

of stake algorithms. Computers are motivated to update the ledger via payments of a token called "gas" [10] and updates being known as transactions. When a transaction is made the user who initiated it will pay a small fee in gas depending on the size of the transaction and the load the network is under. The nature of the block chains cryptography allows for near absolute certainty of the validity of a transaction [10]. Another well-known blockchain is Bitcoin, which differs from Ethereum in that is can only store information relating to how much bitcoin a wallet owns. Ethereum allows for the storage of many different types of information such as code, which the network is able to use to run decentralized apps. This ability to store predefined behavior that operate securely on a distributed network is why it is the choice blockchain for this paper. The major downside of Ethereum however is its greatest strength. Due to the open nature of the ledger every computer on the network is able to see all the data in the ledger. Transactions can also be limited based on how much stress the network is currently under as well. To complement Ethereum a second technology discussed below.

Tor is a networking protocol that anonymizes internet traffic. It works via encrypting data which is then sent through the tor network. The network is made up of many peer to peer connections that bounce your data around preventing any one sever from knowing where your data is going and where it came from [6]. Tor is an open-source project that was originally created by the US navy to hide secret communications [6]. Since then, it has seen wide adoption by the greater public for accessing network resources like websites anonymously. The core interest of this project in tor is its ability to host resources anonymously. Using Tor nodes as rendezvous servers (Severs that allow clients behind network address translators to communicate) it is possible to host servers located behind firewalls [6].

This secure true peer to peer behavior would allow each user on a communication network to be their own message server. Combining Tor and the Ethereum network allows for decentralized anonymous network traffic to be directed by a decentralized authority that can verify users securely and anonymously. Several other projects have attempted to create similarly secure apps which will be discussed below

D.I.M Or Decentralized Instant Messenger is a software that uses the GunJs framework to implement a distributed messaging app [8]. GunJs provides tools to create Distributed Data Bases (DDB) across many nodes [8]. This allows for network of self-hosted server nodes that store user information. Client nodes on the network are able to read user information from the DDB which allows for logins verification from any device. The main benefit of this model is that it is easily used by users who just need to install the app or access the web portal. However, This structure lacks several features due to the inherited server node client - node model, it is not truly peer to peer even though it is distributed. However this model has no single point of failure communication besides access to the web portal or app and uses encryption methods to maintain and ensure transactions on the DDB are legit. Making it secure and easy to use.

Muhammad S. A.[5] proposes a solution not for a chat app but an approach for patients to transfer data from smart sensors to their doctors securely and without a third party. This works is similar in structure to the proposed solution of this paper. [5] focus on two technologies that allow for a secure peer-to-peer architecture [5] states that they use the Ethereum block chain for "a medium for negotiation and record keeping" and that tor is used for "Delivering data from patients to doctors". This architecture is favorable for the medical setting due to its ability to keep data in the hands solely of the patient and the doctor. [5] uses the blockchain for identity management and thus is able to securely transfer the data between doctor and patient using tor rendezvous servers. This is true peer to peer communication and since its using tor it has the added benefit of hiding user metadata such as origin IP.

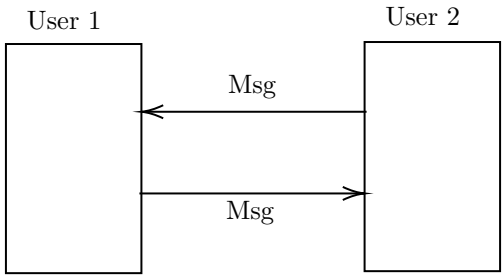
3 Design

The fundamental design approach of this system is to layer several service on top of one another to add features. These layers are as follows; the database, the message server, the Tor hidden service, and the Ethereum authentication service. The goal of the message server is to receive and send http POST requests to other message servers on the network. Each device is used to act as its on message server and client. Using Tor hidden services to bypass the firewall on the local network allows for requests to be sent or received by other servers regardless of local network configurations [7]. To keep track of what address goes to what user an authentication service implemented on the Ethereum network is used to tie users to their TOR hidden service address. The database stores the information of the primary user, the user contacts, and the received messages. Each is designed to work independently of the system below it, allow for more flexibility while implementing the application. In the following sections, the specific design of each layer will be reviewed.

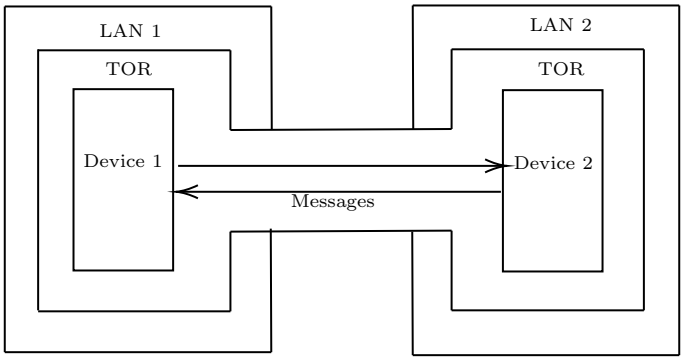
The Database contains several tables: Primary user, Contacts, Conversations, and messages. Primary user is the user of the current instance of the application. Information regarding the address of their TOR hidden server and their username is stored in the primary user table. The Contacts table is used to store the contacts of the primary user, their associated shared keys, and the last updated address of the contact. The Conversations table stores the active conversations, associating the users with the conversations they are apart of. Finally messages are stored in the message table which holds information on the conversation the message is apart of, the message itself, and a time stamp of when the message was received. The message server makes use of the database to store messages and look up conversations to validate received messages.

The Message Server is an http server which receives and sends HTTP post requests and uses SSL for further encryption of the data stream. Requests contain a JSON object with two fields: the conversation hash, and the hashed message. 256SHA is used along with a key shared by each user to decode/hash the data. The conversation hash is used to identify the

conversation that the message belongs to. Using this the unhashed name of the conversation the message server then inserts the message into the database storing the associated conversation id and a timestamp of when the message occurred. To send messages the server again uses POST requests to transmit the data in the same fashion it received it, retrieving destination address from the database. One received the http server will return an OK as an acknowledgement Each user acts as both client and server which forms the peer to peer structure of the network.



Tor hidden services are used to facilitate connections to servers behind NATs without port forwarding in a method know as NAT hole punching [7]. NAT stands for natural address translation, and is used to hide network IP ranges behind a single network address. They are needed because IPV4 only has a set number of IPs that can be assigned in a network. So by splitting larger networks up into their own isolated networks IPs can be reused. Many NATs and firewalls do not allow servers from outside the network to initiate connections inside the network. So for most messaging services like Imessage use centralized servers that devices can call out to retrieve data. Tor can be used to skip this intermediate step by acting as a rendezvous server [6]. Devices behind NATs can reach out to these servers which in turn are able to forward incoming connections to the device. This effectively is what as known as "TCP hole punching" and is what allows devices to act as their own message server. By making a TOR hidden service hosted on the device, incoming http post requests can be received by noctua message servers. The hidden service is separate from the message server and just transfers data between a listening port.



EAS(Ethereum Authentication service) will be used as for identity management and name services. Using smart contracts to allow users to create a username and set an associated address securely. The EAS stores the last update address of the users Tor hidden service. With each username being unique allowing for users to add contacts via usernames like they would with any other service. When a contact is added the application will attempt to initiate a transaction to find the user in the EAS. If found it will add its contact and address to the users hidden service. to keep address fresh, whenever a new hidden service is created, a new transaction will be initiated to update the users hidden service address. If a user sends a message to a contact and that contacts HTTP server does not respond another transaction will attempt to find the new address of the contact. Due to the nature of the Ethereum network transactions are automatically authenticated by the network using the users wallet address [10] meaning only the user can update their own addresses.

4 Implementation

For implementation of the database I chose RoomDB, which is an high level Kotlin library that provides abstraction for creating and managing SQlite databases. RoomDB's can be implemented through several layers: the tables, data access object, repository, and the AppViewModel [3]. The tables are defined using special syntax that correlates to sqlite table qualities like Primary keys and columns. Tables are defined using kotlin data classes using syntax like the following.

Listing 1: Example of table definition.

```
@Entity(tableName = "contacts_table")
data class Contacts (
    @PrimaryKey(autoGenerate = false)
```

```
    val user: String,
    val key: String,
    val address: String
    )
```

Four tables are need for the basic functions of this application: the contact table, primary user table, conversation table, and the message table.

- The contact table has three columns: Contact name(Primary key), key, and last updated address
- Primary user table has two columns: Id(primary key), hidden service address. Only one entry should ever been in this table.
- The conversation table has three columns: ID(Primary key), the contact name, and the conversation name
- The message table has five columns: ID(Primary key), the contact name, conversation name, message, and time stamp

Data Access Objects(DAO) are used to define query's, insertions, deletions, and other SQL activities. Due to the nature of IO like database access it is best to use coroutines to preform database access independently of the Current thread. Coroutines are simple threads that allow for non-blocking code to be easily implemented [2]. This prevents the blocking of UI or of the message server when accessing the database. For the purpose of this application four insertion and four queries are necessary of the base functionality of the app.

Listing 2: Example of DAO definition.

```
@Dao
interface DAO {
    // Adds a contact to the contacts table, takes the contact data class from
    // the Table definition as input and returns nothing.
    // If there is a conflict like a non-unique
    // primary key the function will call a fail exception
    @Insert(onConflict = OnConflictStrategy.ABORT)
    suspend fun addContact(user: Contacts)

    // Adds a primary user to the primary user table.
    // This is where data like username and current hidden address are stored.
    // On conflict the insertion fails and thows a fail exception
    @Insert(onConflict = OnConflictStrategy.ABORT)
    suspend fun addPrimaryUser(user: PrimaryUser)

    // Adds a contact to a coversation by
    // making an entry in the conversations table.
    // This is how contacts are grouped into conversations.
    // Setup in a way to allow for group
    // conversations in the future but currently this is
    // missing feature.
    @Insert(onConflict = OnConflictStrategy.ABORT)
    suspend fun addToConversation(convo: Conversations)

    // Inserts a message into the message database.
    // Messages from all users are
    // stored here even messages from the primary user.
    // They are associated to conversations by the convo ID.
    @Insert(onConflict = OnConflictStrategy.ABORT)
    suspend fun addMsg(msg: Msgs)

    // Selects all msgs that match the conversation ID
    // and returns them ordered by their timestamp.
    // This is used in the UI to
    // show the message history of a coverstation.
    @Query("Select␣*␣FROM␣msg_table␣WHERE␣convo␣=␣:convo␣ORDER␣BY␣time_stamp␣ASC")
    fun getConvo(convo: Int): LiveData<List<Msgs>>

    // Gets all contacts and returns them as a livedata object.
    // Live data objects are update when new data
    // is observed and must be accessed using an observer
    @Query("Select␣*␣FROM␣contacts_table")
    fun getContacts(): LiveData<List<Contacts>>

    // Gets all contacts and returns them as a regular list
    // This is nessacry for when contacts must be
    // checked for validity an observer object would be uneccesary.
    @Query("Select␣*␣FROM␣contacts_table")
```

```
    fun instantGetContacts(): List<Contacts>

    // Gets Information about primary user
    @Query("Select * FROM primary_user_table")
    fun getPrimaryUserInfo(): LiveData<List<PrimaryUser>>

    // Lists all Conversations is used in UI
    // for listing all availble messages
    @Query("Select * FROM conversation_table")
    fun getConvos(): LiveData<List<Conversations>>

}
```

Accessing these functions can be done from the database repository which holds functions that abstract multiple data sources. It acts as an intermediate between the DAO and the database View Model. View models are used for when a database need to be interacted with by UI components. Functions here are define just as normal functions. The syntax for declaring repository functions are as follows.

Listing 3: Example of repository definition.

```
class AppRepository(private val dao: DAO) {

    //Inits variables that can me found on start
    val getConvos: LiveData<List<Conversations>> = dao.getConvos()
    val getContacts: LiveData<List<Contacts>> = dao.getContacts()

    // returns all the messages that have the same
    // message ID orderd by time stamp
    fun getConvo(id: Int): LiveData<List<Msgs>>{
        return dao.getConvo(id)
    }

    // Gets all contacts as a list
    fun instantGetContacts(): List<Contacts>{
        return dao.instantGetContacts()
    }

    // Adds a contact to the contact table
    // Uses the suspend keyword to be used in a coroutines
    suspend fun addContact(user: Contacts){
        dao.addContact(user)
    }

    // Adds a primary user to the primary user table
    suspend fun addPrimaryUser(user: PrimaryUser){
        dao.addPrimaryUser(user)
    }

    // Adds a contact to a conversation table
    suspend fun addToConversation(convo: Conversations){
        dao.addToConversation(convo)
    }

    // Adds a message to message table
    suspend fun addMsg(msg: Msgs){
        dao.addMsg(msg)
    }


}
```

The view model uses calls functions from the repository but calls them inside of coroutines that allow for asynchronous access to the database. This is nessacary because the viewmodel is used by UI componets to interacte with the data base so IO calls should be none blocking to prevent slow UI. The app view model is defined by the following code.

Listing 4: Example of Viewmodel definition.

```
// Takes application as context
class AppViewModel(application: Application): AndroidViewModel(application) {
    private val repository: AppRepository

    val getConvos: LiveData<List<Conversations>>
    val getContacts: LiveData<List<Contacts>>
```

```
    init {
        val appDao = AppDatabase.getDatabase(application).dao()
        repository = AppRepository(appDao)
        getConvos = repository.getConvos
        getContacts = repository.getContacts
    }

    fun addContact(user: Contacts){
        viewModelScope.launch(Dispatchers.IO){
            repository.addContact(user)

        }
    }

    fun addConvo(convo: Conversations){
        viewModelScope.launch(Dispatchers.IO){
            repository.addToConversation(convo)
        }
    }

    fun getMsgs(id: Int): LiveData<List<Msgs>>{
        return repository.getConvo(id)
    }

    fun instanctGetContacts(): List<Contacts>{
        return repository.instantGetContacts()
    }

}
```

In conclusion these are the necessary database functions to implement the base functions of the messaging app. If additional feature where to be added it is likely there would need to be additional changes to the implementation. For example the current design is centered around single user conversations, If group conversation where to be implemented. There would need to be queries for gather all users in the shared conversation to a list. In the next section the implementation of the HTTP server and how it interacts with the database will be outlined and expanded upon.

The http server was implemented using a kotlin library called "Ktor" [4], which is a light kotlin library for creating http servers and clients. Ktor uses coroutines to allow for asynchronous receiving and sending of http requests. For encryption it can use SSL(Secure Sockets Layer) to allow for messages to be sent securely adding an extra layer to the AES256 Encryption that is done when messages are sent. SSL uses private and public certificates to provide asynchronous encryption between servers and clients. Ktor allows for all standard http messages, but the only used for this implementation will be the Post request. The two methods implemented to handle these requests will be the "/txt" http route and the send() function. Below is the Psuedocode for each function.

Listing 5: Psuedocode for /txt.

```
data class Jsonmsg(
        convoEncrypt: String,
        msgEncrypt: String
)

routing{
        // sets the name of the route
        // accessed at http://IP_ADDRESS:PORT/txt
        post("/txt"){

                // When /txt recieves a post request call.recieve
                // retrieves the recieved post data
                lastmsg = call.recieve<Jsonmsg>()

                // Launch a coroutine to prevent the
                // blocking of other messages by IO activities.
                CoroutineScope(Dispatchers.IO).launch {

                        // Retrieves a List of all contacts store by the user
                        // and then iterates over all contacts to see if a
                        // contact key is able to decrypt the convoEncrypt.
                        // Each decryption is checked to
                        // see if it is the name of a conversation
                        // That currently exists. If it does
                        // it adds a message to the database marked with that messageID

                        val contacts = repository.getInstantContacts()
```

```
                        for contact in contacts {
                                convo = decryptAES256(contact.key, lastmsg.convoEncrypt)
                                if(repository.convoexists(convo){
                                        msg = decryptAES256(contact.key, lastmsg.msgEncrpyt)
                                        repository.addmsg(msg(0, convo, msg, timestamp))
                                }
                        }
        }

        }
}
```

Listing 6: Psuedocode for send().

```
        send(msg: String, convoName: String){
                // Find the contact from the contact name
                // and save its keyhash
                contact = repository.findContact(convoName)
                keyhash = contact.key

                // Using AES256 encoding encrpyt the convoName
                // and the msg with the keyhash
                convoEncrypt = AES256(keyhash, convoName)
                msgEncrypt = AES256(keyhash, msg)

                // attempt to post the JSON message using ktor
                // and save the response
                response = http.POST(contact.address){
                        contentType(JSON)
                        setbody(Jsonmsg(convoEncrpyt, msgEncrpyt)
                }

                // Check if the response was OK if not then post
                // "MSG NOT SENT" to approriate chat.
                if(response = http.OK){
                        // Add the primary users msg to the conversation
                        // and add to database
                        repository.addmsg(msg(0, primary.name, convoName, msg))
                } else {
                        repository,addmsg(msg(0, primary.name, convoName,
                        "UNABLE TO SEND TRY AGAIN"))
                }

        }
```

TOR for this project will implement hidden services using Tor Onion Proxy Library a java library [9] developed under Microsoft open source as part of the Thali project [1]. The Thaili project is a group working towards advancing technologies involved in blue tooth and wifi direct networking. The goal of the Tor Onion Proxy Library is to ease the management of TOR proxies and allow each app to act as its own tor proxy. This allows for TOR to be backed into an app instead of relying on third parties like Orbot to create a hidden service. The code for createing the TOR hidden service is relatively simple as little logic is needed to create the service. The code below is sampled from the github for the Tor Onion Proxy Library [9].

Listing 7: Psuedocode for send().

```
String fileStorageLocation = "torfiles";
OnionProxyManager onionProxyManager =
        new AndroidOnionProxyManager(this.getApplicationContext(), fileStorageLocation);
int totalSecondsPerTorStartup = 4 * 60;
int totalTriesPerTorStartup = 5;

// Start the Tor Onion Proxy
if (onionProxyManager.startWithRepeat(totalSecondsPerTorStartup, totalTriesPerTorStartup, true) == false) {
    Log.e("TorTest", "Couldn't start Tor!");
    return;
}

// Start a hidden service listener
int hiddenServicePort = 80;
int localPort = 9343;
String onionAddress = onionProxyManager.publishHiddenService(hiddenServicePort, localPort);

Socket clientSocket =
        Utilities.socks4aSocketConnection(onionAddress, hiddenServicePort, "127.0.0.1", localPort);
```

References

[1]

[2] Concurrency and coroutines: Kotlin.

[3] Save data in a local database using room android developers2022, 2022.

[4] Welcome: Ktor, 2022.

[5] Muhammad Salek Ali, Massimo Vecchio, Guntur D. Putra, Salil S. Kanhere, and Fabio Antonelli. A decentralized peer-to-peer remote health monitoring system. *Sensors*, 20(6), 2020.

[6] Roger Dingledine, Nick Mathewson, and Paul Syverson. Tor: The second-generation onion router. Technical Report ADA465464, NAVAL RESEARCH LAB WASHINGTON DC, 555 Overlook Ave SW, Washington, DC 20375, January 2004.

[7] Daniel Maier, Oliver Haase, Jürgen Wäsch, and Marcel Waldvogel. A comparative analysis of nat hole punching. *HTWG FORUM - Das Forschungsmagazin der HTWG Konstanz*, pages 40–48, 2011.

[8] Sourabh. Inderprastha engineering college. *International Research Journal of Engineering and Technology*, May 2020. Accessed September 7, 2022.

[9] Thaliproject. Thaliproject/tor
$_{o}nion_proxy_library : Providesajarandanaarforembeddingthetoronionproxyintoajavaorandroidprogram, Jun$2014.

[10] Gavin Wood. Ethereum: A secure decentralised generalised transaction ledger.