RoomDB was used for this papers implementation of the database. RoomDB is an high level Kotlin library that provides abstraction for creating and managing SQlite databases. It is implemented through several layers: the tables, data access object, repository, and the AppViewModel [?]. 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 [?]. 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 throw 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 with the exception of using the suspend key word. The suspend key work tells the JVM that the function will be called inside of a coroutine and that it can be a blocking function. The repository uses functions declared in the dao and returns their values. 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 interact with the data base so IO calls should be none blocking to prevent a slow UI. The app view model used for this implementation 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()
    }

}
```

These are the necessary database functions to implement the base functionality of the messaging app. If additional features where to be added it is likely there would need to be 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 to 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 is implemented using a kotlin library called "Ktor" [**?**], 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) allowing 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 one used for this implementation will be the Post request. The two methods implemented to handle these requests are 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, implement of the hidden service was done using kpm-tor [?] which a branch of the Tor Onion Proxy Library [?]. The goal of the kmp-tor 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 baked into an app instead of relying on third parties like Orbot to create a hidden service. The TOR proxy listens to the port that the HTTP server runs on and broadcasts it on the rendezvous server on another port. First, the configuration of the tor server must be done by setting all the ports that it will forward. Then the hidden service is configured and its ports are directed to the ports of the http server, so the messages cant be forwarded. Below is an example of the configuration of the hidden service. The rest of the code can be found at the github for this papers project the /tor package [?].

Listing 7: Psuedocode for hidden service.

```
put(HiddenService()
                        .setPorts(ports = setOf(
                            // Use a unix domain socket to communicate via IPC instead of over TCP
```

```
                HiddenService.UnixSocket(virtualPort = Port(80), targetUnixSocket = hsPath.builder {
                    addSegment(HiddenService.UnixSocket.DEFAULT_UNIX_SOCKET_NAME)
                }),
            ))
            .setMaxStreams(maxStreams = HiddenService.MaxStreams(value = 2))
            .setMaxStreamsCloseCircuit(value = TorF.True)
            .set(FileSystemDir(path = hsPath))
        )

        put(HiddenService()
            .setPorts(ports = setOf(
                HiddenService.Ports(virtualPort = Port(80), targetPort = Port(5001))
            ))
            .set(FileSystemDir(path =
                workDir.builder {
                    addSegment(HiddenService.DEFAULT_PARENT_DIR_NAME)
                    addSegment("test_service_2")
                }
            ))
        )
    }.build()
```

Implementation of the **ENS** is done using Solidity which is a langauge that supports compilation into byte-code that can run on the Ethereum Virtual Machine(EVM) [**?**]. The EVM is similar to the Java Virtual Machine in idea, as it allows for many machines with different architecture to run the same code. When publishing a smart contract to the Ethereum network, it is first compiled into bytecode and is then uploaded and given an address space in the Ethereum ledger [**?**]. Using this address and web3j, a java library that allows users to link their Ethereum wallet to android apps, the application can call the functions of the smart contract. This will allow users to manipulate their data on the ethereum block chain and provide a way for users to query for other users addresses. To facilitate these interactions the smart contract consists of a struct, a dictionary object with String keys and values of instances of the struct, and four functions. Below is the pseudocode for creating the smart contract as well as comments explaining more on the fucntions.

Listing 8: Psuedocode for ENS.

```
struct User {
        String username;
        String address;
        String owner; // Wallet address of the owner of the username
}

// Creates a dictionary object that contains keys
// of the string type that will be
// equal to the username of the user
// and values that will be equal
// to instances of the User struct
mapping(String => User) public ENS;

// Function to add a user to the ENS map
// First check if user exists if not
// add user
function addUser(username: String, address: String){
        if( EAS[username].exists ){
                return "Transaction failed username exists"
        }
        else {
                user = User(username, address, msg.sender)
        {
}

// Attempts to update the address of a user.
// Will not update unless username exists
// and username is owned by sender
function updateAddress(username: String, newAddress: String){
        if( !EAS[username].exists or msg.sender != EAS[username].owner){
                return "Transaction failed because username \
                doesn't exist or msg.sender is not owner"
        }
        else {
                EAS[username].address = newAddress
        }
}

// Looks up a user in the ENS and
// returns address of user
function lookupUser( username: String ){
        return EAS[username].address
}

// Deletes a user from the mapping
deleteUser(username: String){
        if( EAS[username].exists and msg.sender == EAS[username].owner){
                delete EAS[username]
        }
}
```

In conclusion the implementation of the core components of this system are outlined above. The database stores the data captured from the primary user and from incoming messages. While the HTTP server handles the incoming HTTP messages from other users and attempts to add them to a conversation if they are valid messages. The TOR hidden service allows the HTTP server to be accessed by users outside of the local network, and the ENS provides the application with a decentralized way for users to find each other.