Software System Documentation

for

CipherChat

A Secure, Decentralized Chat Application

Version 1.0

Prepared By

Ottor Mills	ID#: 180917	Email: 180917@gist.edu.cn
David Thomas	ID#: 180912	Email: 180912@gist.edu.cn
Nicoy Smith	ID#: 180902	Email: 180902@gist.edu.cn
Kenneth Anglin	ID#: 180907	Email: 180907@gist.edu.cn

Course Instructor: Thomas Canhao Xu

Course: SWEN3010

Date: April 30, 2019

Table of Contents

Overview

CipherChat is a decentralized and secure chat application that was created on the premise that everyone should be entitled to their privacy, especially while using the internet. It is free to use and open source which therefore means that the application's source code can be peer-reviewed by users around the world and be deemed as secure; this is the main advantage of open source software. It should also be mentioned that CipherChat is intended to have no real owners. Although the coders were the only ones who partook in CipherChat's creation, these coders are NOT the owners. This is the key distinction between CipherChat and Telegram. Telegram is privately owned therefore some level of trust is required between Telegram users and the owners, even though its source-code is open source. CipherChat on the other hand is maintained by users who wish to set up their own nodes or random strangers. Using multiple cryptographic techniques, CipherChat creates a dependable network of interconnected nodes in which only the rightful recipient messages know the contents of the message by decrypting it. In other words the servers in a CipherChat network primarily stores encrypted messages for clients.

Figure 1.0 shows the distinction between three popular network structures

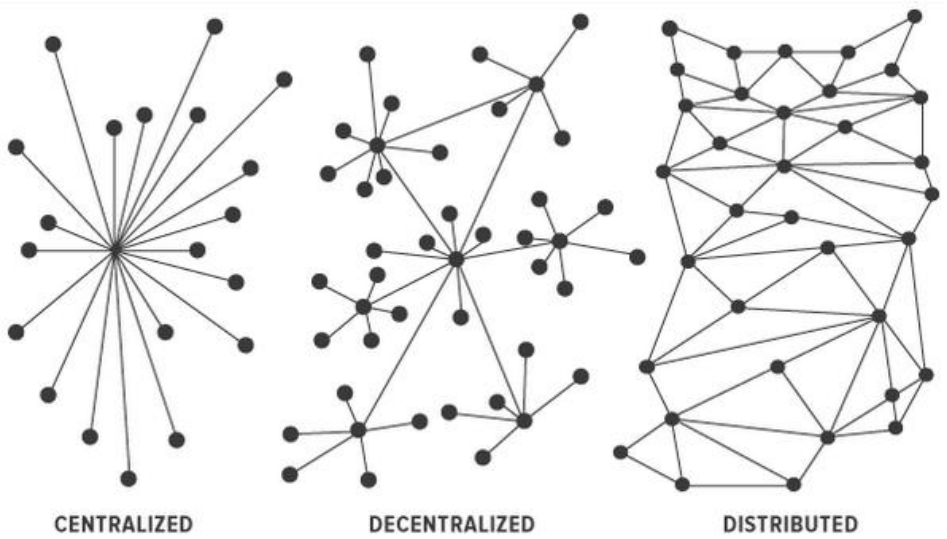

Figure 1.0

The CipherChat project makes use of a total of five languages. They are:

TypeScript	Used to create some server-side components which were transpiled into JavaScript
JavaScript	Main language used to create the server using the Express framework. It is also used as a supplementary language of the mobile application by implementing features not provided by Dart.
Dart	Main language used to create the mobile application using the Flutter framework. Flutter allows user interfaces and functionalities to be created using well defined classes / objects instead of using XML
SQL	The server uses the MYSQL database where as the mobile application uses an SQLite database. Both these databases require SQL in order to be interfaced
Bash	Used to create the operating scripts of the server, in addition to providing additional functionalities of the server such as SSL certificate generation

Server Side Implementation

The CipherChat server was written using the Express framework. The Express framework it excels at handling repetitive, high-traffic but low intensity tasks, making it ideal for chat applications. Incoming requests are immediately handled as a background task (on the same thread but asynchronously) while the server continues to listening for more incoming requests. The CipherChat server utilizes load balancing which is provided by the loadBalancer.js file. Using load balancing incoming requests are immediately sent to one of many server instances running on separate threads.

These requests are subsequently pushed as a background tasked and handled as mentioned earlier, maximizing efficiency. More servers may be added to a single machine (vertical scaling) and more machines may be added with more server processes running (horizontal scaling). The IP address of additional machines would be added to the "remoteServersUrls" array in the server's config.json file.

Figure2.0 CipherChat's Server File Tree (only salient files are shown)

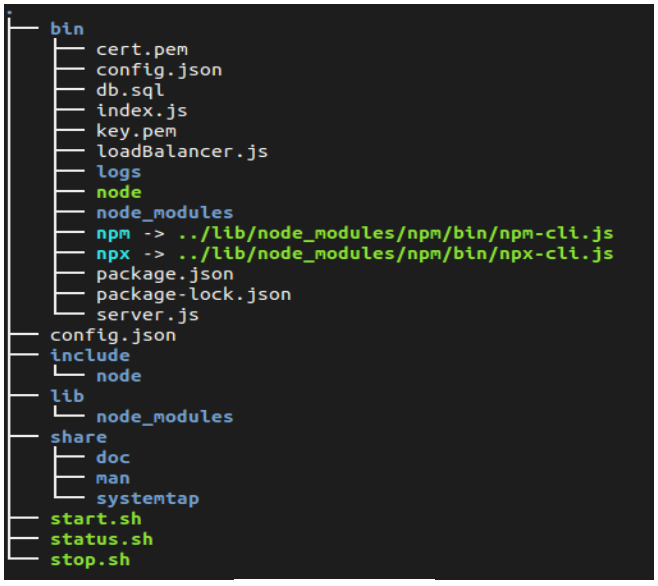

Figure 2.0

Figure 2.1 CipherChat's Server's configuration file

```
{
    "serverIp": "<Server Ip Address Here>",
    "autoIpDetection": true,
    "showAdvertisments": true,
    "admodId":"ca-app-pub-3940256099942544/6300978111",
    "enableHTTPS": true,
    "keyPath":"./key.pem",
    "certPath":"./cert.pem",
    "sha256Password": "e3b0c44298fc1c149afbf4c8996fb92427ae41e4649b934ca495991b7852b855",
    "maxParticipantsPerGroup": 100,
    "port": 6333,
    "instanceServerStartingPort": 3000,
    "numberOfLocalhostServers": 3,
    "remoteServerUrls": [

    ],
    "databaseConfig":{
        "host":"localhost",
        "user":"root",
        "password":"",
        "database":"cipherchat",
        "port": 3306
    }
}
```

Figure 2.1

The config.json file acts as an interface for making quick changes to the server. Below is a description of each field:

serverIp	The public IP address of the server can be manually set using this field.
autoIpDetection	A Boolean value which determines if the server should automatically get the public ip address of the server using and online API
showAdvertisments	A Boolean value which determines if the app should display ads. Ads on the chat screen. (Currently not supported by the app)
admodId	The desired admod ID from which to serve ads
keyPath	The path to the SSL private key used by the server in HTTPS
certPath	The path to the certificate file that is to be distributed among connected peers for HTTPS
sha256Password	The hashed password required by clients to create a new groupd on

	the server
maxParticipantsPerGroup	The maximum number of participants allowed to join a group
port	The port number of the server which is available to the public
instanceServerStartingPort	The starting port number of each load balanced server instance. The following port number are incremented
numberOfLocalhostServers	The number of load balanced server instances
remoteServerUrls	An array containing the private IP addresses of any server instances that reside on other machines
databaseConfig	A JSON object containing the configurations of the mysql database to be used
databaseConfig.host	The IP address of the mysql database
databaseConfig.user	The username of the database
databaseConfig.password	The password for the username of the database
databaseConfig.database	The name of the database to use
databaseConfig.port	The port number of the mysql database (default: 3306)

API Endpoints

The following are the endpoints of the server's REST API that attach to the server url (https://127.0.0.1:6333/ if launched locally and using the default port). Server routes are found in the server.js file.

Endpoints	Details
/	Parameters: Type: GET Response: "HELLO WORLD" Description: Used to test the web service
/newgroup	Parameters: username : string publicKey : string publicKey2 : string passphrase : string Type: POST Response: joinKey : string Description: Used to create a new group for which to add participants to. Returns a token which is used to add participants to the group
/joingroup	Parameters: encryptedMessage : string signature : object : { "r": string "s": string

	“recoveryParam”: int } username : string publicKey : string publicKey2 : string joinKey : string Type: POST Response: errorCode : string Description: Used to join a group. An error code is returned once processing of the request body has been completed

/isusernametaken	Parameters:
	encryptedMessage : string
	signature : object : {
	“r”: string
	“s”: string
	“recoveryParam”: int
	}
	username : string
	joinKey : string
	
	Type:
	POST
	
	Response:
	isTaken : string
	
	Description:
	Checks the database to see if the provided username for a particular group chat is unique. Returns a integer as a string representing true or false with 1 being true and 0 being false.

/message	Parameters:
	encryptedMessage : string
	signature : object : {
	“r”: string
	“s”: string
	“recoveryParam”: int
	}
	joinKey : string

	username : string
	compositeKeys : string
	
	Type:
	POST
	
	Response:
	insertionStatus : object : {
	“mid”: int,
	“timestamp”: int
	}
	
	Description:
	Used to send messages to a group. Returns the message status once the message has successfully been inserted into the database.

/messages	Parameters:
	encryptedMessage : string
	signature : object : {
	“r”: string
	“s”: string
	“recoveryParam”: int
	}
	joinKey : string
	username : string
	offset : string (message ID offset)
	
	Type:
	GET
	
	Response:
	messages : object : {
	messageId : object : {
	sender : string,
	encryptedMessage: string,
	compositeKey: string,
	ts: int
	},
	messageId2 : object : {
	sender : string,
	encryptedMessage: string,
	compositeKey: string,
	ts: int

	}, … } Description: Used to get messages of a particular group from the server.

/anynewmessages	Parameters:
	encryptedMessage : string
	signature : object : {
	“r”: string
	“s”: string
	“recoveryParam”: int
	}
	joinKey : string
	username : string
	offset : string (message ID offset)
	
	Type:
	GET
	
	Response:
	hasPendingMessages : boolean
	
	Description:
	Used to check the server if any messages are unread by the user.

/participants	Parameters:
	encryptedMessage : string
	signature : object : {
	“r”: string
	“s”: string
	“recoveryParam”: int
	}
	joinKey : string
	username : string
	
	Type:
	GET
	
	Response:
	participants : object : {
	username : object : {
	publicKey : string
	publicKey2 : string

	joined : int
	},
	
	username2 : object : {
	publicKey : string
	publicKey2 : string
	joined : int,
	}
	…
	}
	
	Description:
	Used to get the participants of a group chat

Verbose Database Schema (SQL)

```
CREATE DATABASE cipherchat;
USE cipherchat;

CREATE TABLE IF NOT EXISTS groups(
   gid INT(11) NOT NULL AUTO_INCREMENT,
   joinKey VARCHAR(100) NOT NULL,
   ts TIMESTAMP DEFAULT CURRENT_TIMESTAMP NOT NULL,
   UNIQUE(joinKey),
   PRIMARY KEY(gid)
);
CREATE TABLE IF NOT EXISTS participants(
   pid INT(11) NOT NULL AUTO_INCREMENT,
   gid INT(11) NOT NULL,
   username VARCHAR(255) NOT NULL,
   publicKey VARCHAR(100) NOT NULL,
   publicKey2 VARCHAR(100) NOT NULL,
   ts TIMESTAMP DEFAULT CURRENT_TIMESTAMP NOT NULL,
   UNIQUE(gid, username),
   PRIMARY KEY(pid),
   FOREIGN KEY(gid) REFERENCES groups(gid)
);
CREATE TABLE IF NOT EXISTS messages(
   mid INT(11) NOT NULL AUTO_INCREMENT,
   gid INT(11) NOT NULL,
   pid INT(11) NOT NULL,
   message VARCHAR(300) NOT NULL,
   ts TIMESTAMP DEFAULT CURRENT_TIMESTAMP NOT NULL,
   PRIMARY KEY(mid),
   FOREIGN KEY(pid) REFERENCES participants(pid)
);
CREATE TABLE IF NOT EXISTS compositeKeys(
   cpid INT(11) NOT NULL AUTO_INCREMENT,
   mid INT(11) NOT NULL ,
   gid INT(11) NOT NULL,
   pid INT(11) NOT NULL,
   compositeKey VARCHAR(255) NOT NULL,
   ts TIMESTAMP DEFAULT CURRENT_TIMESTAMP NOT NULL,
   UNIQUE(mid, pid),
   PRIMARY KEY(cpid),
   FOREIGN KEY(mid) REFERENCES messages(mid),
   FOREIGN KEY(gid) REFERENCES groups(gid),
   FOREIGN KEY(pid) REFERENCES participants(pid)
);
```

Server Testing

It is salient that a web service is able to withstand heavy load. To get a very basic understanding of how the server withstood “heavy” load a stress tester script was created. Below is a screenshot of the code:

```
const command = function(botID){
  return `
output`+botID.toString()+`=$(curl -d \
"username=bot`+botID.toString()+`&\
publicKey=872648736492384792387498&\
publicKey2=080234803249832432423809&\
passphrase=" -X POST https://127.0.0.1:6333/newgroup -k); printf "Output from \
bot`+botID.toString()+`: "$output`+botID.toString()+`'\n';
`;
}

const intensityLevel = 10;
const pulseIntensity = 2;
//Endpoint stress test
(async () => {
  for(let x = 0; x < intensityLevel; x++){
    setInterval(() => {
      let pulseArr = [];
      for(let y = 0; y < pulseIntensity; y++){
        pulseArr.push(new Promise(function(resolve, reject){
          require("child_process").exec(command(x), function (error, stdout, stderror) {
            if (error)
              console.log('[ERROR]: ' + error);
            else
              console.log(stderror+"\n"+stdout);
            resolve();
          });
        }));
      }
      Promise.all(pulseArr);
    }, 600);
  }
})();
```

Explanation:

The code above generates snippets of bash code which are run using sub-processes. These snippets of code reference the command line utility curl to make requests to the server. The intensity level determines how many concurrent processes or “requestors” are used and the pulse intensity is used to specify how strong each request should be. Each pulse is executed in parallel using Promise all, which magnifies the strength of the request.

Testing Methodology

In order to obtain more accurate data three tests were ran. All tests consisted of spamming the server using the script provided in the previous section. The first test resulted in significant slowdown of the web service whereas in the final two tests little to no slowdown occurred. The results were then averaged. The overall test is regarded as basic because it was only executed from a single device. It was just used to get a basic understanding of how the server is able to perform with regards to its variables such as the server's hardware specifications and size of the span attack. This information in combined with mathematics can produce a very preliminary approximation of how many resources are necessary to meet the demand of a global market can be calculated.

Client Size Implementation

Figure 2.2 CipherChat's Mobile Application File Tree

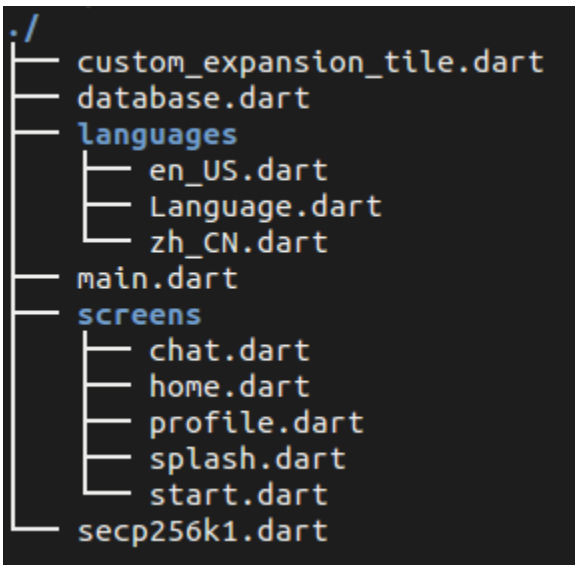

Figure 2.2

The diagram above illustrates the files that were created inside the lib folder of Flutter's default project tree. The following files were used by the project:

- main.dart
- chat.dart
- home.dart
- start.dart
- database.dart
- secp256k1.dart

The main.dart file contains variables, functions and classes that are globally accessible to other dart files of the project. It also defines the screen routes of the application. The chat.dart file contains the layout of the chat screen from which messages is send and received, in addition

to containing functions that aid connecting to the server. The home.dart file contains the layout of the home or index screen. It contains a list of all past conversations. The start.dart file contains the layout for joining or creating a new chat. The database.dart file container the Database class which contains all the necessary methods used to communicate with the the local sqlite database. The secp256k1.dart file contains the methods used in elliptic curve cryptography such as primary key generation and message signing. All other dart files that have not been listed are not yet implemented.

Classes

Below is a listing of all the salient classes and description used in the CipherChat chat application

Home	The Home class contains the layout of the Home or index screen of the cipher chat application.
Language	The Language class contains a reference to language files on the
DatabaseManager	Contains the methods that enable between the application and the SQLite database
Profile	The Profile class is responsible for displaying the profile screen and loading all the information related to the user
Chat	The Chat class is responsible for loading the chat screen. It contains methods used for sending messages as well as a polling method that receives messages in real-time
Start	The Class responsible for loading the connect-to screen. It provides the user fields to enter the ip address and port of the server that they would like to connect to, as well as a join option to join an existing chat by providing a valid joinKey
Secp256k1	This Class contains all the methods necessary for encryption, decryption and signing of messages through the use of a private and composite keys

Future Implementations

Each of the following features may be implemented in future releases of CipherChat:

1. **WebSockets**

   CipherChat is currently uses short polling to retrieve new messages. Although each request only required a few bytes, the total amount of data sent adds up over time, using bandwidth unnecessarily.

2. **Restricted Key Exchange**

   Keys are currently exchanged through the use of a server, however this opens up the possibility of Man-In-the-Middle. In a future release of CipherChat public key exchange will only be allowed to through the use of QR Codes.

3. **Additional Hosting Incentives**

   Additional incentives will be implemented in order to encourage more CipherChat servers to come online.

4. **Notifications**

   Notifications will be implemented to notify the user of the arrival of new messages which is conventional among modern messaging applications

5. **Multilingual**

   CipherChat is currently available in English only however it will support multiple languages overtime

6. **Sending of Files**

   CipherChat will support multiple file formats in future releases.

7. **Broadcasts**

   In a future release CipherChat servers will connect to other nodes and broadcast and receive messages. This will allow users to receive and send messages regardless of which server they connect to.

8. **DDOS Protection**

   Distributed Denial of Service Attacks costs corporations millions of dollars annually. Since anyone can make requests to servers for free, a public server immune to DDOS attacks is next to impossible; although certain measures may be taken to maximize the safety of a web service. The CipherChat network, as it is intended to be, is vulnerable to DDOS attacks. Securing a decentralized network from DDOS attacks is a daunting challenge. Research will be done as to how to secure the CipherChat network, one possible solution is through the implementation of a Blockchain or Tangle.

9. **Migration to TypeScript**

   In non-typing languages such as JavaScript and Python, it becomes harder to debug errors, in other words more time is spend debugging when an error arises. Typed languages on the other hand such as Typescript and Java are specially designed for large applications thus they scale better than non-typed counterparts.