ST AUGUSTINE'S
COLLEGE - SYDNEY

Nodepaths
Design Documentation - Task 2

By Alex Greig

May 12, 2021

Contents

1 Project Portfolio

1.1 Problem Definition

The problem with today's internet is the control large corporations have over information transmission, enabling data to be lost, monetized, or given to the government, all without users knowing. Another problem of the internet is the inherent security flaws associated with centralised control; that is, its ability to be hacked. The final problem is the inequality of the internet as large populations throughout the world are unable to access the internet, with access they would be able to receive better educational resources, progressing humanity forward.

The application that I am creating is called Nodepaths, and it is important for the consumer market as it will provide high security to user's data and a network that is robust and resistant to problems allowing for secure data transfer between devices globally, solving the problems above. The application will have a significant effect on the way we use the internet and allow users to share, communicate and complete complex computational tasks efficiently and securely. The application will be a decentralised network that uses peer-to-peer architecture and mesh topology, allowing any machine capable of connecting to a network to join and become a "node". The application will also include the feature of Wi-Fi peer to peer transmission allowing for nodes to be connected by Wi-Fi radio waves eliminating costs of telecommunication services, however, if users cannot be connected by the mesh then cellular internet is available. This information will be encrypted, end to end. The mesh network will relay data and messages using a wireless ad-hoc network, where data is propagated along a path by hopping from node to node until it reaches its destination. The paths will be calculated based on the Ad-Hoc On Demand Vector Routing protocol (AODV), a routing protocol that determines and maintains routes whenever they are required.

The application that I am creating, Nodepaths, will have three main functions. The first is creating a user, to do this it will need to create identification. The application will create a public and private key using the Curve25519 elliptic curve in conjunction with the Diffie–Hellman key agreement scheme and the Advanced Encryption Standard, then a psuedo-random IPv4 Address and finally validate a username that has been entered by the user. The second function of Nodepaths is to transfer data, normally messages, over a wireless ad-hoc network to another node. The final main function of the application will be to give users the ability to add new nodes (friends) to the application allowing them to select friends that they want to text to. These three functions when integrated will provide the foundation for a user-friendly text messaging application that runs on a decentralised, distributed network. Although the functions may seem simple or basic the underlying backend behind a network of this caliber is complex; to function, innovative solutions are needed.

Future improvements and upgrades to the application will include the implementation of Blockchain technology to provide digital transactions and a cryptographic wallet on the application, eliminating the need for banks and other financial institutions. In the future, the application may also provide assistance to disadvantaged communities as it would give them communication without the cost of telecommunication services.

1.2 Context and Data-flow Diagrams

Figure 1: Context Diagram

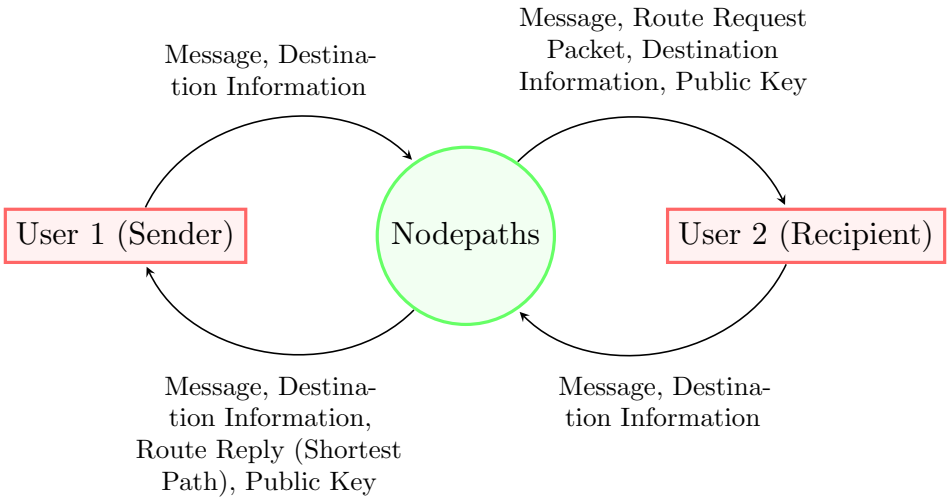
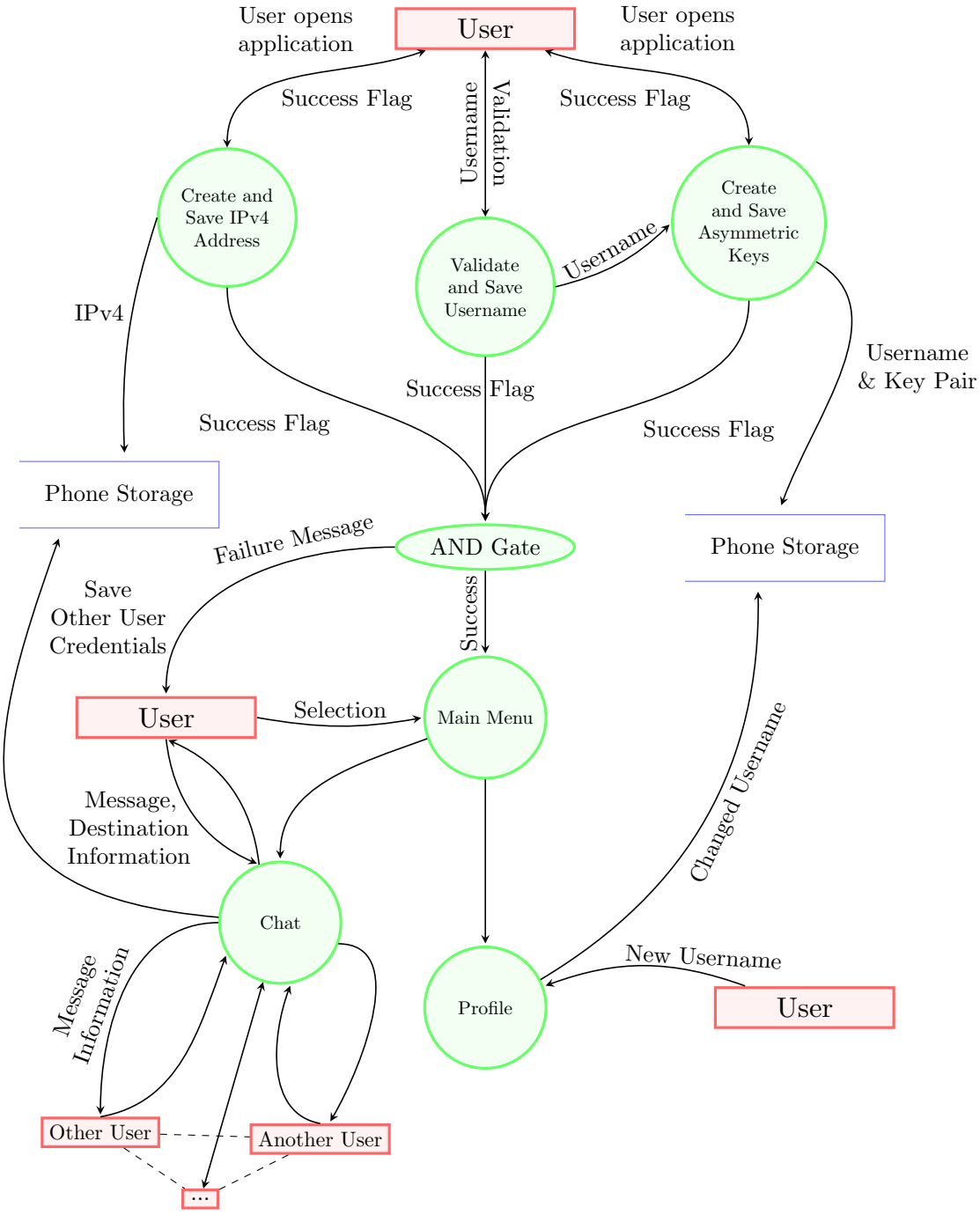


Figure 2: Data Flow Diagram



1.3 Structure Chart

1.4 Algorithms

The core algorithms that will be used in the application are displayed below, written in the pseudocode syntax:

Algorithm 1: Main Program

```
BEGIN MAINPROGRAM
    username, keys, ipv4 = LoadNode()
    DISPLAY "Enter another Node's Username to Message or type Quit
        to Exit"
    INPUT Selection
    WHILE Selection <> "Quit" DO
       Display "Please Input your message: "
       Input Message
       Reply = SendMessage(Message, TargetUser, Reply, keys[1]) //
           keys[1] is the secret key
       Display Reply
       DISPLAY "Enter another Node's Username to Message or type
           Quit to Exit"
       INPUT Selection
    END WHILE
END MAINPROGRAM
```

The main program above displays different processes, inputs and outputs that would be in the real program, however, this code isn't a full replication of the intended end product. If I was to create this application in full I would have more functions that separate the program into smaller pieces, making it easier to read and write. The first difference from the code above is I would separate all the screens into different functions however as previously mentioned this is impractical for the intention of the task.

Algorithm 2: Load Node

```
BEGIN LoadNode(username, keys, ipv4)
    username = ""
    keys = a fixed size array of allowing two byte arrays of
        length, 32 // [[u8; 32]; 2]
    ipv4 = ""
    OPEN "Phone Storage" for INPUT and OUTPUT
       // Checking if config file exists, False if user is opening
            for the first time.
       IF "PhoneStorage/config.toml".exists() == True DO
          lines = READ lines from "Phone Storage/config.toml" into
              an array
          //Below is cycling throught the node config file and
              loading the values into the program from storage
          username = lines[0]
          keys = lines[1].split[","]
          ipv4 = lines[2]

       ELSE DO
          CreateFile("PhoneStorage/config.toml")
```

```
16          INPUT unvalidated_username
17          //Validates Length
18          success = False
19          WHILE success == False DO
20             IF unvalidated_username.len() >= 5 OR
                   unvalidated_username.len() <= 20 THEN
21                success = True
22                username = unvalidated_username
23             ENDIF
24          END WHILE
25          ipv4 = CreateIPv4()
26          keys = CreateEccKeys(secretKey, publicKey)
27       END IF
28       WRITE username, ipv4, keys
29    CLOSE "Phone Storage"
30    RETURN username, keys, ipv4
31 END LoadNode
```

Algorithm 3: Elliptic-Curve Cryptographic (ECC) Key Generation

```
1  BEGIN CreateECCKeys(secretKey, publicKey
2
3     curve = get_curve('curve25519')
4     k = rand_num(0, 627710173538668) //Parameters is the starting
          and ending range of the random number
5     secretKey = k
6     publicKey = privKey * curve.basepoint
7
8     RETURN secretKey, publicKey
9  END CreateECCKeys
```

The Elliptic-Curve Cryptographic Algorithm is efficient and as fast as simply creating a random number, however, it allows for a private and public key to be created. The algorithm itself is based on an elliptic curve which are created through the equation:

$$y^2 = x^3 + ax + b$$

Due to this equation given a curve which has infinite points, we will limit the curve to a finite field. To do this we use modular arithmetic which transforms the equation into the form:

$$y^2 = x^3 + ax + b(mod\ p)$$

where p is a prime number between 3 and 2^{256}. This is a general equation for an elliptic graph however certain specific equations have been named such as secp256k1 (used in Bitcoin), which is

$$y^2 = x^3 + 7(mod\ 17)$$

To check whether a point, for example $P(3, 4)$, lies on this curve you substitute values in to x and y in the equation. A point G over an elliptic curve can be multiplied by an integer k and the result is another EC point P on the same curve and this operation is fast. This is the basis behind the key generation, we multiply a fixed EC point G (the generator point or base point) by certain integer k (k can be considered as private key), we obtain an EC point P (its corresponding public key). It is very fast to calculate $P = k * G$ and extremely slow (considered infeasible for large k) to calculate $k = \frac{P}{G}$.

Algorithm 4: IPv4 Creation

```
BEGIN IPv4Gen(ip)

ip = "192.168"
FOR i = 0 to 1 DO
   APPEND random_int(0, 255) to base_ip
NEXT i
END FOR
RETURN ip
END IPv4Gen
```

Algorithm 5: Send Data Packets over Network

```
BEGIN SendMessage(Message, User, Reply, secretKey)
   transmitter = WifiPeertoPeerApi()
   transmitter.perform_handshake(User)
   transmitter.send_udp_packet(RREQ, User) //finding the shortest
        path to the designated user
   timeout_time = 10
   path = transmitter.reply(timeout_time)
   transmitter.send_udp_packet(Request_Public_Key, User, path)
   their_public_key = transmitter.reply(timeout_time)
   shared_secret_key = diffie_hellman(secretKey, their_public_key
       )
   encrypted_message = encrypt(Message, shared_secret_key)
   transmitter.send_udp_packet(Message, User, path)
   encrypted_reply = transmitter.reply(timeout_time)
   Reply = decrypt(encrypted_reply, shared_secret_key)
   RETURN Reply
END SendMessage
```

1.5 Test data and expected outputs

Algorithm 6: Rust Tests

```
1
2  #[cfg(test)]
3  mod test {
4      use super::*;
5
6      #[test]
7      fn load_node_config() {
8      let node: Node = Node::load_node();
9      }
10     #[test]
11     fn encrypt_decrypt_roundtrip() {
12     let node: Node = Node::load_node();
13     let original_message = "Hello".to_string();
14     let their_public: PublicKey = PublicKey::from([0u8; 32]);
15     let shared_secret = SecretKey::from(node.keys.secret_key).
           diffie_hellman(&their_public);
16     let enc_msg = shared_secret.encrypt(original_message.clone());
17     let dec_msg = shared_secret.decrypt(enc_msg);
18     assert_eq!(
19        original_message.clone(),
20        dec_msg.trim_end_matches(char::from(0))
21     );
22     }
23 }
```

As shown above, I have implemented tests into the my Rust application. The two tests above check if the application is able to:

1) Open or create a TOML file which is the configuration for the user's node and load it into the program from storage to memory to be utilised by the application.

2) Perform a roundtrip from an original message, to an encrypted message using the curve25519 and AES encryption scheme then finally decrypt the message. It then checks if the decrypted message is the same as the original message as this is needed for end-to-end lossless encrypted data transfer. The macro "assert_eq" checks whether the original and decrypted message are the same.

Below is the return of the application when you run the tests and both pass:

```
running 2 tests
test test::load_node_config ... ok
test test::encrypt_decrypt_roundtrip ... ok

test result: ok. 2 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out; finished in 0.02
s
```