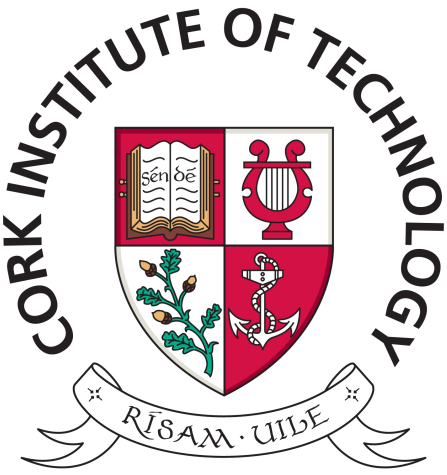

DynamiCrypt

by

Artiom Sumigora

This thesis has been submitted in partial fulfilment for the degree of Bachelor of Science in Software Development

in the
Faculty of Engineering and Science
Department of Computer Science

April 2019

Declaration of Authorship

I, Artiom Sumigora, declare that this thesis titled, 'DynamiCrypt' and the work presented in it are my own. I confirm that:

- This work was done wholly or mainly while in candidature for an undergraduate degree at Cork Institute of Technology.
- Where any part of this thesis has previously been submitted for a degree or any other qualification at Cork Institute of Technology or any other institution, this has been clearly stated.
- Where I have consulted the published work of others, this is always clearly attributed.
- Where I have quoted from the work of others, the source is always given. With the exception of such quotations, this project report is entirely my own work.
- I have acknowledged all main sources of help.
- Where the thesis is based on work done by myself jointly with others, I have made clear exactly what was done by others and what I have contributed myself.

Signed: _____

Date: _____

CORK INSTITUTE OF TECHNOLOGY

Abstract

Faculty of Engineering and Science
Department of Computer Science

Bachelor of Science

by Artiom Sumigora

In today's world information is mostly sent in an encrypted form over the public internet. Traditionally when a client connects to a server the public keys are shared and the same set of public/private key pairs are used for the session and potentially for future sessions, depending on how the system is setup. Provided that industry standard encryption is used it would take the attacker longer than the lifetime of the earth to crack the key using common cracking techniques, making the system secure. The problem arises if the attacker managed to get the key in some other fashion other than brute force it could be possible to decrypt potentially sensitive information that was captured over the network.

This thesis explores an alternative method of generating keys safely over the public network without using public key cryptography schemes such as RSA. This thesis also further increases security by changing encryption keys during the session. In order to achieve this a type of neural network called a tree parity machine will be used.

A tree parity machine consists of input neurons, hidden neurons and one output neuron. A neural network is chosen for this because the weights of neural networks can be synchronised between each tree parity machine stored on different hosts. The weights can then be used to generate a key and since the weights on both tree parity machines are identical the same key will be generated. The weight are synchronised over the network with no information sent about the weights itself therefore the attacker will not be able to figure out the key. The tree parity machines can then be desynchronised on purpose to produce new keys thereby changing keys throughout the session.

This thesis provides a software solution that is capable of handling synchronisation using multiple tree parity machines to exchange encrypted data between multiple hosts. An API will be provided that will allow other any web server to use dynamic encryption. A NodeJs App will also be provided to demonstrate the use of the API in a messaging type environment.

Acknowledgements

This project has taken a substantial amount of work, dedication and research. I would like to say special thanks to:

My friends and family for supporting me.

Dr. John Creagh, project supervisor in semester one. Seamus Lankford, project supervisor in semester two.

Contents

List of Figures

List of Tables

Abbreviations

TPM	**T**ree **P**arity **M**achine
API	**A**pplication **P**rotocol **I**nterface
HTTP	**H**yper **T**ext **T**ransfer **P**rotocol
HTTPS	**H**yper **T**ext **T**ransfer **P**rotocol **S**ecure
SSL	**S**ecure **S**ockets **L**ayer
TLS	**T**ransfer **L**ayer **S**ecurity
AES	**A**dvanced **E**ncryption **S**tandard
RSA	**R**ivest **S**hamir **A**dleman
SSH	**S**ecure **SH**ell
GPG	**G**NU **P**rivacy **G**uard
GNU	**GNU**'s Not Unix
AI	**A**rtificial **I**ntelligence
GAN	**G**enerative **A**dversarial **N**etwork
IP	**I**nternet **P**rotocol
IBM	**I**nternational **B**usiness **M**achines
ID	**ID**entification
PHP	**H**ypertext **P**re**P**rocessor
KiB	**Ki**bi**B**yte
TCP	**T**ransmission **C**ontrol **P**rotocol

Dedicated to my family...

Chapter 1

Introduction

1.1 Motivation

Access to the internet is becoming more and more easier as well as cheaper allowing more devices to communicate between each other. In order for those devices to communicate securely and only communicate with the parties chosen methods of encryption need to be exercised. Constant increase in computing performance has proven to make some encryption methods depreciated due to a high chance of successful decryption with little time involved. This increase can be easily expressed since the computer used to send the first rocket to the moon is less powerful than a cheap smart phone today.

Industry standard encryption methods today rely on the fact that it would take longer than the lifetime of earth to crack an encryption key using today's computing power provided there are no vulnerabilities in the encryption method, and the attacker didn't get lucky and manage to guess the key early on in the attack. The chance of randomly generating the correct key is very very low and therefore when cracking cryptography methods other forms of attacks are used. One of these methods could include having unauthorised access to a host and finding keys on the disk, cache or ram.

Discovery of a key by means of any method will lead to successfully decrypting interactions between hosts of a particular session since each session generally encrypts information with the same key.

Due to this, methods of encryption should acknowledge the possibility of the key being leaked or stolen and therefore minimise or ideally nullify the amount of potential private information available to the attacker with the stolen key. This project will provide a solution to the above by using dynamic encryption to essentially switch keys during a session.

1.2 Contribution

The outcome of this project could benefit companies or individuals who want to transfer information in the most secure way possible. Dynamic encryption as of writing this paper is a niche topic primarily due to the fact that industry standard encryption methods are generally secure enough and quite difficult for attackers to overcome. There is a company called Dencrypt [**?**] that heavily uses dynamic encryption for voip communication so perhaps the use of dynamic encryption will increase and be used by other companies in the future in order to transfer other types of information.

This research paper focuses on improving the computing area of secure transfer of data as well as cryptography with the help of machine learning algorithms. Using machine learning for cryptography is a topic of research undertaken by large companies such as Google, however the researchers mostly focused on creating new encryption algorithms whereas this paper adds additional measures to existing industry standard encryption methods.

1.3 Structure of This Document

Chapter 1 of this document is the introduction to this paper. The motivation describes why I choose to do a project revolving around dynamic encryption and secure transfer of information. The contribution section describes how and why my project will impact companies and security minded individuals. The structure of this document section briefly describes the structure of this document and what each of the chapters and sections entail.

Chapter 2 describes the project background related to computer science. The main area in which the project falls under is explored, the main areas and topics which correspond to this project are identified. Thematic Area within Computer Science describes the technologies and methods that the project uses and builds upon. A Review of thematic depicts the research papers, articles, forums, blogs and anything else that has been used for research.

Chapter 3 explains the problem this project is solving including any objectives and desired achievements. Functional and non-functional requirements are also listed as a conclusion to the chapter.

Chapter 4 specifies how the project will be implemented. Starting with the architecture which includes class diagrams of various parts of the system followed by an explanation of how the system will be potentially built. Any frameworks and critical libraries are

identified and explained how their use will impact the project. The potential risks are introduced and explored that can have a negative or critical impact on the completion of the project. An explanation of the methodology that is used when doing this project is provided. An estimated time schedule is provided including the estimated tasks required for each sprint. How the project should be evaluated is provided and lastly a simplistic working prototype that demonstrates successful synchronisation between two tree parity machine.

Chapter 7 is a reflection of any problems that were encountered during semester one of the project, how they were solved and how that influences the procedure for semester two part of the project. Main conclusions are provided for background, problem description and the solution approach used in this project. And finally anything that should have been done given more time had been available for this report.

Chapter 2

Background

2.1 Thematic Area within Computer Science

The Core topic of this project is safely and dynamically encrypting messages between two parties. The communication will rely on multiple functioning NodeJs servers for transfer of encrypted messages.

The core areas under which my project falls under is cryptography, security for encrypting and securing information. Machine learning will be used for establishing methods of secure information exchange. And finally networking due to the setup required of communicating between different servers and sending encrypted information.

This project will be compatible with NodeJs because it is one of the fastest growing server platform [**?**] that can be easily set up and supports a large amount of modules.

Encryption [**?**] is when the plaintext of any form of data that can be easily read is converted to an unreadable encoded version. In order to retrieve the original data for viewing or processing it must be decoded using a specific algorithm and more than likely some sort of key, usually a lengthy password. Encryption may be used for encrypting files and operating systems on a user's hard drive. In today's world encryption is used religiously for data transferred between networks. Sensitive information like user's credentials are constantly being sent from the browser to the server when logging into websites for personalised content. The same is true for for even more high risk information like banking details, scans of identification documents and even keys. Websites that wish to be secure are now using HTTPS instead of HTTP. The Number of websites using HTTPS is constantly increasing see figure 2.1

HTTP is not secure because information transmitted is in plaintext by default and extra steps are needed to encrypt the data. Because the author of the server can choose how

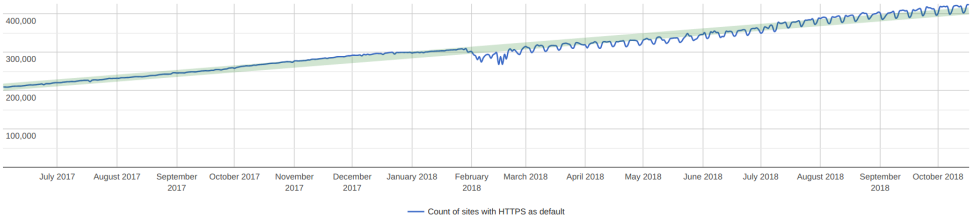

FIGURE 2.1: A graph of HTTPS usage increase[**?**]

the data is encrypted, it can lead to the theft of data as the implementation may not be correct or a weak algorithm is used. On the other hand if a website uses HTTPS which is a common defined standard there will be minimal data theft see figure 2.2. HTTPS uses SSL or TLS which are protocols that use asymmetric keys (will be discussed later). SSL is generally used more often as it requires the server to acquire an SSL certificate from a trusted third party.

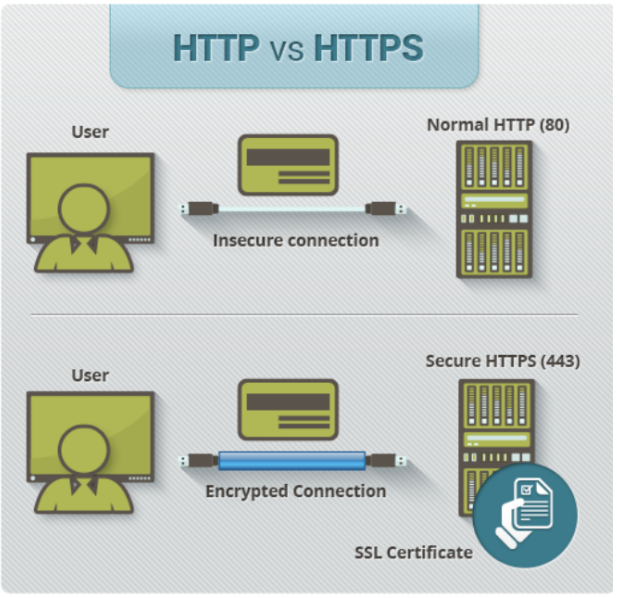

FIGURE 2.2: HTTP vs HTTPS[**?**]

Traditionally there are two encryption types.

1. Symmetric
2. Asymmetric

Symmetric encryption uses the same key to encrypt and decrypt information. This type of encryption is usually used to encrypt information provided by a human generated key. It is not safe to send this key over a network as it can be stolen or the destination being sent to can be spoofed. There are multiple implementations of symmetric

ciphers, the most common being AES, Twofish and Serpent. To increase your security at the cost of encryption and decryption time you can chain multiple ciphers together AES(Twofish(Key)).

Asymmetric or commonly known as public key encryption methods are commonly used for sharing data between between computers on different networks. This is because a set of keys are generated one being private and the other public. The private key is never shared and remains on the host that generated it. The public key on the other hand can be shared with the party you want to communicate securely with. The public key is used to encrypt the data that is about to be sent back. This data can only be decrypted using the private key. Therefore you can share your public key with anyone and they wont be able to decrypt messages sent from another host who used the same public key. The most common algorithm is RSA. Certain protocols also use public key algorithms like SSH for secure remote connections to foreign hosts. And GPG for verification of packages on Linux systems and an alternative over https for Github.

Protocols like SSH create a set of keys during the start of the session and those keys remain constant therefore if the private key was leaked the whole conversation could be decrypted if the packets have been captured and stored.

Encryption in its general form is simply a mathematical algorithm that takes plaintext and combines it with some sort of key over a number of iterations eventually producing the ciphertext. This might entice some people to try and break those ciphers and recover the original plain text. Quite a number of attacks do exist. Private keys are sometimes stored on the disk or in temporary files that are saved by programs during their execution, until reboot or they are cleared after a number of days. The attacker may be able to access the server physically or remotely using an unrelated exploit and copy the key.

Social engineering is an attack where a human pretends to be of an authority figure and convinces an unaware human to give up the key. This can be done by an attacker pretending to be an executive engineer in a company and convince the victim indirectly or directly to give up the keys by running obfuscated commands in the terminal which then send over the key to the attackers server.

If the key used is created by a human and not some sort of machine generator there are a few number attacks that can be performed that would not be feasible or possible if the key was generated or quite long. These attacks include brute force which creates keys in usually ascending order or based on some algorithm to increase the chance of success. Brute force will eventually try every key possible however even a small sized key of 12 characters containing numbers, symbols, upper and lover case numbers it would

take around 200 years [**?**]. Dictionary attacks can be used if the key is part of a large dictionary of human created passwords.

Attacks on proper keys that are generated by machines are more sophisticated and rely on cracking the algorithm or device used for encryption more so then the key. Linear cryptanalysis [**?**] is a plaintext attack which means that the attack can use any plain text they want and receive the ciphertext for it after putting it through a system. This attack uses linear approximations to describe the behaviour of the block cipher. After large number of pairs of plaintext and cipher text there is a possibility to learn something about the key.

Algebraic attacks [**?**] can be used if the ciphers exhibit a high probability of a mathematical structure.

Reverse Engineering [**?**] can be used to either examine the source code of the algorithm or disassemble the binary which uses the algorithm to look as the assembly code of the algorithm. Machine key generators usually use some form of a random number generator which are algorithms that usually take in a seed hopefully something that isn't the current time but that has been known to be used and return a key. Attacks can be made on this number generator if the seed is something predictable or the generator generates predictable numbers.

If the device on which the algorithm is performing on is an embedded device you can perform side channel attacks [**?**] where you measure the spikes and frequency of the power consumption when the encryption is taking place.

Machine learning is a category of algorithms that allow systems to automatically learn and improve from experience without being explicitly programmed [**?**]. Basically this means that the algorithm can update when input is received this in turn updates the the output even if the same inputs are used later on. Typically machine learning software processes large amounts of data and looks for patterns constantly updating either variables or adding logic branches. Recommendation engines use machine learning to personalise the logged in users feed. So if user one looked at product X and user 2 bought product X and also bought product Y then user one will most likely see product Y as a recommendation. There are three types of recommendation systems. Collaborative Filtering [**?**] where similarities between customers is taken into account. Content Based Filtering [**?**] is when the liked and purchased items are taken into account. See figure 2.3 for illustration. Finally there is Hybrid Recommendation Systems which is a mix between the previous two and is typically the one used in industry.

The most common machine learning algorithms are often classified as supervised learning and unsupervised learning. Supervised learning [**?**] relies on a set of training data which

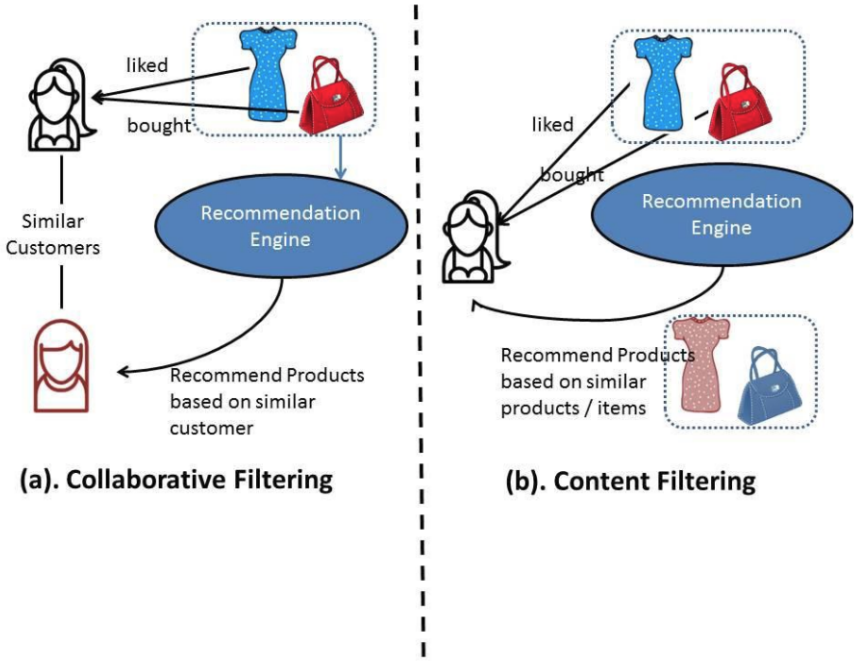

FIGURE 2.3: An example of recommendation engine[**?**]

is constructed of various inputs and and correct outputs corresponding to those inputs usually labels. The training data is usually left unchanged and when the test data or data taken from the current user is used as input the algorithm attempts to correctly classify the data based on the training data. The disadvantage of supervised learning is if the incoming data is radically different from the test data the probability of classifying the data into the correct label will be similar and sometimes even lower than classifying into the incorrect label. However because the training data exists it can be quite quicker to setup a supervised learning system then an unsupervised.

Unsupervised learning [**?**] uses data that is neither classified or labelled. This allows the algorithm to draw its own conclusions about the data as well as discover hidden structures. If some of the data is similar in any way to another piece of data the algorithm tends to group those pieces of data together. Unsupervised learning is generally more complex and can be more accurate since the algorithm might find subtle discrepancies that might be impossible to notice for a human. This however can lead to unpredictable behaviour where there is expected to be two classifications but instead the data is classified into a lot more than two classifications. Unsupervised learning algorithms typically require more training data than supervised in order to detect similarities and come up with labels.

There are also algorithms that use techniques found in both supervised and unsupervised machine learning these are called Semi-supervised [**?**] machine learning these use a

much larger amount of unlabelled training data than labelled. These types of algorithms usually tend to be more accurate than supervised and unsupervised alone.

Another interesting suite of machine learning algorithms is classified into Reinforcement Learning [**?**]. These algorithms interact with the environment and received rewards or penalties based on the actions carried out within the environment. If the action carried out receives a reward it is likely that the same action will be repeated again.

Achieving synchronisation [**?**] is an important part of this project. Synchronisation is the process of making two or more data storage devices or programs (in the same or different computers) having exactly the same information at a given time. Synchronisation is common in multi-threaded software where any number of threads can manipulate the same set of data or even wait for a certain thread to finish execution. For example you wouldn't want your parent thread to finish before the child threads as this can lead to data loss and resources being hogged by a zombie thread. Pthread [**?**] is a great API that can be used for handling threads and shared data between those threads. Data Synchronisation [**?**] is an on going process of having the same data present on selected servers. Large databases that operate on multiple servers usually use data versioning to keep check on the latest data. MongoDB is one such database [**?**] this is the reason for a delay when you upload a YouTube video (YouTube doesn't use MongoDB this is just an example) it might not be instantly available for other users to watch as it needs to propagate through a number of servers hosting the databases. Popular websites use data synchronisation for mirroring websites. This allows users to be distributed among relatively identical servers in order to avoid bandwidth bottlenecks as well as increasing availability just in case one server dies users will be redirected to a different mirror seamlessly without interruptions. File synchronisation is the method of choice for home backups this is preferable over the traditional backup methods where data is simply dumped onto another hard drive. This process prevents copying identical files which leads to faster transfers and a less chance of errors occurring. My personal favourite file synchronisation tool is Unison [**?**]. It is also possible to synchronise folders over the network between two home computes, Unison does this through SSH. Blockchain [**?**] the method used for secure crypto ledgers which also relies on synchronisation between peers. Synchronisation for the ethereum crypto currency blockchain can be expressed as follows. Synchronisation proceeds from head to known block by requesting and fetching block hashes iteratively from head to root. Based on block hashes, blocks can be requested. Based on the parenthash of a block, independent sections can be linked and a chain established. By checking if a block is found in the block chain the root of the blockpool can be established and the chain can be inserted in the blockchain [**?**].

2.2 Review of the thematic area

To achieve my goal of dynamically encrypting messages between both parties, each party must know the encryption key without explicitly sending that key to each other. For this to have a dynamic effect multiple processes will in this case synchronise which each other and a separate process will take care of swapping keys.

The research papers that will be cited synchronise once therefore I will just have to take what I learned and apply it to multiple instances.

Two identical neural networks that originally have a random generated state. This state is different between the two networks and to achieve synchronisation this state must be the same, this is because when the state is the same it will be essentially the key used to encrypt messages. And all this will be achieved without ever sending the key over the network even in an encrypted form as in public private encryption techniques.

A tree parity machine is a common method used across all papers in order to achieve this goal figure 2.4 visualises such a machine.

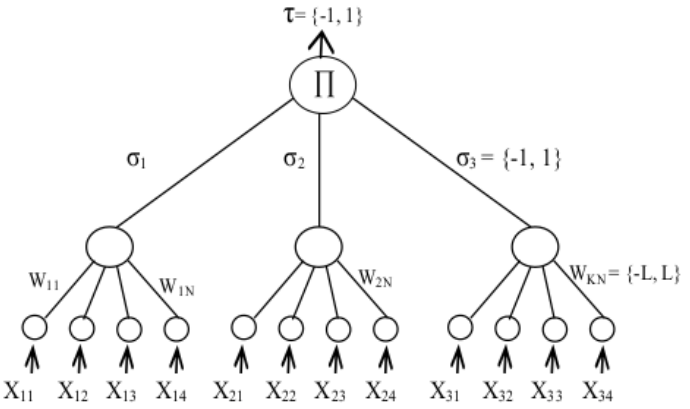

FIGURE 2.4: Tree parity machine with L=[-4,4], K=3 and N=4[**?**]

According to [**?**] both parties should have the same layout of the tree parity machine where N is the number of neutrons used as input for each hidden neutron. The input neutrons have the X and are at the bottom of the diagram, in this case there are four input neutrons for each hidden neutron. The hidden neutrons are referenced as K, these are the neutrons in the middle with the W in the diagram. These are the weights which are updated if the output of both of the tree parity machines are the same. These weights in the end will be the key used for symmetric encryption between the two parties. T will be the output value which will be compared to the other tree parity machine.

There is a low number of machine learning algorithms that can be used to synchronise and the paper [**?**] as well as the majority of papers use the hebbian-learning rule figure 2.5

$$w_i^+ = w_i + \sigma_i x_i \theta(\sigma_i \tau)\theta(\tau^A \tau^B)$$

FIGURE 2.5: Hebbian learning rule[**?**]

The other two learning rules that can easily be substituted according to this paper [**?**] are as follows anti-hebbian learning rule figure 2.6 and random-walk rule figure 2.7.

$$w_{i,j}^+ = g(w_{i,j} - x_{i,j}\,\tau\theta(\sigma_i \tau)\theta(\tau^A \tau^B))$$

FIGURE 2.6: Anti-Hebbian learning rule formula[**?**]

$$w_{i,j}^+ = g(w_{i,j} + x_{i,j}\,\theta(\sigma_i \tau)\theta(\tau^A \tau^B))$$

FIGURE 2.7: Random-Walk learning rule formula[**?**]

There are a number of steps involved in synchronising the tree parity machines. Step 1. The weights at the beginning should be randomly initialised using local randomisation techniques because there is a chance that downloading data from random APIs can be spoofed which will result in the attacker easily synchronising with your tree parity machine.

Step 2. Generate random input which will be used by both of the tree parity machines. This input can be generated by a third party server or one of the two parties.

Step 3. Calculate the value of the weights based on the random input using the formula in figure 2.8

$$\sigma_i = \operatorname{sgn}\left(\sum_{j=1}^{N} w_{ij} x_{ij}\right) \quad \operatorname{sgn}(x) = \begin{cases} -1 & if \quad x < 0, \\ 0 & if \quad x = 0, \\ 1 & if \quad x > 0. \end{cases}$$

FIGURE 2.8: Hidden neutron formula[**?**]

Step 4. Calculate the output neutron based on the weights using the formula in figure 2.9

$$\tau = \prod_{i=1}^{K} \operatorname{sgn}\left(\sum_{j=1}^{N} w_{ij} x_{ij} \right)$$

FIGURE 2.9: Output Neutron Formula[**?**]

Step 5. exchange the output between both of the tree parity machines through a network and if the outputs are not the same repeat again from step 2. And if the outputs are the same apply the hebbian learning rule to the weights and update them accordingly. Repeat step 3 to step 5 until both of the tree parity machines have the same weights.

This technique can be considered too simplistic and the attacker has a greater probability in synchronising with the two parties. I will cover attacks related to tree parity machines further in the section.

To increase the speed of synchronisation and lower the chance of the attacker being able to synchronise with the parties. A technique called queries is used [**?**]. This technique replaces step 2 from before where the input would be randomly generated by a third party or one of the party. A query consists of a generated vector based on a field of the weights. These queries are then sent from each party to the other interchangeably. To calculate the new local field value the following formula can be used figure 2.12

$$h_k = \frac{1}{\sqrt{N}} \sum_{l=1}^{L} (c_{k,l} - c_{k,-l})$$

FIGURE 2.10: Local Field Formula[**?**]

It is possible to use a different algorithm where the output

$$\sigma_k$$

is chosen random for the hidden unit. It is possible to use the following formula to calculate the local field value.

$$h_k = \sigma_k H$$

To calculate the list of c values that will be used to affect the generation of input values it is possible to use the following two formulas.

$$c_{k,l} = \left\lfloor \frac{n_{k,l}+1}{2} + \frac{1}{2l}\left(\sigma_k H\sqrt{N} - \sum_{j=l+1}^{L} j(2c_{k,j} - n_{k,j}) \right) \right\rfloor$$

FIGURE 2.11: C Formula one[?]

$$c_{k,l} = \left\lceil \frac{n_{k,l}-1}{2} + \frac{1}{2l}\left(\sigma_k H\sqrt{N} - \sum_{j=l+1}^{L} j(2c_{k,j} - n_{k,j}) \right) \right\rceil$$

FIGURE 2.12: C Formula two[?]

Where one formula is chosen randomly in each calculation. The probability of one formula being chosen is 50 percent. Th inputs are then generated and if the inputs are associated with zero weights then the inputs are randomly generated since they do not influence the local field value. The input is then divided into L groups and are selected randomly and assigned to

$$x_{k,j} = sgn(W_{kj})$$

Finally the remaining input values are set to

$$x_{k,j} = -sgn(W_{k,j})$$

To achieve secure key exchange with queries the parties should choose the parameter H so that they can synchronise quickly while the attacker would not be able to do so in time. Despite the queries being based on the weights an attacker cannot predict the query generated by either party since the weights are never shared. Therefore an attacker can collect the queries but won't be able to establish a mathematical connection between them. After the synchronisation is complete the weights can be used as a seed for a random generator. As the attacker doesn't know this seed the output of this generator won't be predicted.

There are a number of different attacks that can be carried out on the tree parity machines. These attacks are more successful if using the basic synchronisation method without the use of queries. This is because up to this date there is no documented or rumoured methods of being able to synchronise with parties who are using queries.

The most basic and useless attack would be a brute force. This involves trying every possible values for the weights. If using a tree parity machine consisting of 3 hidden neurons, 300 input neurons and 3 weights will result [?] in

$$3 * 10^{253}$$

possible values for the weights making this attack quite impossible using modern computing power.

The attacker can attempt to learn the weights by using their own tree parity machine with the same number of hidden neutrons and inputs [**?**]. This is essentially identical to the parties tree parity machines but with different initial weights and the attacker synchronises indirectly. There are three possible situations that can occur with this attack. In these situations A and B will be the two parties trying to synchronise and E is the attacker. situation 1. Output of A doesn't match output of B and therefore none of the parties including the attacker update their weights. situation 2. Output of A matches output of B and output of E is also the same. This time A, B and E update their weights. situation 3. Output of A matches output of B but output of E doesn't match. parties A and B update their weights but E cannot do that and therefore it will take E more time to synchronise to A than it would for B to synchronise with A. Because the learning is ceased after A and B are synchronised E will be left in the dark and not synchronised state. You can further decrease the success rate of this attack by increasing the synaptic depth of the neural network [**?**]. Increasing this will have a performance impact on the parties polynomially however the chance of successful attack decreases exponentially.

More sophisticated attacks can be found in this research paper [**?**] such as the genetic attack. This uses a form of genetic algorithm the key to a successful attack using this method revolves around E being able to determine the fitness of her neural networks. The attacker spawns a number of tree parity machines which attempt to synchronise with A. After a set time period a selection is made where the most successful synchronised tree parity machines are used to generate the next population. This selection works best if there are certain observable differences between attractive and repulsive effects. This attack can be quite successful if the synaptic depth of the neural network is not too large. This can be expressed with the following formula in figure 2.13 where the probability of E being successful decreases exponentially with the synaptic depth L.

$$P_E \sim e^{-y(L-L_0)}$$

FIGURE 2.13: probability of E being successful[**?**]

In order for this attack to be likely successful the number of tree parity machines the attacker will use will need to be exponentially increased with the increase of the synaptic depth see figure 2.14.

$$M \propto e^{L/L_E}$$

FIGURE 2.14: Tree Parity Machines needed for synaptic depth[**?**]

The most successful attack for the simple tree parity machine is the majority attack [**?**]. The genetic attack works better than the majority attack if the synaptic depth is not large. The majority attack uses multiple tree parity machines just like the genetic attack however there is no selection of the fittest tree parity machines. This way the number of tree parity machines is constant. This attack uses the Bayes learning rule [**?**] to find the overlap between the hidden units of one of the parties network and the attackers network. It is possible to use the following formula in figure 2.15 to calculate the probability of synchronisation. At the beginning of the attack the probability of

$$\rho_i^{AE} = \frac{1}{M} \sum_{m=1}^{M} \frac{\mathbf{w}_i^A \cdot \mathbf{w}_i^{E,m}}{||\mathbf{w}_i^A||\,||\mathbf{w}_i^{E,m}||}$$

FIGURE 2.15: probability of E being successful[**?**]

the attacker being successful is 0 and when the attack is successful the probability is 1. The average overlap between two of the attackers networks can be calculated using the following formula figure 2.16. If the result of this formula is 1 then the two of

$$\rho_i^{EE} = \frac{1}{M(M-1)} \sum_{m=1}^{M} \sum_{n \neq m} \frac{\mathbf{w}_i^{E,m} \cdot \mathbf{w}_i^{E,n}}{||\mathbf{w}_i^{E,m}||\,||\mathbf{w}_i^{E,n}||}$$

FIGURE 2.16: probability of overlap[**?**]

the attackers networks tested are identical which can reduce the majority attack to the geometric method. If trying to attack a tree parity machine with 3 hidden neutrons, 1000 input neutrons and a synaptic depth of 5 it is possible to successfully synchronise at around 1000 steps. this is done with only 100 attacking networks. Figure 2.16 demonstrates the this attack over 1000 steps.

My project falls mostly under cryptography and secure transfer of information therefore the companies that could be potentially interested would need to be involved in that market. Google is known for the many cloud services it provides like gmail, docs and drive. The data needs to be encrypted before sending it to the user to display. When using the products Google claims to use several layers of encryption to protect customer data [**?**]. Google has its own cryptographic library called Tink [**?**], ensuring that

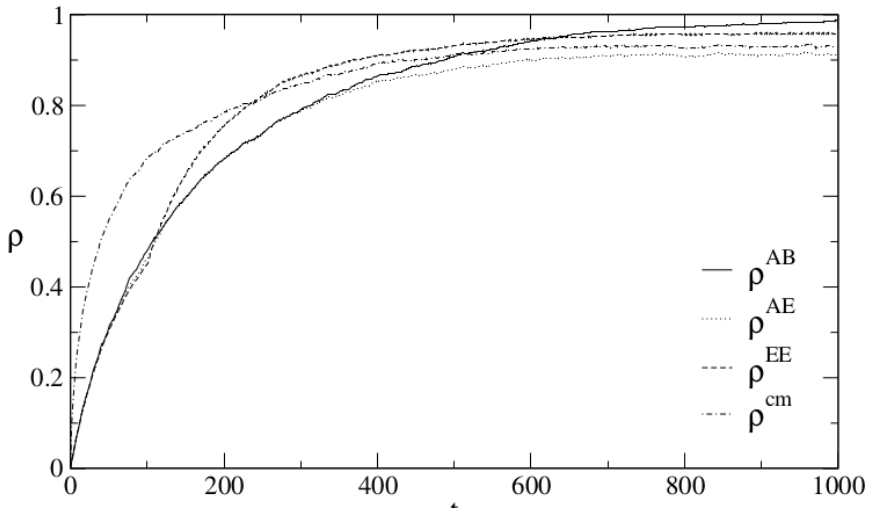

FIGURE 2.17: Majority attack results[**?**]

they have an interest for cryptography development. Google also takes part in less conventional methods of cryptography like using their own neural network to come up with obscure encryption methods [**?**]. Google's AI successfully created secure algorithms [**?**] that use inhuman cryptographic schemes making them harder to crack. This technique is called GAN Cryptography [**?**] for which a research paper can be found. The encryption methods my project uses integrate a machine learning algorithm which could peak the interest of Google.

Dencrypt [**?**] is a company that uses dynamic encryption for voice over IP communication. They use a wrapper around AES-256 to achieve the dynamic nature. They claim dynamic encryption is the state of the art in cryptology and I believe they would be interested in different methods of achieving dynamic encryption. Dencrypt explains that even if someone manages to decrypt a single data transaction the next transaction would not reveal information as it is encrypted in a different way, This is exactly what I'm aiming to achieve with my project just using different methods.

Baffle [**?**] offers advanced data protection solutions for financial services, healthcare, secure cloud migrations and saas providers. Their product also protects data while it is in memory and while it is processing this means they are very serious about data protection and try to close gaps in the data threat model. In order to achieve this they use a number of different methods to secure data as seen in figure 2.18. The project on which I'm working on could be useful to Baffle when transferring data between servers. Baffle attempts to avoid legacy encryption solutions and favours more advanced versions which my project may peak their interest.

IBM offers many cloud products that require encryption for security purposes. IBM Guardium [**?**] is a product that is purposely built for encrypting files and databases.

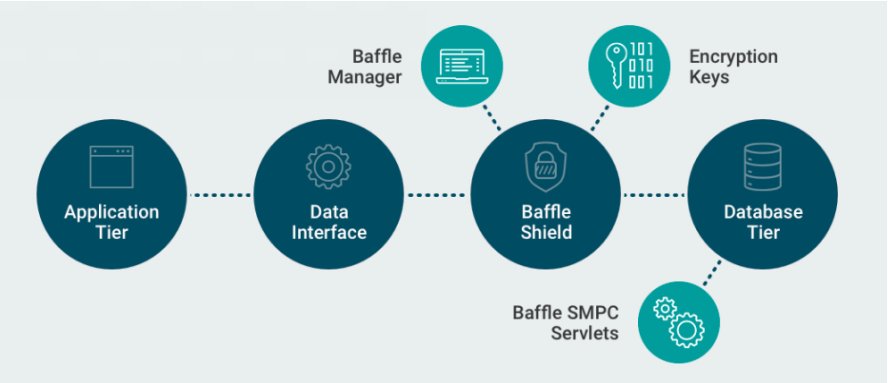

FIGURE 2.18: How does Baffle work[**?**]

This features centralised key and policy management as well as granular encryption of files and folders each protected with its own encryption key. The same is true for volumes of data. There is a clear indication that good security results in the use of multiple keys. I believe IBM could be interested in some sort of dynamic encryption perhaps not for IBM Guardium as that is mainly for safe storage and my project revolves around secure transfer of data between hosts, but in one or more of their many other products.

Dell has a product called Encryption Enterprise [**?**] which handles everything like file encryption, malware prevention and protecting data in motion. Dell also takes pride in the fact that this product integrates well with other solutions like Bitlocker. The Encryption Enterprise software is also used by the government, healthcare and other critical infrastructure ensuring its high reputation status. The data in motion would relate to my project which could be of interest to Dell as data sent between hosts needs to be encrypted for maximum security.

Chapter 3

DynamiCrypt

3.1 Problem Definition

The problem this project is designed to solve is increase the level of security when transferring data between hosts. Traditionally when transferring data between two hosts an encryption key for that session is generated and normally used until the session ends. In some cases the same key is used for future communications. This can cause issues and potential data theft if the attacker manages to capture the session and get access to the key by any means necessary, it would be possible to decrypt the session and potentially reveal important/private information.

The information revealed can potentially cause data breaches. In 2018 alone major data breaches have occurred leaking millions of private user data including Facebook where 87 million users were compromised [**?**]. The largest data breach this year affected 1.1 billion registered Indian citizens. The company that was compromised is Aadhaar [**?**] causing personal addresses, phone numbers, emails and names to be available to anyone willing to part with a few hundred rupees.

This project will mitigate this potential problem by periodically changing the encryption key throughout the duration of the session with both parties being aware of the current key in use while the attacker will not know the key. By using this method if the attacker manages to get the encryption key only a small portion of the session could be decrypted. The random keys generated for encryption will be isolated from each other therefore if the attacker has one key it will not be possible to perform mathematical operations on the key to calculate the next keys used for encryption.

3.2 Objectives

The objectives of this project focus on providing a sort of an API that allows to dynamically synchronises with other remote hosts and shares encryption keys with other servers running on the same machine, in order to provide dynamically encrypted information.

Objectives are identified in a way that makes then achievable under the time constraint.

1. Provide an API that can be configured only by servers running only on the same machine.
2. Provide a number of tree parity machines that will synchronise with remote tree parity machines these will be configured through the API and will be able to access machines outside of localhost.
3. Provide methods of identification for threads, hosts and threads for the different hosts.
4. Provide methods for allowing multiple hosts to access the API.
5. Evaluate performance and maybe use and test different learning algorithms.
6. Create an easy to use NodeJs module for easy implementation with NodeJs Applications.
7. Provide an optional feature where extremely sensitive information like passwords should be split into pieces and sent over the network with each piece encrypted with a different key.

3.3 Functional Requirements

1. Share connection information between hosts. Port number(s), number of synchronising threads, thread id(s) and maybe more.
2. Threads should be easily identified as there will be multiple threads synchronising simultaneously.
3. Each thread synchronises with remote thread.
4. API sends one key to another server running on localhost and only servers running on localhost after synchronisation between one or more threads occurred.
5. After key is extracted from thread the localhost thread and corresponding remote thread will desynchronise and begin to synchronise again.

6. API will be notified when a host disconnects and connects to clean up accordingly.
7. sensitive information in ram should be protected/encrypted.
8. sensitive information will not be saved in logs or cache.

3.4 Non-Functional Requirements

1. The first thread to synchronise should take no longer than 2 seconds.
2. The program should work on any Linux host with a kernel version of at least 3.10.
3. The NodeJs module should support NodeJs version 8 and above.
4. The NodeJs module should provide easy to use callbacks and hide the complexity.
5. The install script should work on all supported distros provided dependencies are met.
6. The API should require authentication.
7. The synchronising threads should have some sort of digital signature or a method of identification.
8. The API should support the ability to synchronise with multiple hosts at the same time or multiple instances of the API can be executed or some form of master deals with multiple instances. Perhaps a different port can be used for each connecting host.
9. Program should run under its own user to minimise tampering.

Chapter 4

Implementation Approach

4.1 Architecture

Designing a software architecture is a crucial step of creating software. This will attempt to illuminate any uncertainties that come up with the functionality or implementation. Planning out the architecture before coding often results in a more stable and scalable software. This section will be dedicated to explaining how the project is going to work. The technologies and languages used for this project will be introduced as needed. High level diagrams and class diagrams will accompany the explanation, these diagrams will most likely change during the implementation phase as it would be rather amazing if I got every thing correct here.

The project will be written in C++ as it is a binary compiled language resulting in fast execution of code, this is required as synchronisation takes up a reasonable amount of execution time. C++ has libraries for dealing with networking and threads therefore building an API will be no problem. I also plan on doing a NodeJs module for easy integration with the API using any NodeJS application, this will be done in JavaScript.

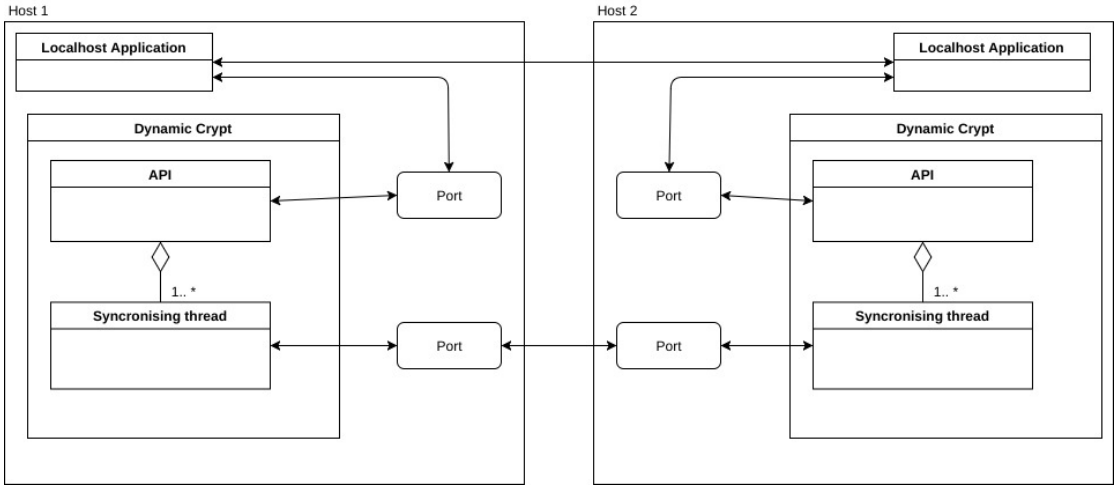

FIGURE 4.1: Two hosts using Dynamic Crypt System

The diagram 4.1 shows a very high level overview of the system in action. There are two hosts in this example. Each host has a localhost Application which is any application that uses the API, it can be a python web application, NodeJS web application, PHP web application etc...

Each host has a Dynamic Crypt software running which is what this project is about.

The Dynamic Crypt has two main sort of components if you like. The API which is responsible for communication between applications running on localhost only for security purposes. The localhost applications communicate with the API through a port that will be configured with iptables to only be available for local addressees only so 127.0.0.1 only.

The Synchronising thread "component" is a single Tree Parity Machine as well as the necessary functionality required for networking and synchronisation between the partner thread of the other host. Each synchronising thread will have its own unique id as well as the partner thread id, IP, port etc.. from the other host in order to be able to send packets directed at the partner thread. This is needed because there will be multiple instances or threads for each host connected to a host. In the host 1 to host 2 example there will most likely be only ten threads synchronising with each other this is to ensure a steady supply of encryption keys, this number of threads for each host may change during the implementation phase.

How all of this works is going to be similar to the following.

Host 1: Localhost application establishes a connection with Host 2: Localhost application using standard methods.

When local Host 1: Localhost application wants to use dynamic encryption to send data to Host 2: Localhost application it contacts the Host 1: Dynamic Crypt API with an init request.

Host 1: Dynamic Crypt API takes note of Host 1: Localhost application details such as the applications name or port number to distinguish the application from other local host applications as will be demonstrated in a later example. and initialises ten Synchronising threads. The API associates this list of synchronising threads with the application and sends back the details of each thread, the port the threads will use and the IP of Host 1 to Host 1: Localhost application.

Host 1: Localhost application then essentially forwards this information to Host 2: Localhost application which will know that this is an dynamic synchronisation init request and will forward that information to Host 2: Dynamic Crypt API through the Host 2: Dynamic Crypt API port. The Host 2: Dynamic Crypt API will initialise ten Synchronising threads and assigns a partner to them which is essentially the partner id of one of the Host 1: Dynamic Crypt threads.

Host 2: Dynamic Crypt threads will send a request to Host 1: Dynamic Crypt threads. The Thread manager will then forward the info to each thread assigning the partners thread id for Host 1 threads. Now synchronisation between threads will occur how synchronisation occurs is discussed in the research phase but I will provide a diagram shortly.

When a thread is synchronised the API of both hosts will notify the corresponding Localhost application that dynamic synchronisation may occur. The key of the synchronised thread is saved in the API and the thread is forced to desynchronise and begin synchronisation again in order to get a new key. These keys will be added in a queue on both of the hosts API.

The Localhost application can now call the API with an encrypt message request and pass a message to be encrypted. The API uses a key to encrypt a message then the key is discarded from the API, the encrypted message is returned to the Localhost application for ready for sending.

The message arrives at the other Localhost application. The Localhost application will send a decrypt request to the API with the encrypted message. The API will use the appropriate key to decrypt the message and send it back to the Localhost application.

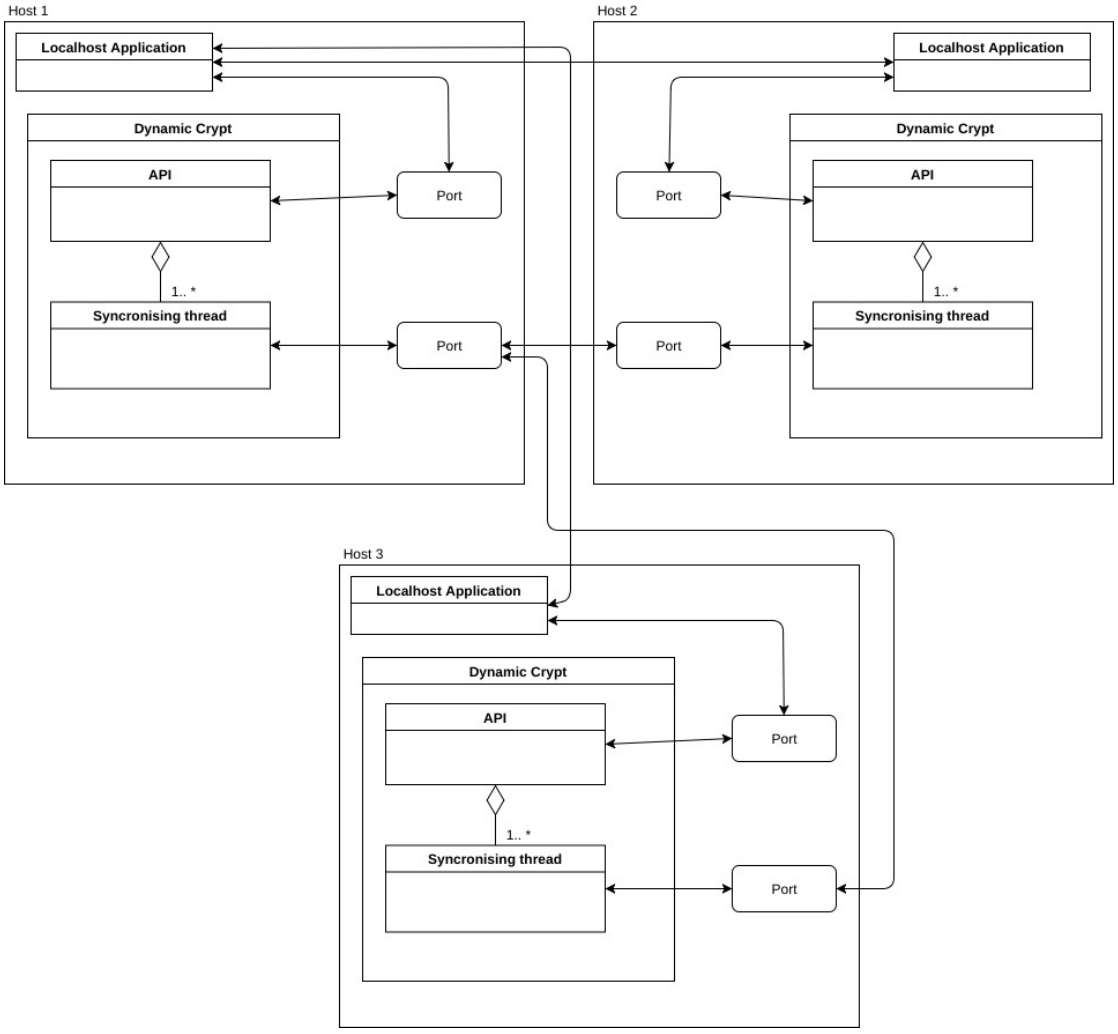

FIGURE 4.2: Host one using Dynamic Crypt System with host two and three at the same time

The diagram 4.2 demonstrates how dynamic encryption can be implemented when Host 1: Localhost application wishes to send dynamically encrypted information between Host 2 and Host 3 simultaneously. Because Host 1 is attempting to use dynamic encryption to communicate between Host 2 and Host 3 only this means that Host 2 and Host 3 are not aware of each other as there is no need for them to communicate.

This works in quite a similar fashion as with only two hosts but now there are three.

Host 1: Localhost application establishes a connection with Host 2: Localhost application and Host 1: Localhost application establishes a connection with Host 3: Localhost application using standard methods more than likely this will happen one after the other as extra data might be required from Host 3, however there is not going to cause any problems doing it at the same time.

Host 1: Localhost contacts the Host 1: Dynamic Crypt API with an init request twice one for Host 2 and Host 3. In the init request Host 1: Localhost application also provides any name for Host 2 and Host 3 to distinguish quickly between the two later on.

Host 1: Dynamic Crypt API takes note of Host 1: Localhost application details such as the applications name or port number and the name given to Host 2 and Host 3, and initialises ten Synchronising threads for Host 2 and another ten Synchronising threads for host 3. The API associates this list of synchronising threads with the application and sends back the details of each thread, the port the threads will use, the IP of Host 1 and lastly the name given to Host 2 and Host 3 to Host 1: Localhost application. These details are sent back for Host 2 and Host 3 separately to allow for easy growth since Host 1 might want to dynamically encrypt information and send it to Host 3 much later than Host 2.

Host 1: Localhost application then essentially forwards this information to Host 2: Localhost application and Host 3: Localhost application these will know that this is an dynamic synchronisation init request and will forward that information to Host 2: Dynamic Crypt API through the Host 2: Dynamic Crypt API port and Host 3: Dynamic Crypt API through the Host 3: Dynamic Crypt API port. The Host 2: Dynamic Crypt API and Host 3: Dynamic Crypt API will each initialise ten Synchronising threads and assign a partner to them which is essentially the partner id of one of the Host 1: Dynamic Crypt threads.

Host 2: Dynamic Crypt threads will send a request to Host 1: Dynamic Crypt threads followed by another request from Host 3: Dynamic Crypt threads to Host 1: Dynamic Crypt threads. The Thread manager will then forward the info to each thread assigning the partners thread id for Host 1 threads. Now synchronisation between threads will occur.

The last three steps are identical as in the last example.

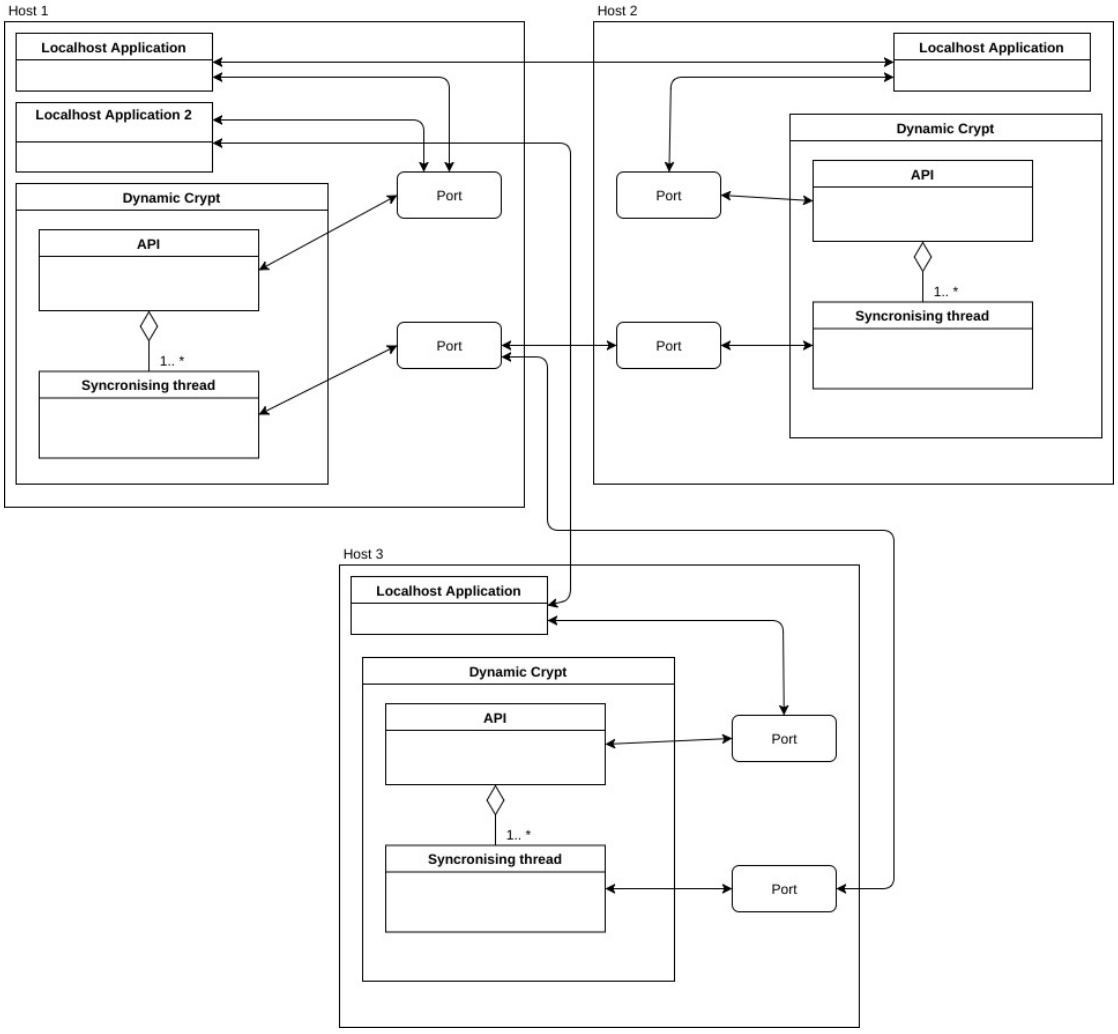

FIGURE 4.3: Host one using Dynamic Crypt System with host two and three with different applications using the API

The diagram 4.3 is quite similar to the last example where as in this case there are two local host applications running on Host 1: In this case localhost application 1 is using the dynamic encryption system to send data to Host 2 and localhost application 2 is using the dynamic encryption system to send data to Host 3. They both use the same API and therefore this example is quite similar to the previous one.

Host 1: Localhost application 1 establishes a connection with Host 2: Localhost application using standard methods. Similarly Host 1: Localhost application 2 establishes a connection with Host 3: Localhost application using standard methods.

Host 1: Localhost application 1 contacts the Host 1: Dynamic Crypt API with an init request. Host 1: Localhost application 2 contacts the Host 1: Dynamic Crypt API with an init request.

Host 1: Dynamic Crypt API takes note of Host 1: Localhost application 1 details such as the applications name or something to distinguish it from Host 1: Localhost application 2, and initialises ten Synchronising threads. The API associates this list of synchronising threads with the application and sends back the details of each thread, the port the threads will use and the IP of Host 1 to Host 1: Localhost application 1 and similar but different values are sent to Host 1: Localhost application 2 as well.

Host 1: Localhost application 1 forwards this information to Host 2: Localhost application which will know that this is an dynamic synchronisation init request and will forward that information to Host 2: Dynamic Crypt API through the Host 2: Dynamic Crypt API port. Host 1: Localhost application 2 forwards this information to Host 3: Localhost application which will know that this is an dynamic synchronisation init request and will forward that information to Host 3: Dynamic Crypt API through the Host 2: Dynamic Crypt API port. The Host 2: Dynamic Crypt API and The Host 3: Dynamic Crypt API will initialise ten Synchronising threads and assigns a partner to them which is essentially the partner id of one of the Host 1: Dynamic Crypt threads.

Host 2: Dynamic Crypt threads will send a request to Host 1: Dynamic Crypt threads followed by another request from Host 3: Dynamic Crypt threads to Host 1: Dynamic Crypt threads. The Thread manager will then forward the info to each thread assigning the partners thread id for Host 1 threads. Now synchronisation between threads will occur.

The last three steps are identical as in the first example.

From the three use case examples the API must be able to support:

- A single localhost application connecting with another dynamic encryption system.
- A single localhost application connecting with multiple other dynamic encryption systems.
- Multiple localhost applications connecting with other dynamic encryption systems.
- Multiple localhost applications connecting with multiple other dynamic encryption systems per each local host application.

Language : Framework	Max time amongst 98% of requests ms, smaller is better	Average requests per second #/sec, larger is better	Lines of code in sample #, count without blanks
C++ : cpprestsdk / default JSON implementation	51	30.70	48
C++ : cpprestsdk / RapidJSON	44	47.06	47
C++ : restbed	7	224.18	39
C++ : pistache	6	319.99	40
PHP : Native implementation	10	146.95	14

FIGURE 4.4: Comparison of different C++ rest frameworks[**?**]

In order to build a proper API I will use a framework as making one from scratch would be too time consuming. There are not many C++ only rest APIs frameworks so the choice was not too difficult. Table 4.4 shows the most popular C++ rest frameworks. Cpprestsdk is made by Microsoft and is generally the most popular one to use. However I have tried installing the library and had issues compiling the examples provided on their GitHub a simple hello world server worked perfectly but the rest API example had issues. This is perhaps for the better that I was unable to get it to work and had to find other frameworks to work with. I stumbled upon the websites that the table is from and the framework Pistache [**?**] seems to be leaps ahead of Cpprestsdk having a low request time of only 6 milliseconds and a large 320 requests per second capability which is what I need for this project and considering that it will take quite a lot of requests to synchronise tree parity machines. Fortunately I was able to install the library correctly and the example rest API code for Pistache compiled and worked correctly.

Therefore I will be using Pistache as the rest API part of this project and the tree parity part of the code also since I wanted my software to utilise two ports.

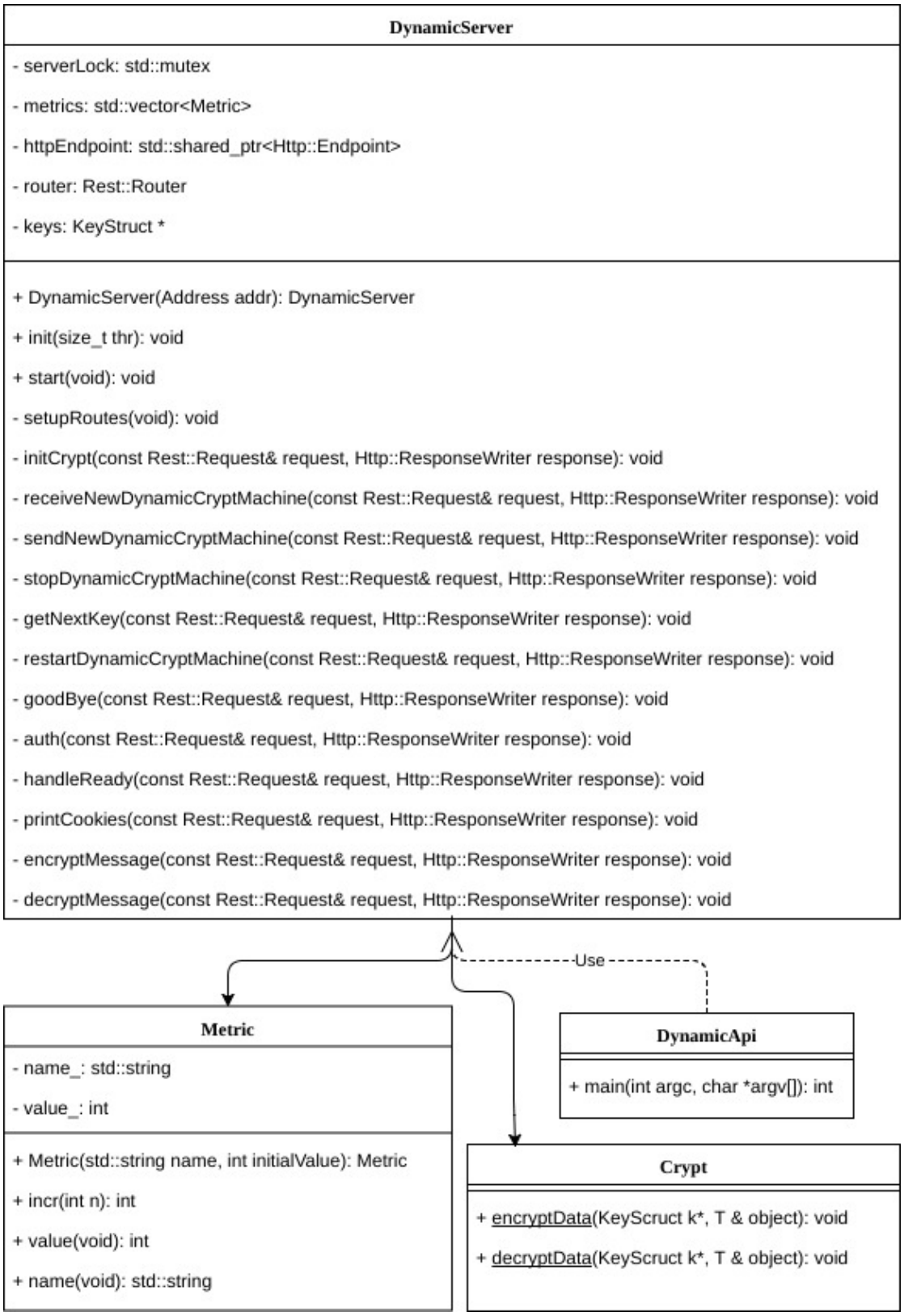

FIGURE 4.5: Rest API class diagram

The class diagram 4.4 demonstrates roughly how the API will look like. The variables at the top of the DynamicServer class are mostly used for the library. serverLock is

to prevent issues with threads changing the same variable at the same time. metrics is for measuring various server performances this uses the Metric class defined in the diagram. httpEndpoint is a pointer used by the framework internally. router is used for handling different routes in the setupRoutes function. keys is a struct array that has the generated encryption key from the tree parity machine, the id of the tree parity machine and a status whether it was sent of or not more properties will most likely be added in the implementation phase to this struct.

The functions after setupRoutes are all route handlers that will need to be associated with a route in the setupRoutes function. initCrypt function will tell the tree parity machine handler to setup the tree parity machines. receiveNewDynamicCryptMachine is when a different host setup the tree parity machines and sends you ids and other information to allow for partnering of this hosts tree parity machines. sendNewDynamicCryptMachine is the opposite of the previous this time the api is sending info of its tree parity machines to another host. stopDynamicCryptMachine this is used when the application believes it doesnt need any more new keys for dynamic encryption so the synchronisation will cease to avoid expensive unnecessary operations. getNextKey will send the next if available key to the application. restartDynamicCryptMachine would be typically called after stopDynamicCryptMachine to generate more keys once again. goodBye will be called when the application doesn't need to use the API anymore this will remove any references and any tree parity machines relating to the application. When the application disconnects from the API a similiar set of instructions will also occur. auth is when authorisation will be implemented for increased security. handleReady is more of a test function to determine if server is up and running. printCookies will most likely never be used. The encryptMessage and decryptMessage allow the user to pass in a message to be encrypted. The Crypt class will handle the encryption and decryption of such data. The KeyStruct struct will contain information about which tree parity machine generated the key so there will be no confusion when encrypting and decrypting between multiple hosts.

The Metric class is used for saving different server metrics. A metric has a name, value and increment amount.

Lastly the DynamicApi class holds the main method that initialises the server.

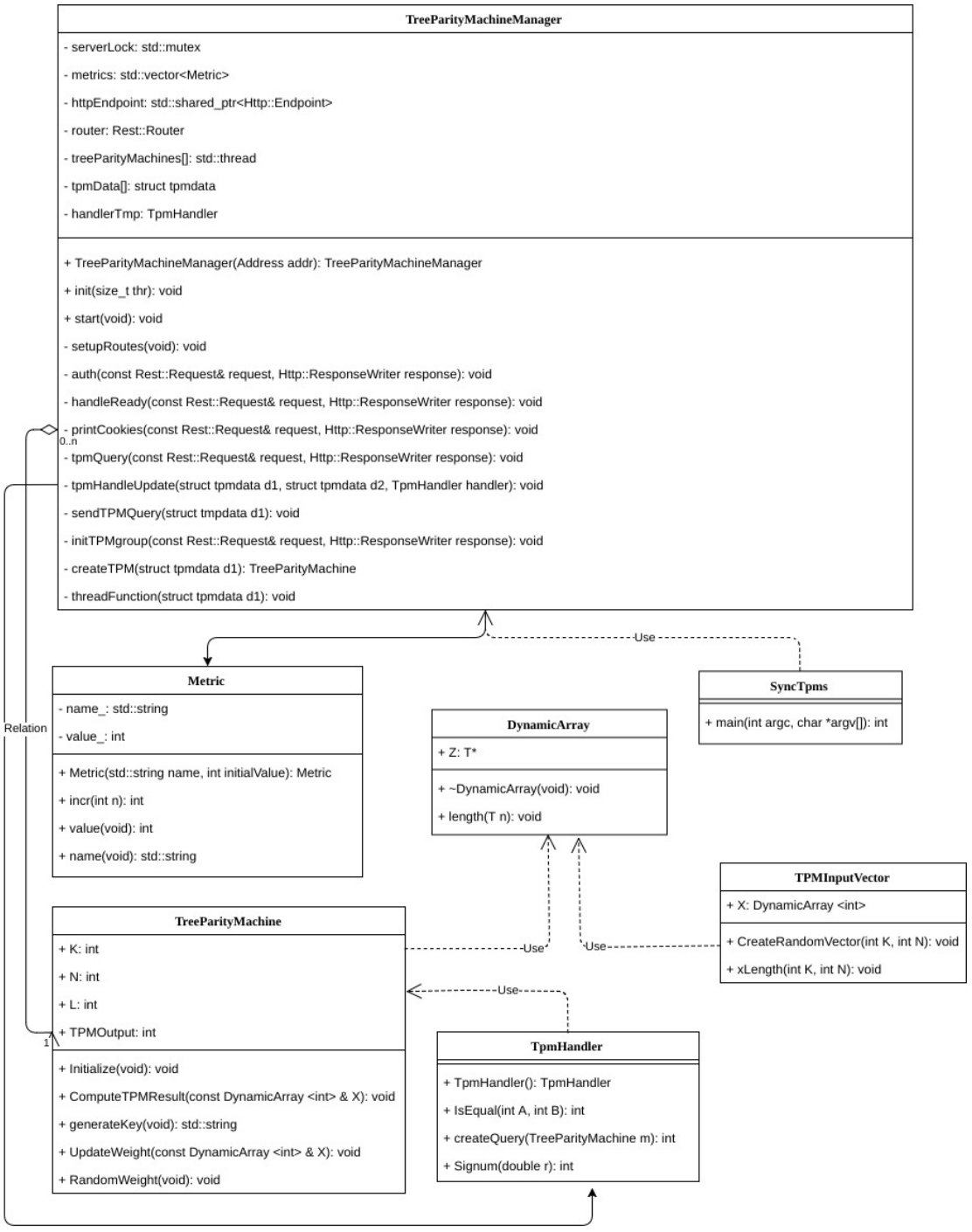

FIGURE 4.6: Tree Parity Machine Manager class diagram

This is a class diagram4.6 of the second application. There are two applications primarily because it will be easier and less strain on the API server if the synchronisation activities were running on a different server. These servers will communicate with each other via network protocols and or via shared memory which will be discussed later in this section. The TreeParityMachineManager class has similar variables and functions as DynamicServer from the previous diagram because both of them are implementing

the Pistache server framework. Therefore I wont mention the variables and functions required for the framework here again. This class may have to implement the main method as it requires multithreading but I will have to do some tests before I can finilise my answer.

The threads will be held in an array called treeParityMachines and each thread will execute the function threadFunction which will be used to process the tree parity machines. a struct tpmdata will be passed in to this function, this struct will hold the information and pointers for each tree parity machine and information of the thread that is executing that tree parity machine. The tpmQuery and sendTPMQuery functions will send and receive queries from different hosts to syncronise the tree parity machine. tpmHandleUpdate function is called inside the tpmQuery and sendTPMQuery functions and will update the tree parity machine weights if needed accordingly. initTPMgroup function will create a group of partner tree parity machines for each host that connects with the help of the createTPM function which takes a tmpdata struct and updates it when the tree parity machine is created.

The TreeParityMachine class is the actual tree parity machine itself. K,N,L are tree parity machine parameters these will be discussed later in this section. The TPMOutput is the output of the machine after a query is received. This class has some basic functions required for the tree parity machine to operate. Initialize will initialise the tree parity machine with random weights. ComputeTPMResult will update the TPMOutput variable after processing a query. generateKey willl generate a key based on the tree parity machines weights. UpdateWeight updates the weights if needed. RandomWeight sets the tree parity machines weights to random weights.

The TpmHandler class help out in synchronising the partner tree parity machine with the one locally stored on this host. IsEqual checks if the output bits of both syncronising tree parity machines is equal and therefore other functions can proceed to update weights or not. createQuery creates a query based on the tree parity machines weights. Signum is used for calculations.

The DynamicArray is used for an adaptable array needed for the TreeParityMachine class. The TPMInputVector class is a helper for creating a vector that is also used by the TreeParityMachine class.

And lastly the SyncTpms holds a main method which might be removed in the implementation phase as there could be issues with threads I will have to experiment to determine this outcome.

```
#include <boost/interprocess/shared_memory_object.hpp>
#include <boost/interprocess/mapped_region.hpp>
#include <cstring>
#include <cstdlib>
#include <string>

int main(int argc, char *argv[])
{
   using namespace boost::interprocess;

   if(argc == 1){  //Parent process
      //Remove shared memory on construction and destruction
      struct shm_remove
      {
         shm_remove() { shared_memory_object::remove("MySharedMemory"); }
         ~shm_remove(){ shared_memory_object::remove("MySharedMemory"); }
      } remover;

      //Create a shared memory object.
      shared_memory_object shm (create_only, "MySharedMemory", read_write);

      //Set size
      shm.truncate(1000);

      //Map the whole shared memory in this process
      mapped_region region(shm, read_write);

      //Write all the memory to 1
      std::memset(region.get_address(), 1, region.get_size());

      //Launch child process
      std::string s(argv[0]); s += " child ";
      if(0 != std::system(s.c_str()))
         return 1;
   }
   else{
      //Open already created shared memory object.
      shared_memory_object shm (open_only, "MySharedMemory", read_only);

      //Map the whole shared memory in this process
      mapped_region region(shm, read_only);

      //Check that memory was initialized to 1
      char *mem = static_cast<char*>(region.get_address());
      for(std::size_t i = 0; i < region.get_size(); ++i)
         if(*mem++ != 1)
            return 1;   //Error checking memory
   }
   return 0;
}
```

FIGURE 4.7: shared memory with boost[**?**]

The two applications that I am building are executed on the same host therefore it would be in my best interest to share a chunk of memory between them not only will this increase performance as accessing shared memory is much faster than using networking protocols. Using shared memory will also alleviate some of the stress that would be induced on the servers as they would need to communicate between themselves. To achieve this in a timely manner i will use the help of the boost [**?**] library which includes functions for shared memory management.

The source code for a simple implementation can be found on the boost website which is included in figure 4.7. One application would have to initiate a memory region with

```
mathshared_memory_object shm (create_only, "MySharedMemory", read_write);
```

The other server can then open this shared memory using the following

```
shared_memory_object shm (open_only, "MySharedMemory", read_only);
```

Before using the shared memory it must be mapped to the process therefore both processes will need the following

```
mapped_region region(shm, read_only);
```

Lastly to make this project easily adaptable I will be making a NodeJS module or specifically middleware compatible with ExpressJS [**?**]. Middleware in NodeJS is functions that take the request or response and process it in some way or form and return or send the modified request or response to the user see figure 4.8. My project is definitely usable without this module however this will simplify things even further for people who wish to use this project with NodeJS.

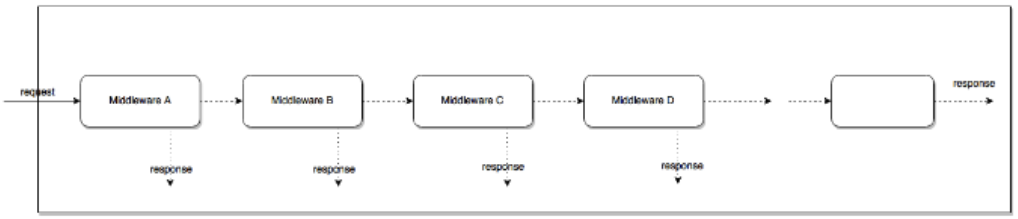

FIGURE 4.8: middleware in express [**?**]

This middleware will be mostly interacting with the DynamicServer API and will use promises that will return various information when said information is acquired, it might take a while for the initial key to arrive so I believe promises would be the best way to go.

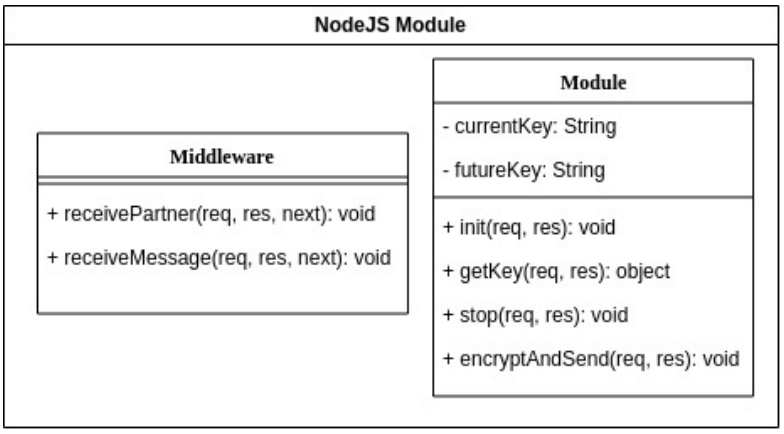

FIGURE 4.9: NodeJs Module diagram

The diagram 4.9 is a bare bones representation of what the NodeJS module will look like. I will be doing this part last during the implementation phase so a lot will change by then and therefore this isn't described in too much detail. Since JavaScript can't really be represented in a class diagram I have split the methods onto middleware functionality and everything that is not middleware is represented in the module class.

The methods in the middleware are as follows: receivePartner this will perform actions if the request is part of the dynamic encryption and specifically the one that asks the server to use dynamic encryption if the request is anything else the middleware will just call next() and the next middleware in line will perform functions. This function will call the init route in the API with the information of the partner and perhaps more functionality will be required during the implementation phase. This functionality needs to be implemented as middleware since the other host can request to use dynamic encryption with this host at any time. receiveMessage will detect if the response is using dynamic encryption and will decrypt that message and update the response object with the decrypted message.

The methods in the module are as follows: init will send a request to the init route in the API like the receivePartner method but without the partner information. getKey will request a new key from the API. stop will stop all syncronising tree parity machines for when dynamic encryption is not needed anymore. It is possible to call init once more to continue syncronisaton. encryptAndSend encrypts a passed in message and sends it to the other host. There are two variables in the module currentKey and futureKey. When you call getKey there may be encrypted messages that arrive at a later time and therefore will use currentKey for decryption and newer messages will use futureKey.

4.2 Risk Assessment

Creating a reasonably complex project leads to a possibility of risks that may potentially cause my project to fail. This involves risks such as technical issues, implementation issues and design issues.

The amount of data transferred between each tree parity machine might overwhelm the servers due to many tree parity machines synchronising at once. This can lead to very slow operation or servers crashing therefore I may have to run multiple instances of the servers or reduce the number of tree parity machines.

This project uses a lot of specific algorithms that are projected through mathematical formulas. Failure to understand or implement these formulas into code can cause a delay or failure to complete some functionality.

The framework that I'm planning on using for the rest APIs might not be suitable for the task. This is unlikely to happen as I have researched the most responsive rest frameworks. Pistache seems to have the highest request per second processing time.

This project relies on having a stable network to send data to and from different hosts. A network failure would cause the synchronisation process to cease. Or a congested network will slow down the synchronisation process to an unusable state where it would take a very long time to synchronise due to a plethora of lost packages.

A number of C++ libraries are used for this project as well as some NodeJS libraries that could potentially be used for the NodeJS module. The availability of these libraries could cease or they could become closed source will cause issues in finding alternatives.

C++ and JavaScript in particular the NodeJS library are the main languages used in this project if they become unavailable I will have to research and implement the project in other languages.

Due to this project being self managed poor planning such as task estimation time or spending too long on a single task may cause the project to be not finished at the deadline set.

This project is built using primarily one device which can lead to laziness with backups. The project should be backed up to an external hard drive or because I am using git for version management I will be occasionally pushing it to GitHub.

4.3 Methodology

I employed a various range of techniques while finalising the background chapter of this paper. I initially researched any reference or any details about dynamic encryption and discovered that it is a mildly discussed topic. A closed source product explained briefly that in order to achieve dynamic encryption they use wrappers. I didn't find wrappers very interesting. And because wrappers at the time of my research were the only method of achieving dynamic encryption publicly available I decided to look at the problem from a different perspective.

In order to achieve dynamic encryption the key must essentially be changed this can be done many ways by sending the key over the network however my goal was to not send the key over the open network, at least not directly.

I researched topics about key generation and other unconventional ways of sharing keys. I came across neural network synchronisation which can be used to generate a key without directly sending sensitive information over the network.

The only information about this topic I could find was in research papers, luckily there was a good number of relevant papers with a reasonable variety of information available. All of these were more or less proof of concept as there were no products or examples based of these. After reading and understanding the similarities between these papers I was able to commence chapter 2.

While constructing chapter two I outlined what is needed for a tree parity machine to operate and the differences between the possible tree parity machines such as different learning algorithms and the benefit of using queries instead of random inputs all while referencing the resources I had available before hand and using relevant points and diagrams from those resources.

Most of the computer science skills required for this project I already posses like building a NodeJS module. However there are many aspects of the project which are new to me. Starting of with the main language C++, which I do not know as well as C however I have used C++ many times in the past as I find multithreading and sleep style functions to be more diverse in C++ as well as the fact that C++ is object orientated making C more difficult to use for larger projects. My C++ skills will need to be expanded a little bit more to complete this project. I have never wrote a rest API server using anything other than NodeJS therefore writing it in C++ is new to me, however I have researched a number of C++ rest frameworks and found Pistache to be similar enough to what I recall from NodeJS and was able to reasonably understand and execute the example server code provided on the Pistache GitHub. I will have to experiment with incorporating custom

multithreading functions with Pistache as that might cause issues with multiple clients connected. I will also attempt to implement memory sharing between two applications to minimise the stress on the network part of the programs. I have never used shared memory in the past however the library boost provides methods that can be used to allocate a block of memory to be shared amongst applications. I am able to use and understand the basic example however the memory there is constantly the same size which will not be the case in this project so I will need to experiment with this.

For this project I will agile mythology in particular a sprint based one, where I will undergo a two week sprint followed by a retrospective and planning for the next sprint. I have chosen to follow this technique as that is what was used during my work placement in a well known company.

It is also very unlikely that a project can be fully created just from class diagrams, ideas and functionality all at the very beginning. Issues and further functionality may have to be resolved and introduced in order to provide a working product, there is no way to predict the exact way to develop software without actually making a basic version and expanding on it until all the features required are implemented and fully working.

This technique is commonly used for projects will multiple people however there is no harm in following this even if the project only consists of one member. I believe it is important to follow industry standard methods as much as possible since these methods are developed and used by the top companies on earth therefore using these methods must be the best way to accomplish projects as of writing this paper.

4.4 Implementation Plan Schedule

I will be using agile methodology in particular two week sprints followed by sprint retrospect and sprint planning after each sprint is completed. Therefore it is difficult to put a schedule as the next sprint is usually planned after and typically based on the outcome of the sprint before hand. Bellow is a rough description for each sprint if everything goes according to plan.

TABLE 4.1: Sprint Plan

Sprint	Tasks
1	Expand prototype to work with two process instances instead of just in the same process. Can use normal files to communicate.
2	Implement the use of queries instead of random inputs.
3	Create basic rest server and experiment with multithreading and multiple clients.
4	Extend functionality of server to mock synchronisation between identical server on a different hosts. e.g. send incremental data to and from each host.
5	Replace mock with a real tree parity machine.
6	Add support for multiple tree parity machines. This will need to be stress tested and load tested to ensure that the server lasts under heavy conditions.
7	Develop the API server for communicating with local host servers.
8	Develop NodeJS module.
9	Implement and allow multiple instances to run for performance and stability increase.
10	Thoroughly test system including NodeJS module and fix any remaining bugs.

4.5 Evaluation

The evaluation of this project can be conceived by examining if the functional and non functional requirements are met. The goals aimed to be achieved by this project will also be compared with the final project.

The final project should be able to successfully synchronise with another instance preferably running on a different host for realism and be able to synchronise using multiple tree parity machines.

The project should not be vulnerable to tree parity machine attacks, this is unlikely as using queries will negate any possible attack methods known.

The communication between the tree parity machine manager and API should be secure and robust.

An encrypted message send from host 1 should be successfully decrypted after it arrives to host 2.

The NodeJS module should be easy to use and can be implemented in any NodeJS project provided ExpressJS is used.

4.6 Prototype

The prototype can be found on my GitHub here [**?**]. This prototype borrows pieces of code from here [**?**] and demonstrates a very simple synchronisation method between two tree parity machines. The Prototype consists of two tree parity machines that synchronise with each other using random input vectors. The finial version of the tree parity machines will use queries instead of random input vectors this will result in faster synchronisation time and tree parity machine attacks will not be possible when using queries making it secure. The prototype has both tree parity machines running in the same main function so there is no networking involved thus the two tree parity machines can be accessed easily and freely.

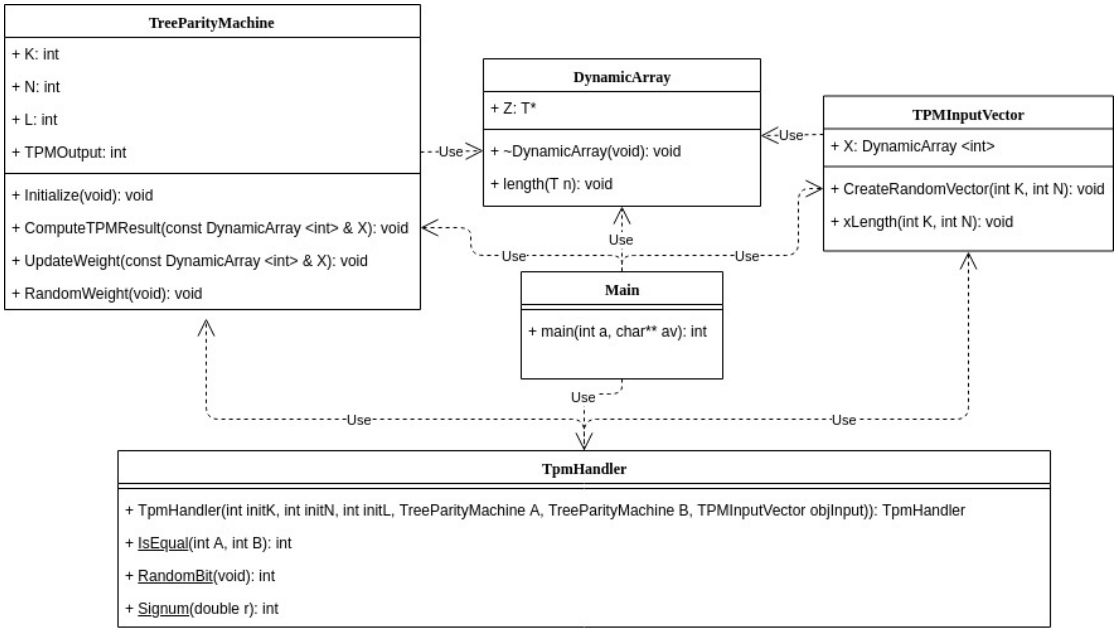

FIGURE 4.10: Prototype class diagram

This is a class diagram of the prototype 4.10. Since the two tree parity machines synchronise with each other in the same process the diagram is quite small.

```
Dictionary size 38
Maximum Iterations: 171072
Synchronizing TPM Networks...Status: SUCCESS! after 16594 itterations
DataExchanged: 2203 KiB
Key length: 66
Key: 1bg7ladno1fai44cnlaeh228bb5e19_f8cd57dc8dnlmn47m8m_394k_elmkdgod2_
```

FIGURE 4.11: Prototype in action

Figure 4.11 shows a successful synchronisation between the two tree parity machines. This run uses the following tree parity machine parameters 12 input neutrons, 11 hidden neutrons and a weight range of 6 meaning the possible value for each weight element is from -6 to 6. The Dictionary size refers to an array from which different elements will be picked to build the encryption key seen on the last line.

```
34   const char Dictionary [] = "0123456789_abcdefghijklmnopqrstuvwxyz";
```

FIGURE 4.12: Dictionary used

The code snippet 4.12 contains all the possible characters possible that will make up the key.

Maximum Iterations is the maximum times the tree parity machines will receive input in order to synchronise. This is used just in case something goes wrong the tree parity machines will not be trying to synchronise forever. The Maximum Iterations is calculated based on the input neutrons, hidden neutrons and the range of weight values. The Maximum Iterations is calculated in this case to be 171,072 which is quite a lot however the tree parity machines manage to synchronise way before that number is reached. In this run the tree parity machines synchronised successfully after 16,594 iterations. DataExchanged is an estimation of the amount of data required to synchronise the two tree parity machines. This is mostly affected by the number of iterations taken to synchronise the more iterations the higher the data. Using the above stated tree parity machine parameters a key of length 66 is produced, increasing and decreasing the tree parity machine parameters will have an impact on the key size.

```
Dictionary size 38
Maximum Iterations: 171072
Synchronizing TPM Networks...Status: SUCCESS! after 11759 itterations
DataExchanged: 1561 KiB
Key length: 66
Key: agmcbm41df6h65ci2j24lm5519f3c9ccdf4dkgaa22kc1i7b_ec2le4c5odon5chec
```

FIGURE 4.13: Prototype in action

Figure 4.13 shows another successful synchronisation between the two tree parity machines using the same tree parity machine parameters as the previous example. The key in this case is obviously different and so is the number of iterations required to

synchronise. This also means that the amount if estimated data is less in this case it is 642 KiB less.

By changing the tree parity machine parameter variables to lower values of 8 for the hidden neutrons, 10 input neutrons and a weight range of 4 the encryption key is now 20 characters long. Figures 4.14 and 4.15 are executed using this configuration.

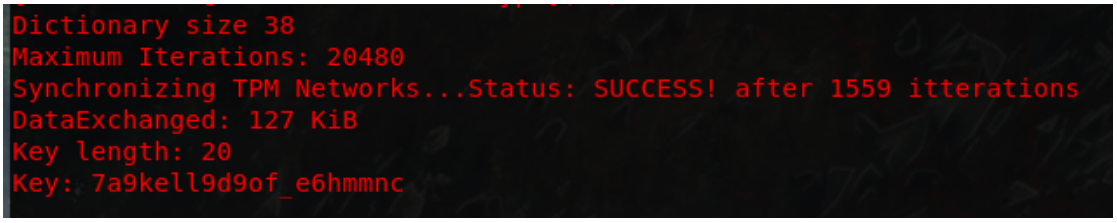

FIGURE 4.14: Prototype in action with a key size of 20

Because the number of neutrons is reduced so is the number of weight and therefore the number of iterations required for synchronisation is reduced to only 1,559 in figure 4.14 and 3,661 in figure 4.15 which is considerably smaller than 11,759 and 16,594 seen before.

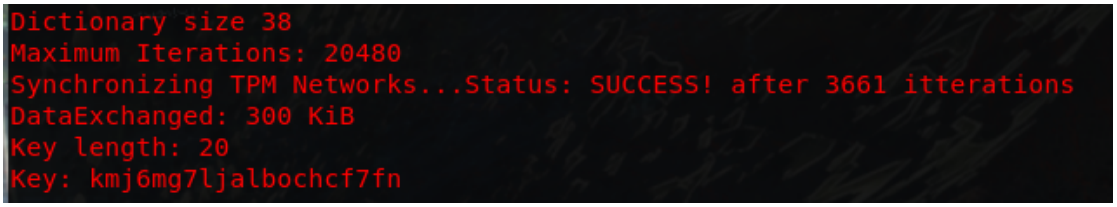

FIGURE 4.15: Prototype in action with a key size of 20

The data exchanged is also significantly reduced with the fewer number of iterations with only 127 KiB in 4.14 and 300 KiB in figure 4.15 compared to 2,203 KiB and 1561 KiB in the previous two examples.

In order for synchronisation to occur the weights of both tree parity machines must be identical since the generated key is based on them. The weights are essentially an Array or specifically they are an DynamicArray object and thus can be represented.

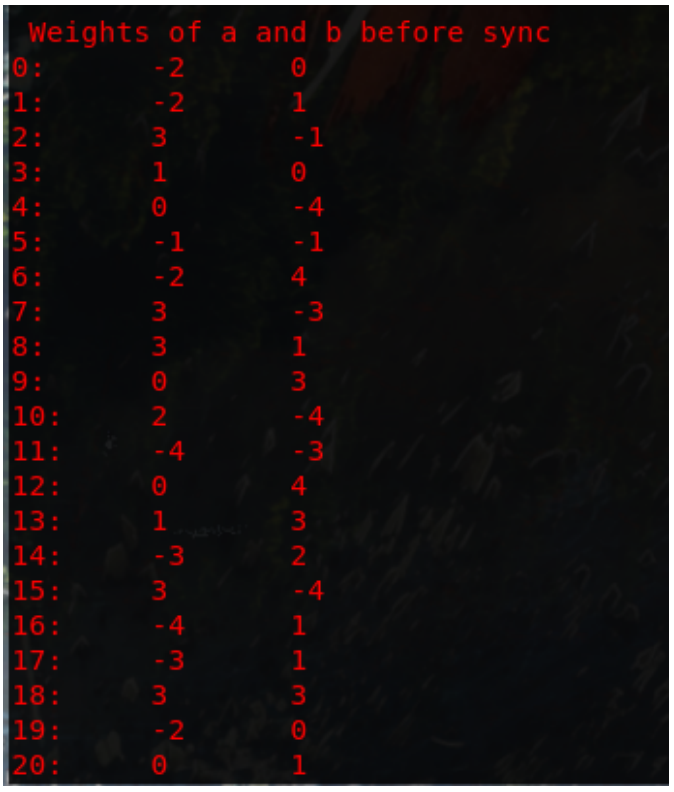

FIGURE 4.16: Weights of A and B before synchronisation

The figure 4.16 is the representation of the fist 21 weight values, there is a total of 80 weight values because this run is using 8 hidden neutrons and 10 input neutrons and therefore the size of the list of weight values is 8 * 10. The first column just shows the number in the list for which this weight is associated with. The second column is the weights of the tree parity machine A and the third column is the weights of the tree parity machine B. These weights are displayed before the synchronisation process begins and as you can see they are nothing alike. Note that the maximum and minimum values you can see is -4 to 4 this is because the weight range for this run is set to 4.

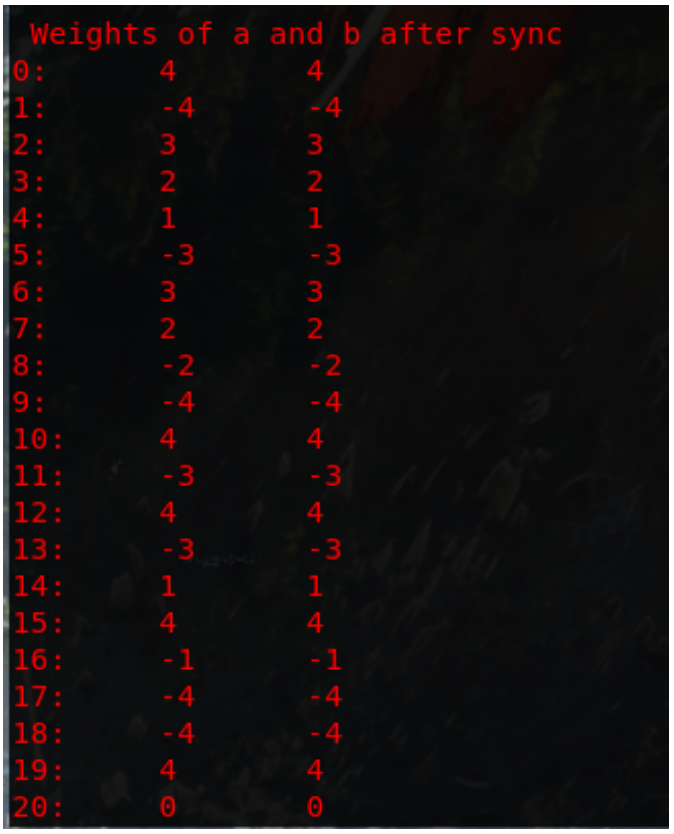

FIGURE 4.17: Weights of A and B after synchronisation

The figure 4.17 is taken from the same run but the weights are displayed after the synchronisation process and as you can see the weights of both of the tree parity machines are now identical therefore they are able to produce the same key.

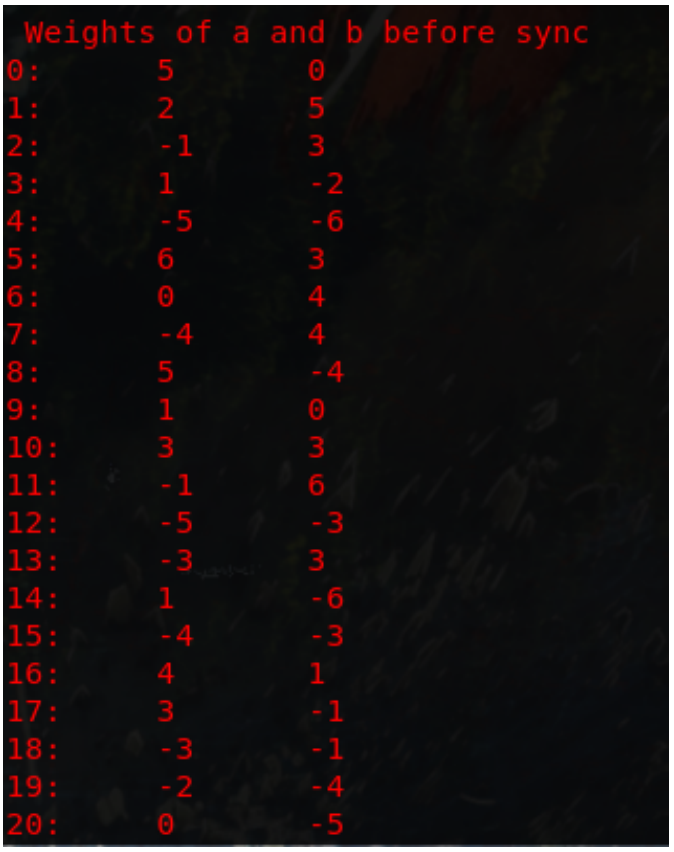

FIGURE 4.18: Weights of A and B before synchronisation with an increased range

The figure 4.18 is displaying the weights of both tree parity machines before synchronisation just like before except this time the weight range is increased to 6 that's is why the minimum and maximum values you can see this time is -6 to 6. Again the weights are completely different from one another as they are initialised randomly at the beginning. In the prototype time is used to seed the random generator this will not be acceptable for the final project as time can be easily guessed.

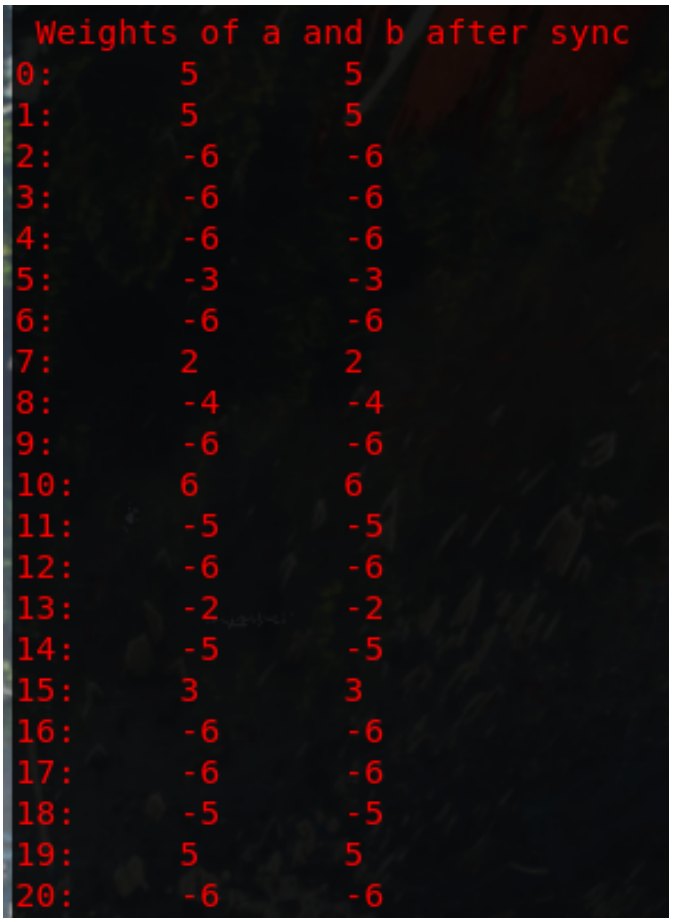

FIGURE 4.19: Weights of A and B after synchronisation with an increased range

The figure 4.19 is the weights of the same tree parity machines as in figure 4.18 but after synchronisation and as you can see the weights are now identical. The reason one might choose to increase the weight range is because these weight are used in calculations and give a little bit more diversity. Increasing the weight range can also increase the length of the encryption key. Figure 4.16 uses the same parameters as 4.19 but with a weight range of 4 instead of 6. The key length of 4.16 is 20 and the key length of 4.19 is 40. In my testing I found that there isn't really any benefit in increasing the weight range past 6 as anything above 6 doesn't increase the key length by much and increases the number of iterations by a good bit. I have found that it is better to increase the number of hidden neutrons if you need a larger key length followed by the number of input neutrons and just leave the weight range at 6.

Chapter 5

Implementation

The implementation of the system has changed drastically since the original plan due to an oversight where it was believed that the Pistache framework could handle the functionality for both the API and the synchronisation between the tree parity machines. The project turned out to be much more involved and difficult than originally expected. This means that the original sprint plan is no longer representative of the actual sprint plan used. This is however to be expected in an agile environment where the following sprint is normally determined after the evaluation of a sprint that was just completed. Despite all of this the system is functional and produces the same outcome as intended.

5.1 Difficulties Encountered

The major difficulties I encountered were primarily related to the architecture of the project. The initial class diagrams were way too basic and didn't account for the fact that Pistache would now only be used for the API and nothing else.

Easy Difficulty

1. The NodeJs module was replaced with simply a NodeJs App. The NodeJS module is not implemented due to a time constraint and is not necessary to the outcome of the project. The NodeJs module was originally meant to simplify the communication with the API for people who wish to use it. A NodeJS App would need to be implemented regardless in order to use that module and demonstrate the usability of the API. The API doesn't have many routes to use therefore it would not be difficult to use the API with any NodeJs App as interacting with the API is simply done with POST requests. The final NodeJs App that is in the

same GitHub repository, consumes the API perfectly as well as purposely demonstrating some aspects of dynamic cryptography that would normally be hidden. Any user that wishes to use the API can simply copy the functions in the NodeJs App that interact with the API and change them accordingly to send and receive their own data. It is recommended for users to copy said functions as there are specific steps that need to take place to register the NodeJS App with the API in particular the synchronisation service. For this reason The NodeJs App uses pure NodeJs libraries like ExpressJs [**?**] which is the most popular routing framework for NodeJs. The NodeJs App has no front end as it would be difficult for users wishing to use the API to adapt the front end too, therefore only a basic HTML template engine is used to deal with the HTML. JavaScript for the front end is purely used for visual purposes and is not required since all of the information sent from the browser to the NodeJs app is done through classic HTML forms. This doesn't have any impact on the architecture as the NodeJS App is not really included in the architecture as it is simply designed to consume the API. This does impact the schedule in a positive way as it is easier and quicker to simply make a NodeJS App than a NodeJs module and a NodeJs App.

Medium Difficulty

1. The sprints needed to be adjusted to accommodate the new implementation plan. The final sprint plan will be placed here as it briefly reflects the major changes that took place, these changes will be discussed in more detail in the following sections. The Sprint plan follows a two week sprint approach.

Table 5.1: New Sprint Plan

Sprint	Tasks
1	Create a basic API server that can process GET and POST requests. Familiarise self with RapidJSON library [**?**] and create functions to parse C++ objects into JSON "strings" and JSON "strings" back into C++ objects. Research peer to peer libraries, no useful ones were found. Decided to make a custom one with the help of the Boost ASIO [**?**] library. Never used Boost ASIO before therefore a decision was made to follow the developers tutorials on the Boost ASIO website. This was not enough to understand the complicated library therefore other tutorials such as this one [**?**] were also followed.
2	Further knowledge of Boost ASIO needed to be acquired before it was possible to proceed with coding the actual peer to peer network. Thankfully a very informative book called Boost.Asio C++ Network Programming [**?**] by John Torjo was purchased and ended up being the last resource needed in order to complete the peer to peer network. A basic peer to peer network was written however was unstable at the end of this sprint
3	Peer to peer network works as expected. Implemented basic synchronisation with real tree parity machines. Implemented command line parsing using Boost Program Options [**?**] to enable different options such as selective outputs and port configurations. Completed synchronisation between tree parity machines fully therefore they produce the same weights successfully. Implemented "logging" to external terminal windows.
4	Research how to proceed with the API i.e have two separate processes or contain the API and peer to peer network within the same process. Decision was made to contain both in the same process for security reasons. Basic API server was constructed and could be easily spawned along side the peer to peer service.
5	Implement API functionality for synchronisation. Changes needed to be made to the peer to peer service to accommodate unexpected requirements needed to begin synchronisation when using the API. AES encryption decryption implemented using Crypto++ [**?**] library. Encrypted data needed to be encoded using base64 again with the help of Crypto++ library. Implement a NodeJs app to consume and demonstrate various features of the API.
5	Test the system. Implement options to choose which parts of the system to log since logging everything at once can be difficult to demonstrate different aspects of the system. Clean up the code. Document instructions on how to setup the system in the GitHub Readme file.

2. The API and the synchronisation could not be done using only the Pistache framework as originally believed. This was a fairly significant set back since I have experimented before with Pistache framework and was familiar with how it operated functionality wise. It was only until I attempted to use Pistache for synchronisation I discovered that my original plan will not work since Pistache is a very good REST framework and not much else. To synchronise the tree parity machines a peer to peer network seemed most appropriate in how the synchronisation was envisioned to take place. After searching for easy to use libraries that supported peer to peer networks nothing that matched the requirements was found. This lead to the conclusion that a custom peer to peer network would have to be built from scratch. The attention grew towards Boost Asio as it is the most flexible generic networking library for C++. Unfortunately this library is quite extensive and is not as abstract as some of the previous peer to peer libraries encountered. Since with Boost Asio you will still deal with sockets however this will allow for greater control provided you know how the library works and Boost Asio has excellent asynchronous support which is essential for a responsive server. It took almost two full sprints to learn how the library works which set my initial schedule back as it was planned to spend time to learn the new technologies that would be used however it was not expected to spend so much time learning said new technologies. However some of the lost time was made back since the design of the peer to peer network was simple and efficient which allowed for a large number of synchronisation requests to take place per second resulting in generating a valid key in around one second which satisfies the requirement and no further optimisations needed to take place. This occurrence changed the architecture significantly since originally this wasn't meant to be present. Overall I'm glad that this happened as I had a chance to learn one of C++'s most popular networking library. Another benefit of using Boost Asio for synchronisation over Pistache apart from the fact that its impossible to achieve using Pistache is because the Boost Asio library is a lot lighter the performance increase is rather large. Obviously this would be impossible to test fairly however based on Pistache stress tests it can manage roughly 300 - 400 requests per second. The peer to peer Boost Asio can process around one thousand requests per second as it takes around one thousand requests for the tree parity machines to synchronise and they synchronise generally once per second.

Hard Difficulty

1. A major difficulty that has prevented the implementation of a non functional requirement was the ability to use multiple tree parity machines per peer. Originally

the plan was to implement multiple tree parity machines per peer, however in practice after spending more than a week on the problem a decision was made to move on and only allow one tree parity machine per peer. This theoretically does not have any impact on performance and certainly has no impact on security. A single tree parity machine per communication works as expected, a peer can have multiple tree parity machines where for example peer1 communicates with peer2 and peer3 at the same time. In this scenario peer1 would have two tree parity machines as the peer will communicate with two other peers. This scenario works flawlessly in the final implementation of the project. If the original plan were to be implemented peer1 would have twenty tree parity machines, ten per peer it communicates to. The reason why this is difficult is because the ten tree parity machines are reading and writing to the same socket. Whereas with one tree parity machine it has exclusive read and write access to the socket. An attempt was made to implement multiple read and write buffers, a queue, custom read and write objects however a decision was made to abandon this feature for now as it took up too much time and didn't add anything beneficial to the project apart from "being cool". The architecture of the project can be said to have changed although it is insignificant as the number of tree parity machines is rather virtual in nature. This issue did not represent any major risk to the project as everything works perfectly with one tree parity machine per connection. The implementation schedule was affected as a lot of time was spent on a feature that did not get implemented and it is better to spend time on features that actually do get implemented.

5.2 Actual Solution Approach

Most parts of my original approach to this project have changed vastly in particular the architecture. Every major change will be explored in the following sub sections.

Architecture

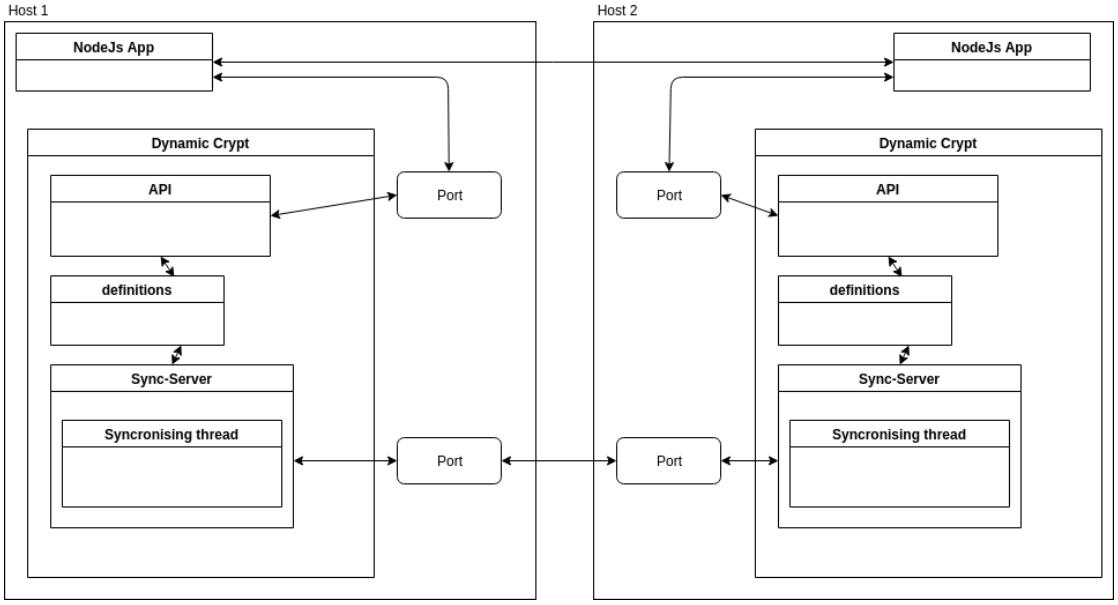

FIGURE 5.1: High level overview of architecture

The high level overview of the architecture changed slightly as shown in figure 5.1. The functionality remains identical as described in the previous chapters. Initially the API and the server handling the synchronisation of tree parity machines executed in the same sort of architecture where the API would spawn threads for each tree parity machine. For the final implementation the API and the sync-server are separated into two servers. These two servers execute in the same process but obviously run on different threads. It was impossible to accomplish the task by using the API server alone as initially believed. This is because Pistache is limited to only API type operations as it is at its core built to code API rest applications.

In order to be able to synchronise with any computer in the world the sync-server needed to be of a peer to peer nature. How this all works and why it is necessary will be explained in more detail later in chapter 6. Initially the API would spawn the tree parity machines due to the introduction of a new system this is still partially true however in the final implementation the API has access to a global function defined inside definitions.cpp that creates an instance of a peer object and connects to another listening peer. When the DynamiCrypt app executes an API instance is created and an instance of a listening peer is also created. This way the listening peer can accept any incoming connection and when an incoming connection arrives another peer is created and listens for more connections this way it is possible to theoretically have as many peers as possible synchronising provided the hardware supports it.

By default the API is setup to use two threads this allows for multiple asynchronous requests to the API to be made with minimal wait time. The sync-server has a more

much more heavy workload potential therefore by default it can use up to four threads. The sync-server by design is also asynchronous in operation therefore all of the four threads would most likely not be used at once.

Initially there was not much of a consideration of how the API would communicate with the sync-server as it was overlooked since the system was vaguely defined at that point. However in the final implementation this is handled mostly through a global APIServiceDataHandler object that stores data relating to the NodeJs apps and contains the key store for each app "using" to the API.

FIGURE 5.2: DynamiCrypt Class Diagram

Figure 5.2 represent the class diagram for DynamiCrypt as a whole. It is quite hard to make out the different elements since the class diagram is rather large therefore for the next sections this diagram will be split into mostly the API related classes and mostly sync-server related classes. This is here to briefly demonstrate how different classes interact with each other and how the the API interacts with the sync-server through mostly functions defined definitions. As you can tell this is nothing like the initial class diagram since to implement the system required a completely different solution that was not present in the initial plan.

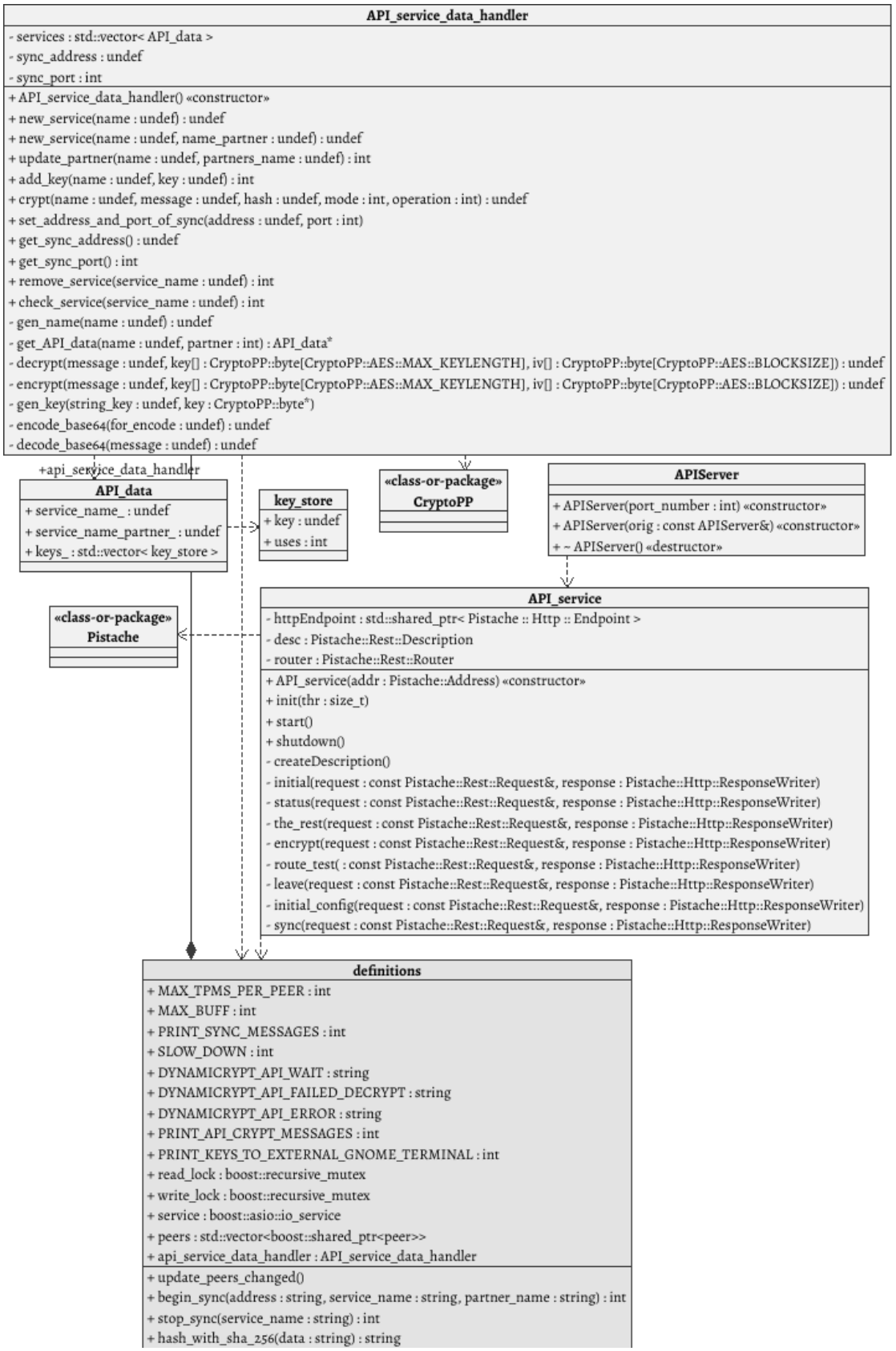

FIGURE 5.3: DynamiCrypt API Class Diagram

Figure 5.3 is a more clear and isolated representation of most of the classes used by the API. The APIServer is more of a wrapper class for APIservice as it sets up necessary Pistache objects required to use the Pistache REST framework and creates a fresh APIservice instance. The APIservice class itself is responsible for all the GET and POST routes available for the API and also deals with parsing JSON data as well as calling appropriate functions inside the API-service-data-handler object.

The API-service-data-handler object is defined as a global object inside definitions.cpp as it needs to be accessed by both the sync-server when saving keys and the API when the API passes data for encryption and decryption as well as information about different services that want to use the API. For encryption and decryption the Crypto++ library is used, as well as for encoding and decoding to base64. The API-service-data-handler object has access to a key-store or sort of like an in memory only mini database that holds information about the services that are currently accessing the API and generated keys for said services, This is essentially a vector containing references to the APIdata object which in itself contains a vector of the keystore object. The API-service-data-handler object is called from within the APIservice when new services require the use of DynamiCrypt. The API-service-data-handler object also contains operations for custom dynamic encryption and decryption operations which will be discussed in detail later as it is quite interesting how they are designed.

The definitions is a place that contains the applications global functions and variables such as defining the maximum size to use for the buffers, read and write locks for shared objects and other constants for defining which log messages to print. This also contains a vector of peers since by default there will be one peer listening for connections. The beginSync function creates a new peer which connects to a listening peer on another DynamiCrypt app. So each time a peer accepts a connection from another peer a new peer will be created to listen to more connections in its place and all of these peers will be added to this vector and can then be removed accordingly when a service decides to stop using DynamiCrypt. This occurs when the stopSync function is called which is called directly from the APIservice. Definitions also has a function for hashing strings as that is commonly used by the APIservice in the encrypt route.

The next two class diagrams 5.4 and 5.5 are both part of the sync-server classes so basically everything else that is not the API. However these are quite large therefore they are split into two separate diagrams. An attempt was made to make them look as continuous as possible where in the first diagram you can see relation lines going down of the page and continuing in the second diagram.

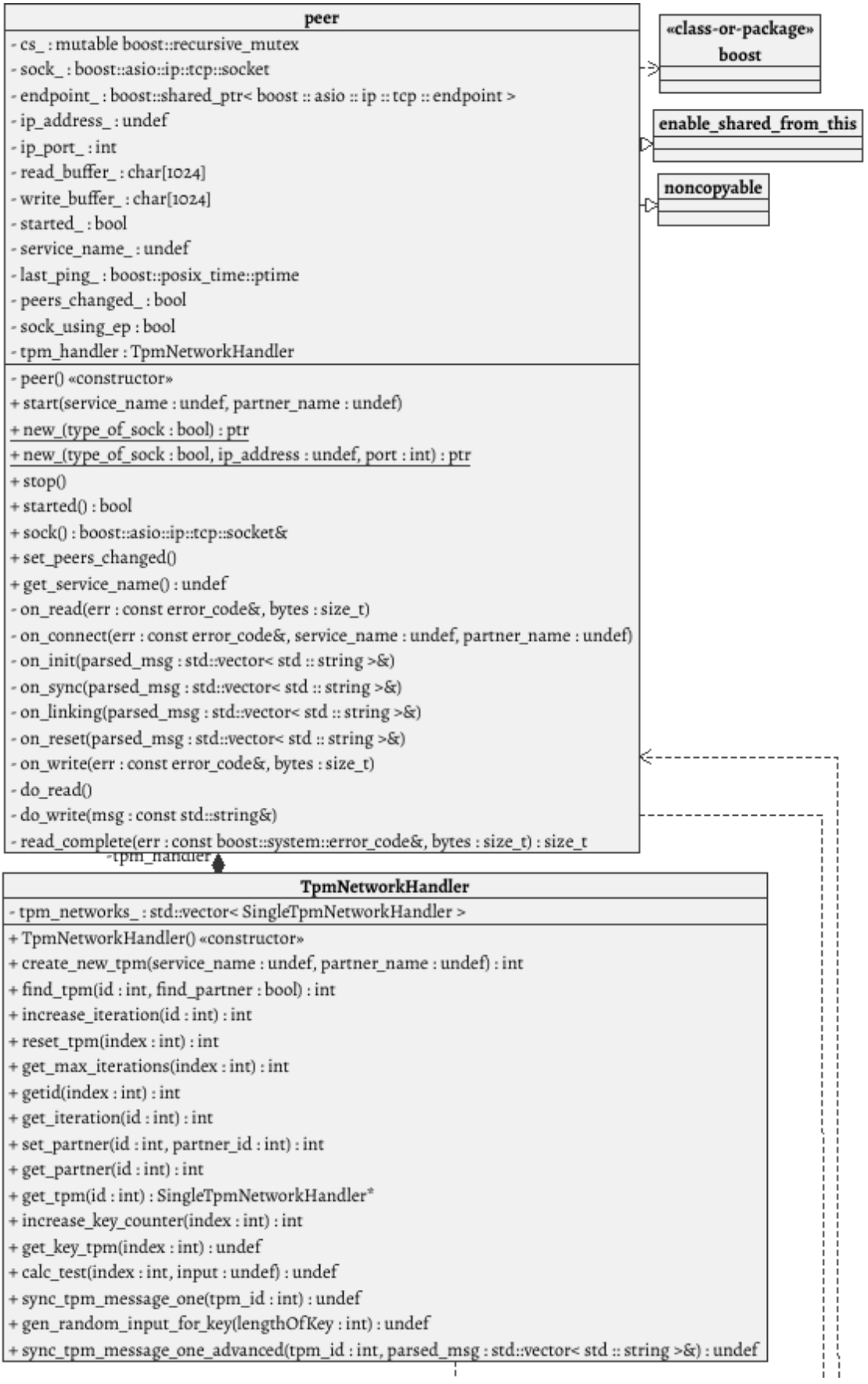

FIGURE 5.4: DynamiCrypt Class Diagram part 1

Figure 5.4 shows mostly the peer class and the TpmNetworkHandler class. The peer class contains the operations that read and write data to the socket and send it over to another peer. A more in dept explanation will follow in chapter 6 as to how data is read and written between peers. For now a basic explanation of the class will take place. The peer class uses Boost Asio to help with networking. The peer class implements interfaces such as enable shared from this and noncopyable as can be seen on the right hand side of the diagram. Enable shared from this allows the peer object to be referenced by a smart pointer, which is handy when iterating through peers in a loop, a smart pointer is used because peer is an asynchronous class and if using a standard pointer there could be an issue with de-allocation of memory. Noncopyable prevents objects of this class to be deep or shallow copied, this is mainly because this class has direct access to a socket and copying sockets is a really bad idea. Therefore peer objects should be only referenced by smart pointers and nothing else which noncopyable makes sure of.

As stated before peer is actually part of a peer to peer network hence the name however in this case the peer to peer network is very simple as it is a direct connection from one peer object to another, there is no such a thing as peer discovery or anything of that nature. In the thesis I mentioned listening peer and connecting peer multiple times. This is because how this object is initialised depends one the type of peer that is created. If initialising a peer in the following way.

```
ip::tcp::acceptor acceptor(service, ip::tcp::endpoint(ip::tcp::v4(),
    listen_port));
peer::ptr initial_peer = peer::new_(false);
acceptor.async_accept(initial_peer->sock(),
     boost::bind(handle_accept,initial_peer,_1, &acceptor));
```

This peer will listen for other peers that connect to it. This is used exclusively in the main function after setting up the API server. And if initialising a peer in the following way.

```
peer::ptr initiating_peer = peer::new_(true, address, port);
initiating_peer->start(service_name, partner_name);
```

This is a connecting peer which connects to the address and port specified in the parameters of its creation. The servicename and partnername are also passed in these are the details of the service that requested the use of DynamiCrypt through the API.

The rest of the functions deal with the different types of messages generated by each peer. Currently the messages are simple TCP data packets that indicate the type of message by simply by a number. More on this will be discussed in chapter 6.

The TpmNetworkHandler object exists for every peer and accompanied by a vector of SingleTpmNetworkHandler objects seen in figure 5.5 takes care of the tree parity machines required for the peer. At the moment really only one tree parity machine is used for each connection between peers however provided I can fix the multiple buffer issue discussed earlier multiple tree parity machines could be used straight away for each peer. The TpmNetworkHandler object handles accessing, updating tree parity machines, generating input vectors, testing tree parity machines outputs as well as dealing with the more complicated TCP messages that were best left out from the peer class. The reason these were left out of the peer class is mostly because they are really long and require multiple accesses to tree parity machines therefore it is more efficient placing them inside TpmNetworkHandler where there is less references to go through.

The SingleTpmNetworkHandler object in figure 5.5 is quite similiar to the TpmNetworkHandler object in terms of some of its methods however it is only in charge of only one tree parity machine and therefore accesses it directly whereas the TpmNetworkHandler object would select the right tree parity machine to use and call methods found in the SingleTpmNetworkHandler object. The SingleTpmNetworkHandler object is also responsible for logging of important information about the tree parity machine it handles to an external terminal, such as how long it took to generate the key time wise and how many many synchronising iterations it took to to generate said key.

The SystemHelper class provides a static method that helps to launch the external terminals. This class will be expanded in future versions of DynamiCrypt to allow for more OS specific functions such as running under its own user for more security.

The TpmHandler, TPMInputVector, TreeParityMachine and DynamicArray classes are very similar to the original ones defined in the prototype. There are a few changes made to make them more secure but these can be considered unchanged from the original architecture.

Finally the definitions is here again it is blank this time as it was already discussed in figure 5.3. This time the peer sometimes uses the updatePeersChanged function in definitions when it disconnects and the SingleTpmNetworkHandler also uses definitions indirectly when adding the key to the keystore.

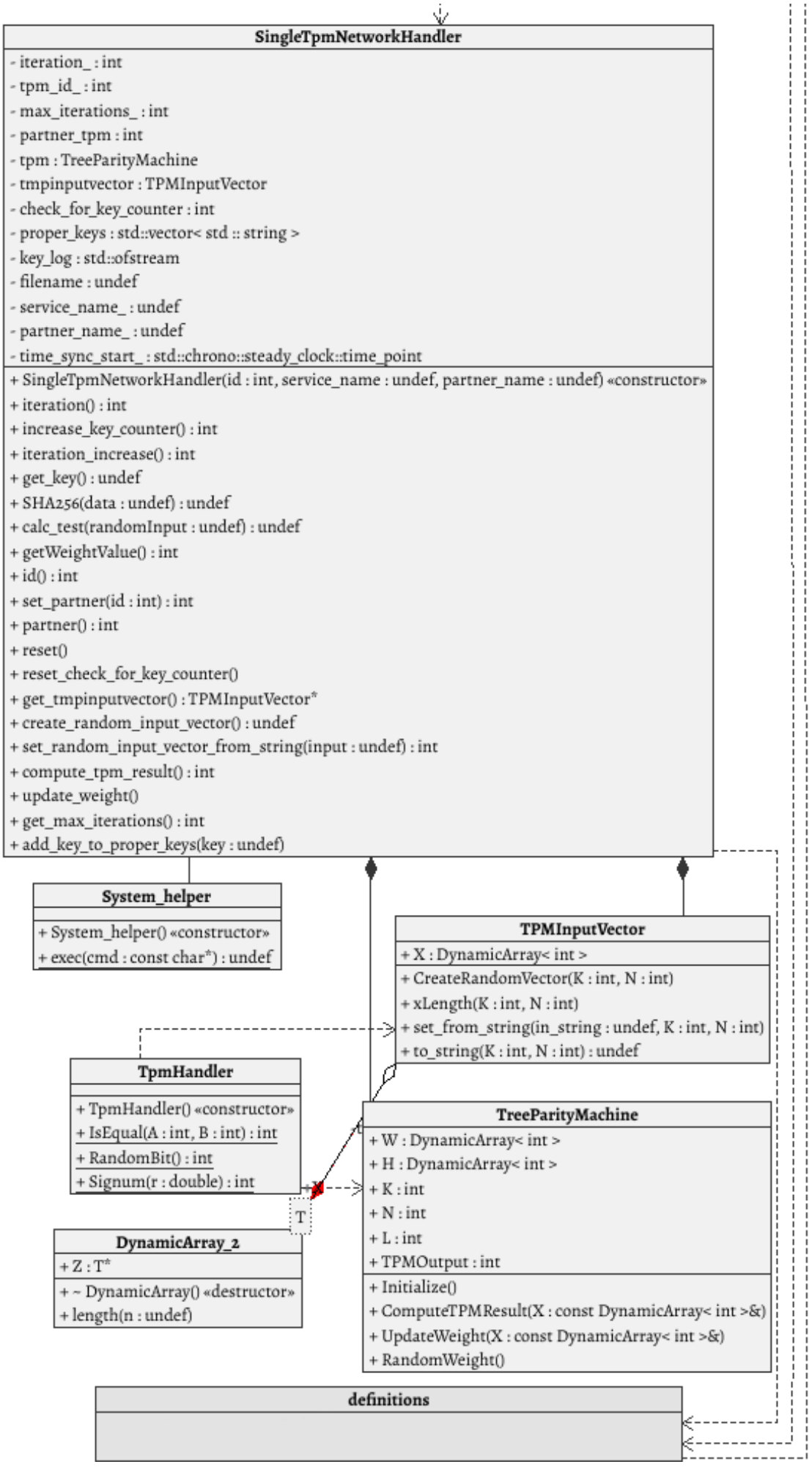

Figure 5.5: DynamiCrypt Class Diagram part 2

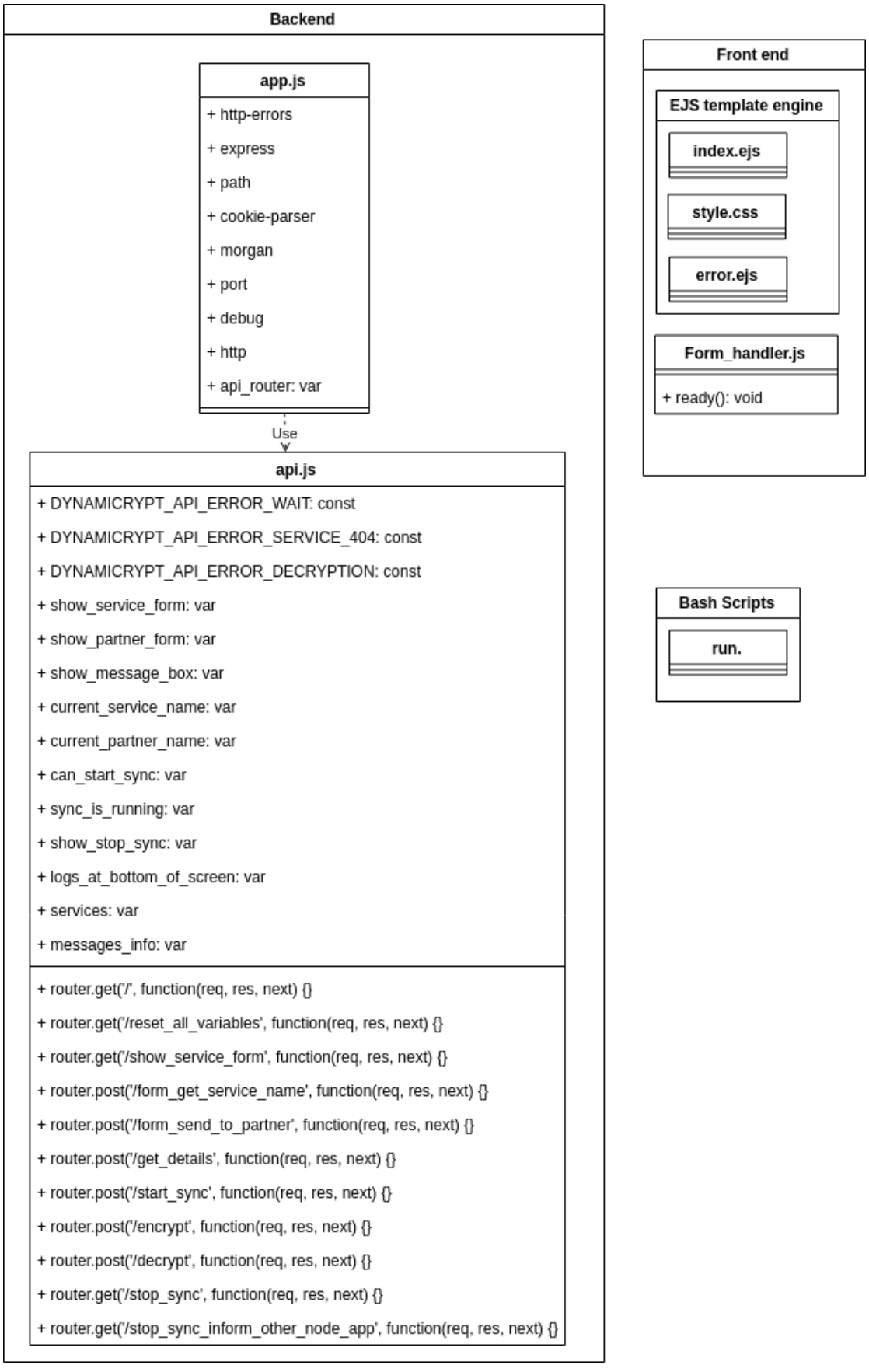

FIGURE 5.6: NodeJs App architecture

The Final change in the architecture is the NodeJS app represented in figure 5.6. Due to the nature of NodeJs in particular JavaScript you cannot really represent the language as a class diagram so I tried my best to represent the App as a whole and its major functions in a weird box diagram. Initially a NodeJs module would be made along side a NodeJS App however due to time constraints and a lot of time needed to be allocated for writing this thesis a decision was made to simple make a NodeJs app.

The NodeJs app simply exists to consume and demonstrate the use of the API and the different modes of encryption / decryption supported by the API currently (more will be added in the next version).

In order for other people to easily copy how the app interacts with the API every function that is related to the consumption of the API is located in the api.js file in the backend. For this reason no front end framework is used and the only interaction the backend has with the browser is through the forms submitted by the browser. It would be very easy to use something like ReactJS and provide a fluid front end that interacts with the DynamiCrypt API directly for some of the API calls however it would not be easy to copy and will confuse people trying to use DynamiCrypt API with their own custom Apps. In the future a better app demonstrating the API will be made that can be viewed as more of a real life app than something that demonstrates a technology will be made. But for now since only one app exists it is better to make it as clear as possible for people.

For parsing HTML EJS template engine is used which basically renders various forms depending on which Boolean variables are set, thereby making this NodeApp single paged. This also makes it easier for people to understand since all the routes are based on the root directory. and redirect to the root directory after each call so you will always see "/" in the browser. Since it a single page app only the index.ejs was required, by default Webstorm also creates error.ejs however I have not touched this file. Finally there is a little bit of JavaScript using jquery however this is purely for visuals such as scrolling divs to the end and refreshing the page to make it seem more interactive, obviously if a front end framework were to be used this would not be required.

Lastly a bash script called run can be seen in the diagram this is just made so that it is easy to execute the NodeJs app on whichever port you want for example ./run 4000 will use port 4000 for the app, this is because to easily demonstrate the DynamiCrypt system it is better to have NodeJs apps running on multiple ports.

Use Cases / Objectives The use cases or objectives are exactly the same as defined before apart from point number 7: "Provide an optional feature where extremely sensitive information like passwords should be split into pieces and sent over the network

with each piece encrypted with a different key." Which is not implemented in the current version of DynamiCrypt. Although it would be easy to fake implement this feature simply using multiple calls to the encrypt route from within the NodeJs app however performance would be better if it were to be done through the API natively. With DynamiCrypt there will be multiple encryption options available depending if you want a faster throughput of encrypted data or security. At the time of writing this there are two encryption modes available. There are a total of four modes planned for the next version.

1: fast, this is when data is received through the API the key that will be used as encryption would be the latest key available. This means that if you send four pieces of data three of those could be encrypted with the same key and by the time the fourth data piece is received, a new key might be generated and therefore it will be used to encrypt the last piece of data.

2: security, this time a key will only be used once to encrypt a piece of data. With this mode you are guaranteed that each time you request encryption from the API it will always use a unique key, this way if you send identical plain texts over and over they will produce different cipher texts. How this works in a basic high level description is the system searches the key store for keys that have never been used. (more on this later in the code explanation).

Risk Assessment No changes needed to be made to the risk assessment after the implementation stage.

Development Methodology The methodology remained the same as originally described where biweekly sprints took place and the content for the next sprint would be decided after completing the current one.

Implementation Schedule As discussed in the difficulties section, the implementation schedule is completely different than originally planned. This was expected due to the sprint methodology. The implementation schedule was filled in after each sprint and the final sprint plan can be seen in the table 5.1.

Evaluation The evaluation remains relative the same although there will be more focus on evaluating the tasks carried out in the sprint plan 5.1.

Prototype In the initial stage of the project a prototype was created to demonstrate synchronisation between two tree parity machines in a very basic manner. Some of the code used for the prototype still exists in the current version of DynamiCrypt as I am still using the simple tree parity machine synchronisation method. For the next version

of DynamiCrypt the synchronisation will use a slightly different formula therefore most of the code used for the prototype will be gone will be different.

Bellow are screen shots of the DynamiCrypt system in action and the NodeJs apps using it.

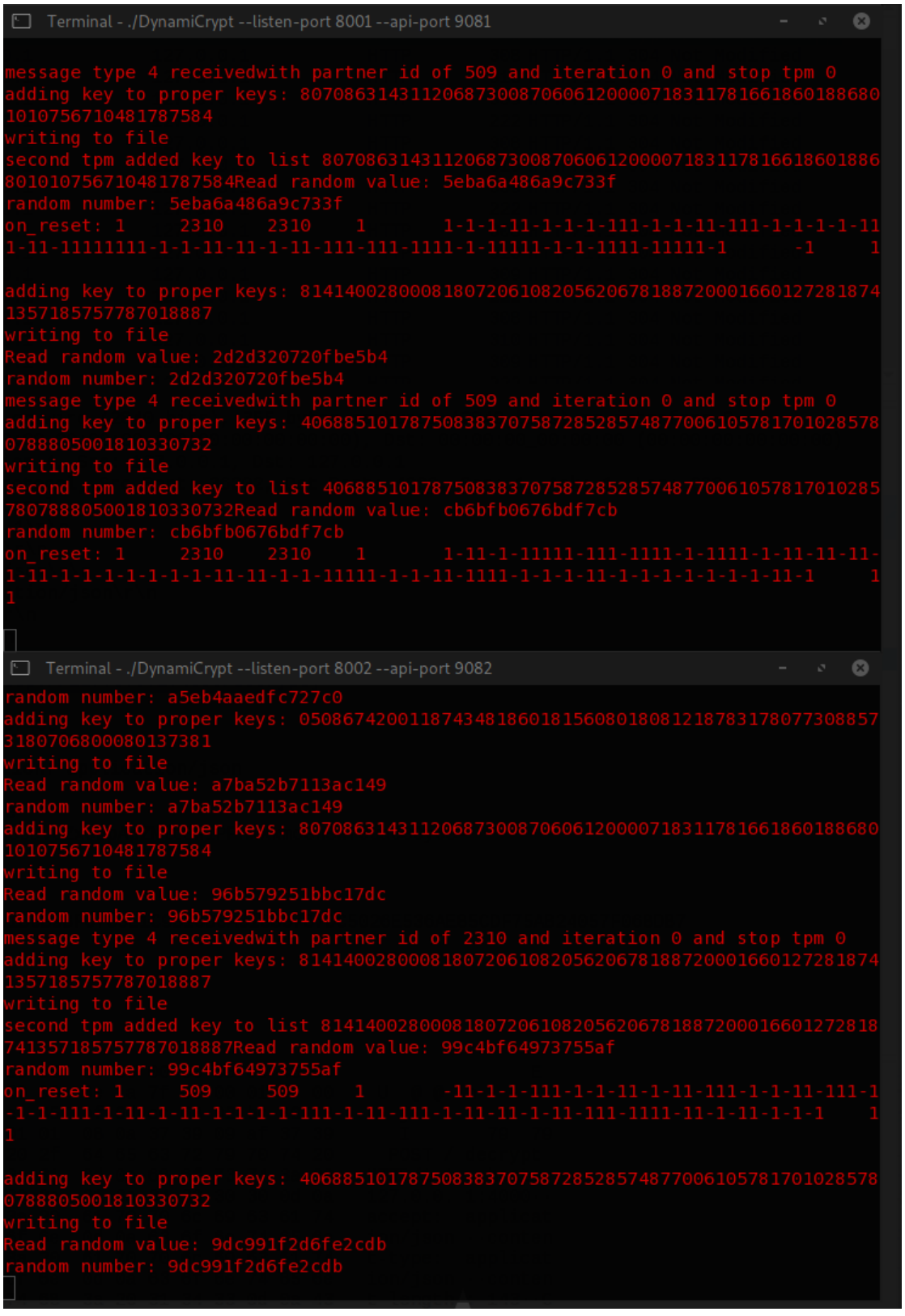

FIGURE 5.7: DynamiCrypt services in action

Figure 5.7 shows a snapshot of the two DynamiCrypt services in action (both API and sync-server share the same executable), mostly synchronising.

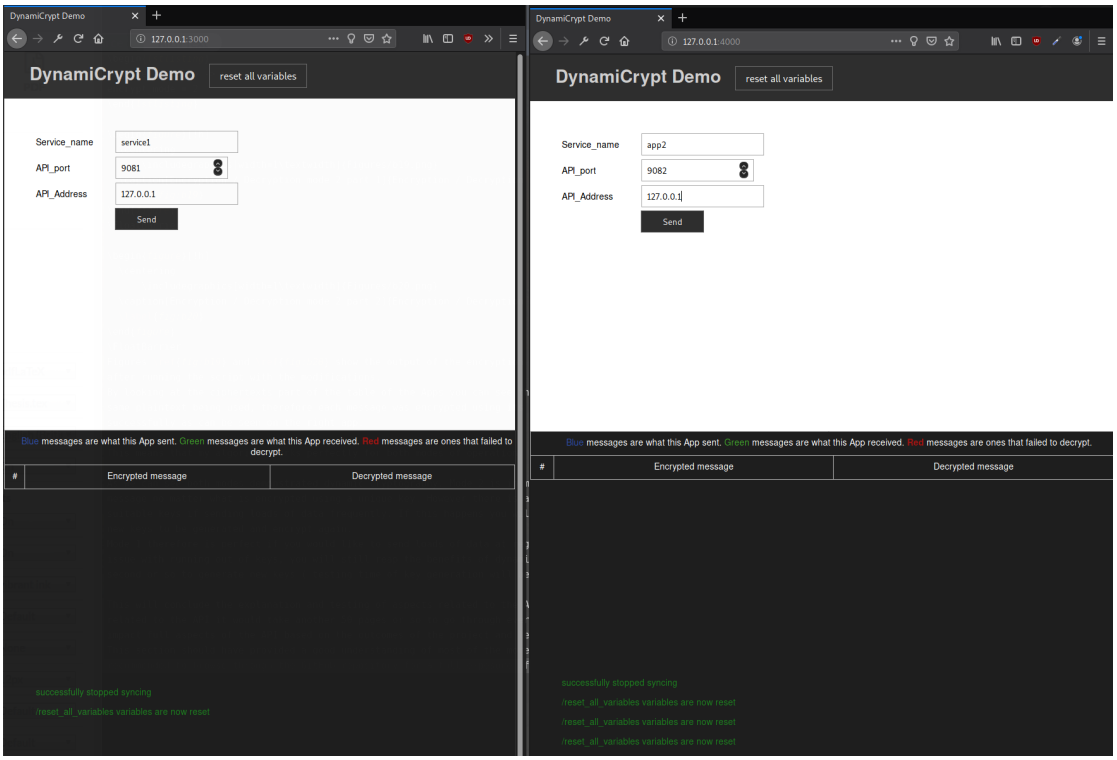

FIGURE 5.8: NodeJs Apps registering their names with the API

Figure 5.8 shows the two NodeJs apps about to register their names with the API.

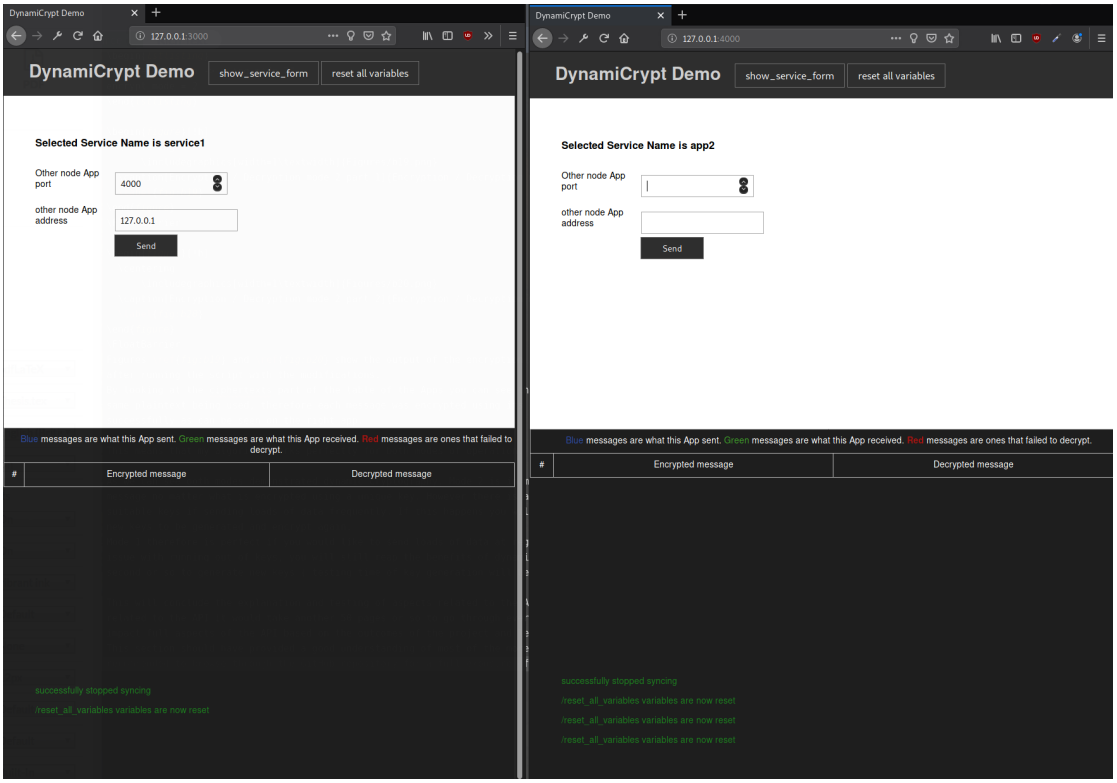

FIGURE 5.9: NodeJs Apps sharing details

Figure 5.9 shows the left NodeJs app about to connect to the right NodeJs app and share details.

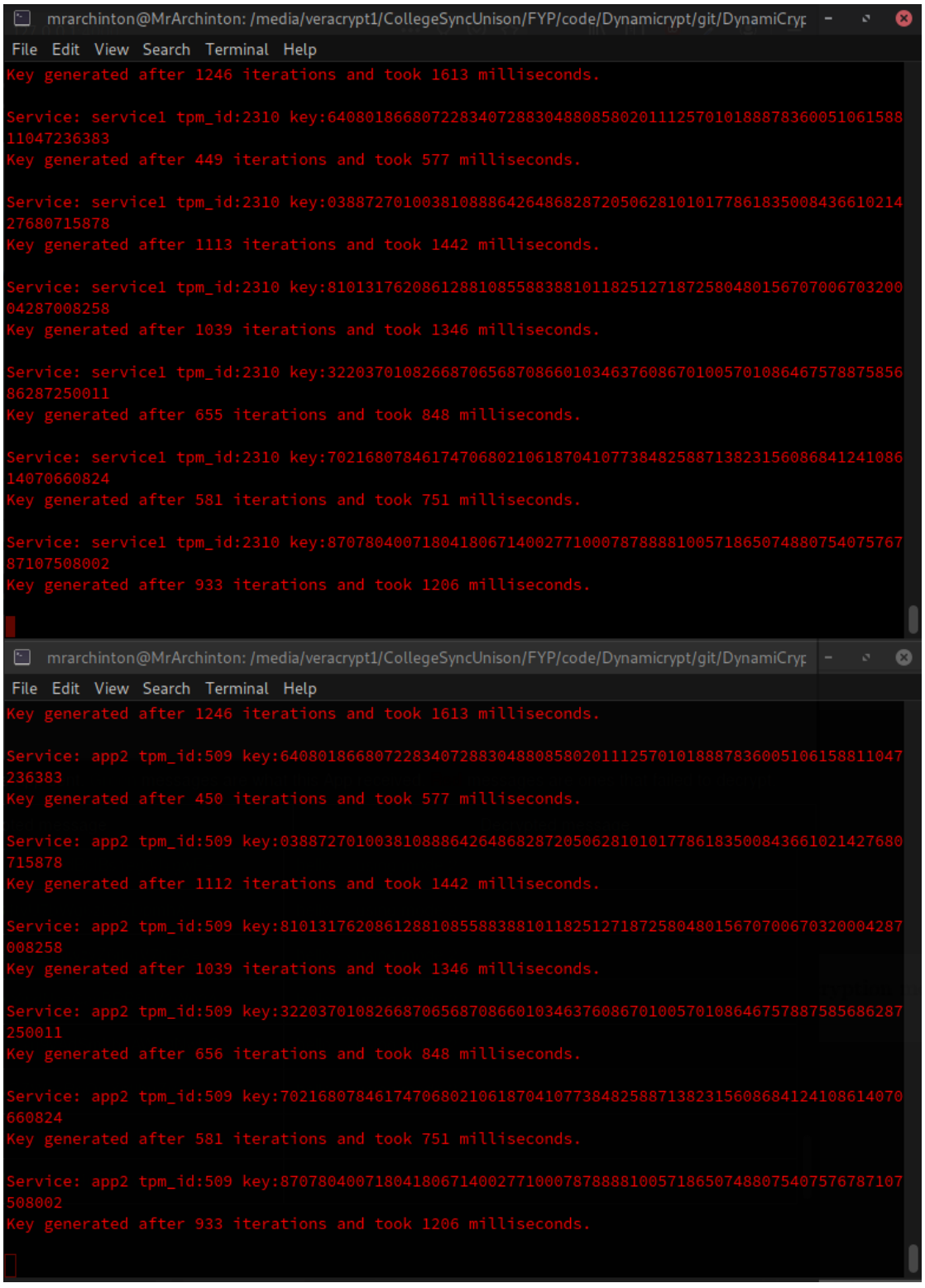

FIGURE 5.10: Tree Parity Machines Logs

Figure 5.10 shows a snapshot of the tree parity machines logs for a service called service1 and another service called app2. Both of these are synchronising with each other therefore they produce the same keys.

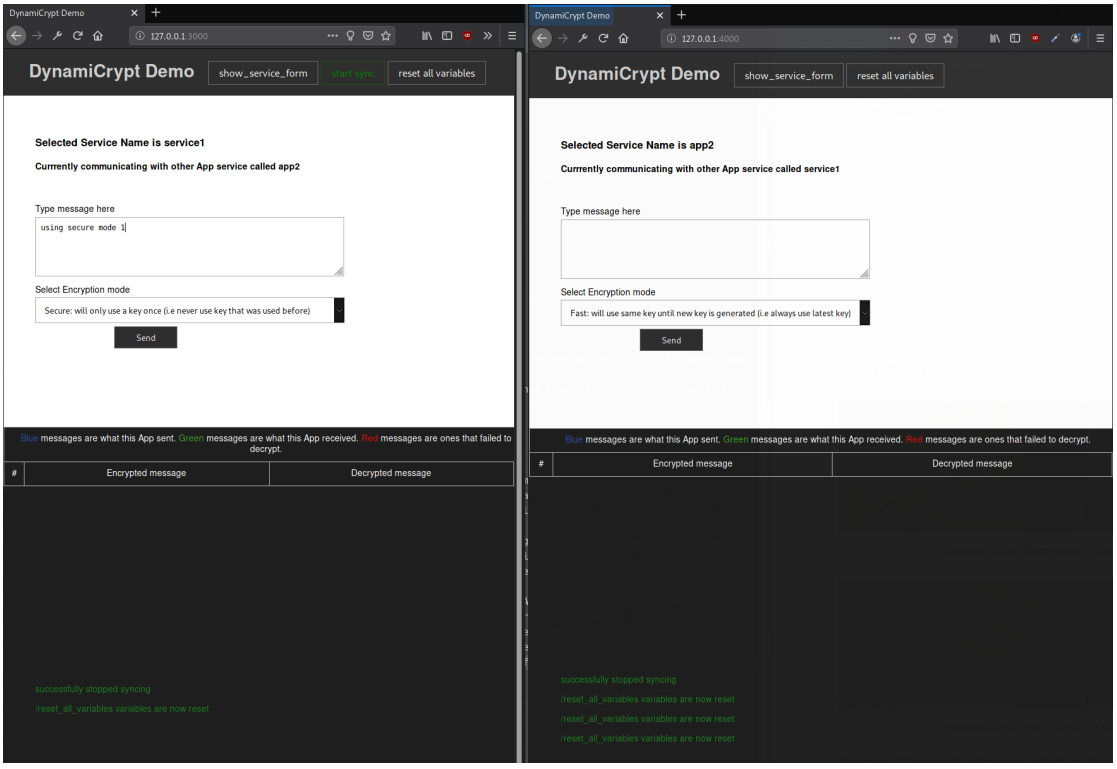

FIGURE 5.11: Encryption / Decryption mode 2 part 2

Figure 5.11 shows the left NodeJs app about to send a message to the API to be encrypted and forward the information received from the API to the right NodeJs app for decryption. The right NodeJs app will call the API to decrypt it.

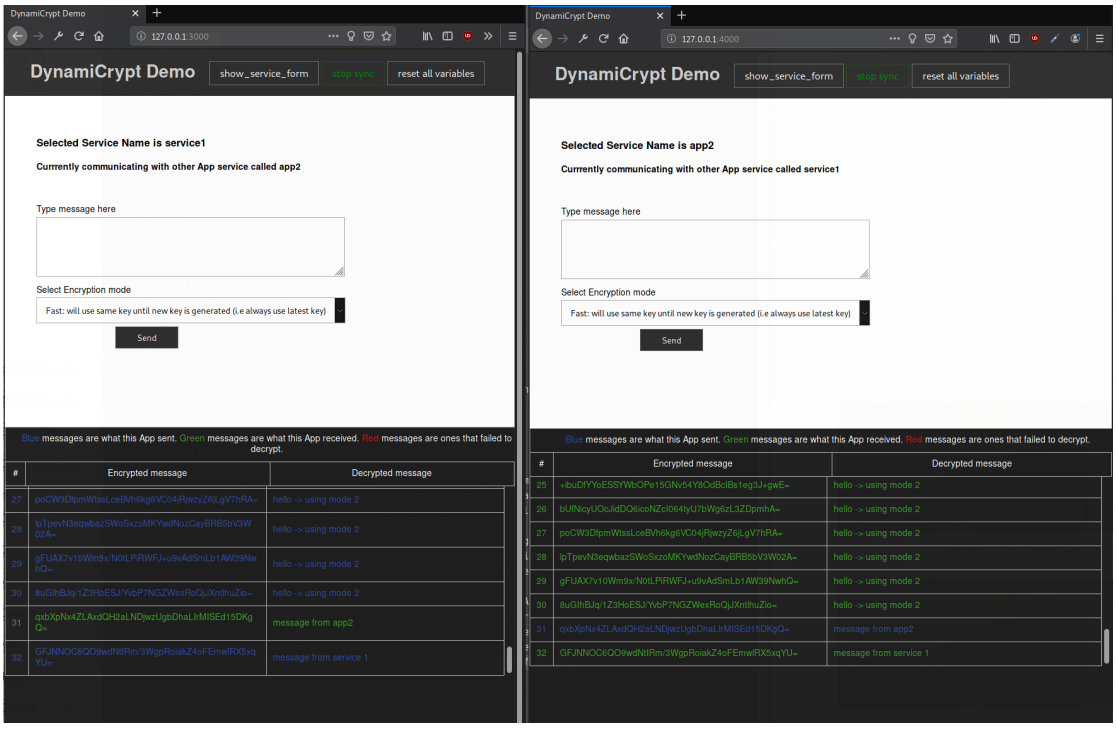

FIGURE 5.12: NodeJs apps after sending multiple encrypted messages to each other

Figure 5.12 shows the two NodeJs apps after sending multiple encrypted messages to each other.

Chapter 6

Testing and Evaluation

This chapter will go into more detail into how DynamiCrypt actually works. There will various code snippets present throughout this chapter. After an explanation of how parts of the system work tests will also be carried out.

6.1 The Beginning

C++ is a compiled language and DynamiCrypt requires quite a few libraries to be present for linking. The command to compile DynamiCrypt is the following.

```
g++ *.cpp -lboost_system -lpthread -lboost_thread -lboost_program_options -
    lcryptopp -lpistache -o DynamiCrypt
```

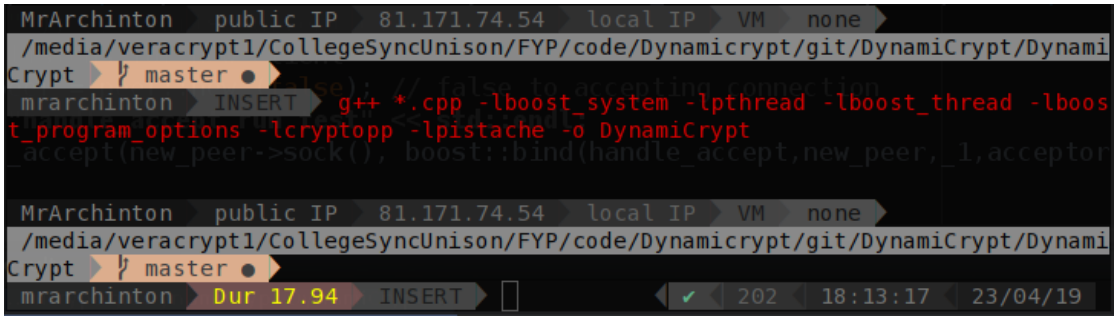

FIGURE 6.1: Compiling DynamiCrypt

Figure 6.1 shows the result of the compilation and as you can see there are no errors or warnings present, this is an indication that the code is syntax correct and operations are performed for the correct data types.

Every program starts of with a main function. DynamiCrypt's main function is rather light, all it does is processes the command line arguments, creates a listening peer and

creates the API server object. To create an initial peer the already seen before code is used.

```
ip::tcp::acceptor acceptor(service, ip::tcp::endpoint(ip::tcp::v4(),
    listen_port));
peer::ptr initial_peer = peer::new_(false);
acceptor.async_accept(initial_peer->sock(), boost::bind(handle_accept,
    initial_peer,_1, &acceptor));
```

Here an acceptor object is created this allows for the peer object to listen on a port on the localhost.

```
peer::new_(false);
```

Is used to notify the peer object that this peer will be waiting for a connection.

```
void handle_accept(peer::ptr peer, const boost::system::error_code & err,
    ip::tcp::acceptor* acceptor) {
    peer->start("","");  // starts current client
    // creates and listens for new client
    peer::ptr new_peer = peer::new_(false); // false to accepting
    connection
    //std::cout << "handle_accept run test" << std::endl;
    acceptor->async_accept(new_peer->sock(), boost::bind(handle_accept,
    new_peer,_1,acceptor)); // this
}
```

The above function is called when the asynchronous connect object receives an external tcp request. The start method is called on the current peer followed by a creation of a new peer which will start listening for the next connection. This way there is always only one peer waiting for new connections.

Because of how Boost manages asynchronous programming by using a service object to manage threads as well as using the proactor design pattern where asynchronous operations can occur using only one thread. However because DynamiCrypt is operation heavy when it comes to synchronisation I have used both threads and the proactor approach, this way when an asynchronous operation is about to take place Boost will choose a random available thread for it to run on. For this reason there are a few operations taking place in the main.cpp file to set all of this up.

```
boost::thread_group threads;

start_listen(4);

void start_listen(int thread_count) {
    for ( int i = 0; i < thread_count; ++i)
```

```
        threads.create_thread( listen_thread);
}

void listen_thread() {
    service.run();
}

}
```

A group of threads variable is allocated, then the start listen function is called in the main function, this function creates four threads in this case and executes the service.run() function in each thread. This just executes boost asynchronous handler on each of the four threads.

Finally to ensure a clean exit without leaving any zombie processes the main function will wait for all the threads to finish their execution.

```
threads.join_all();
```

The API server is a bit more straight forward since the threads and asynchronous operations are more hidden away from the user by the Pistache library.

```
api_service_data_handler.set_address_and_port_of_sync("127.0.0.1",
    listen_port);
APIServer api_server(api_port);
```

The first line here sets the address of the sync-server and the port on which it is listening on as this information will be shared with the NodeJs app later. The actual creation of the API is simply calling the constructor with a port number.

The API and the sync-server require each other for operation, however it would be confusing if both were explained at the same time therefore the API will be covered initially followed by the sync-server.

6.2 API

The constructor of the API server creates the variables needed for the server and ofloads it to the APIservice object.

```
APIServer::APIServer(int port_number) {
    Pistache::Port port(port_number);
    int thr = 2;
    Pistache::Address addr(Pistache::Ipv4::any(), port);
    //cout << "Cores = " << hardware_concurrency() << endl;
```

```
    std::cout << "API Using " << thr << " threads" << std::endl;
    API_service api(addr);
    api.init(thr);
    api.start();
    api.shutdown();
}
```

For the API two threads were defined as this is enough to handle multiple clients. The threads are created as follows.

```
void API_service::init(size_t thr = 2) {
    auto opts = Pistache::Http::Endpoint::options()
        .threads(thr)
        .flags(Pistache::Tcp::Options::InstallSignalHandler);
    httpEndpoint->init(opts);
    createDescription();
}
```

Pistache is a much more high level library than Boost therefore for the setup I mostly followed the examples they have in their GitHub repository.

```
void API_service::start() {
    router.initFromDescription(desc);
    httpEndpoint->setHandler(router.handler());
    httpEndpoint->serve();
}
```

start() sets up the description which in Pistache's case is the information about all the routes available, some licensing and API info.

Here is a small extract from the function that creates the description

```
void API_service::createDescription() {
    desc
        .info()
        .license("Apache", "http://www.apache.org/licenses/LICENSE-2.0");

    auto backendErrorResponse =
        desc.response(Pistache::Http::Code::Internal_Server_Error, "An
    error occured with the backend");

    desc
        .schemes(Pistache::Rest::Scheme::Http)
        .basePath("/v1")
        .produces(MIME(Application, Json))
        .consumes(MIME(Application, Json));
```

```
auto versionPath = desc.path("/v1");

auto path = versionPath.path("/options");

path
    .route(desc.get("/test-ok"))
    .bind(&API_service::route_test, this)
    .produces(MIME(Application, Json), MIME(Application, Xml))
    .response(Pistache::Http::Code::Ok, "ok");

path
    .route(desc.post("/init"), "Initiate Communication")
    .bind(&API_service::initial, this)
    .produces(MIME(Application, Json))
    .consumes(MIME(Application, Json))
    .response(Pistache::Http::Code::Ok, "Initial request")
    .response(backendErrorResponse);
```

The path is built by firstly specifying the version of the API this way it is possible to support legacy code using you old API. Next I decided to put all the DynamiCrypt operations preceded by /options this way in the future if the API can be expanded to do other things like maybe formatting / encoding data. Next is the actual routes used by the DynamiCrypt API. The first one is /test-ok a GET route, this is handy if you want to check if the API is up and running. To access this route you would send a get request to this for example 127.0.0.1:9081/v1/options/test-ok/. This simple route will just return "ok" as can be seen when a curl get request is being made as can be seen in figure 6.2.

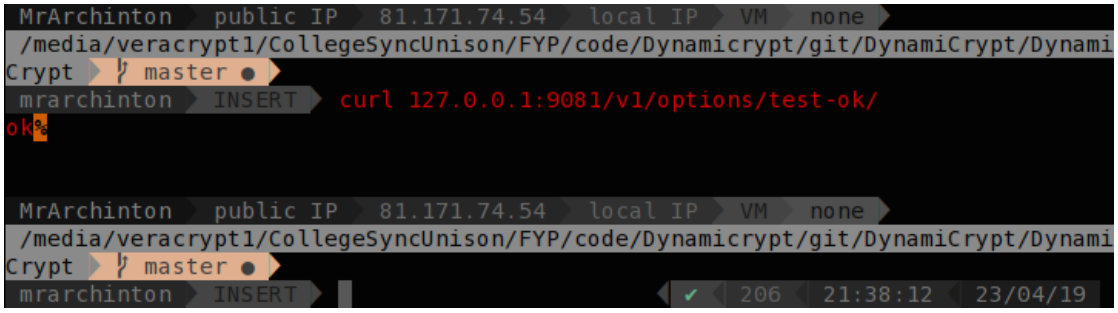

FIGURE 6.2: 127.0.0.1:9081/v1/options/test-ok/

The list of all routes currently available in the API and a brief description are as follows.

```
/v1/options/test-ok          GET
// test if API is up
/v1/options/init             POST
// Initial request
/v1/options/init_config      POST
// Initial Config
```

```
/v1/options/sync              POST
// Begin Sync
/v1/options/status            POST
// check if connected to tpm ok
/v1/options/encrypt           POST
// encrypt / decrypt data
/v1/options/exit              POST
// delete tpm and data associated with the app using the API
/v1/options/:rest             POST
// custom 404
```

Every post route is handled by a specific function.

```
path
        .route(desc.post("/init"), "Initiate Communication")
        .bind(&API_service::initial, this)
        .produces(MIME(Application, Json))
        .consumes(MIME(Application, Json))
        .response(Pistache::Http::Code::Ok, "Initial request")
        .response(backendErrorResponse);
```

In this case every time the init route is called the initial() function handles it and produces its own response. Here is the initial function bellow.

```
void API_service::initial(const Pistache::Rest::Request& request, Pistache
    ::Http::ResponseWriter response) {
    rapidjson::Document document;
    // make into json object
    char * jsonBody = new char [request.body().length()+1];
    strcpy (jsonBody, request.body().c_str());
    document.Parse(jsonBody);

    std::string service_name;
    int data_ok = 1;

    if(document.HasMember("service_name")){
        if(document["service_name"].IsString()){
            service_name = document["service_name"].GetString();
        }
        else{
            data_ok = 0;
        }
    }
    else{
        data_ok = 0;
    }
```

```
    std::string respond_service_name;
    rapidjson::StringBuffer buffera;
    rapidjson::Writer<rapidjson::StringBuffer> writera(buffera);

    if(data_ok){
        respond_service_name = api_service_data_handler.new_service(
service_name);
        writera.StartObject();
        writera.Key("service_name");
        writera.String(respond_service_name.c_str(), respond_service_name.
length());
        writera.Key("address_of_this_tpm");
        writera.String(api_service_data_handler.get_sync_address().c_str(),
 api_service_data_handler.get_sync_address().length());
        writera.Key("port_of_this_tpm");
        writera.Uint(api_service_data_handler.get_sync_port());
        writera.EndObject();

    }

    else{//error with request
        writera.StartObject();
        writera.Key("error");
        writera.String("invalid request");
        writera.EndObject();
    }

    response.send(Pistache::Http::Code::Ok, buffera.GetString());
}
```

Similar to NodeJs each of these handlers have a reference to a request and response object. For extracting JSON data the RapidJson library is used, it is also used for creating JSON data. The API interacts with the

```
api_service_data_handler
```

object for managing the services.

It will take too long to go through all of the routes and how they function so a more basic explanation will suffice. I would recommend watching my demo video here https://www.youtube.com/watch?v=LsR4XsGrDCYfeature=youtu.be on YouTube as it would be easier to understand.

Initially the NodeJs apps must register with the API this unfortunately turned out to be a multi step process however it is necessary since the API needs data from both of the Apps that wish to use DynamiCrypt.

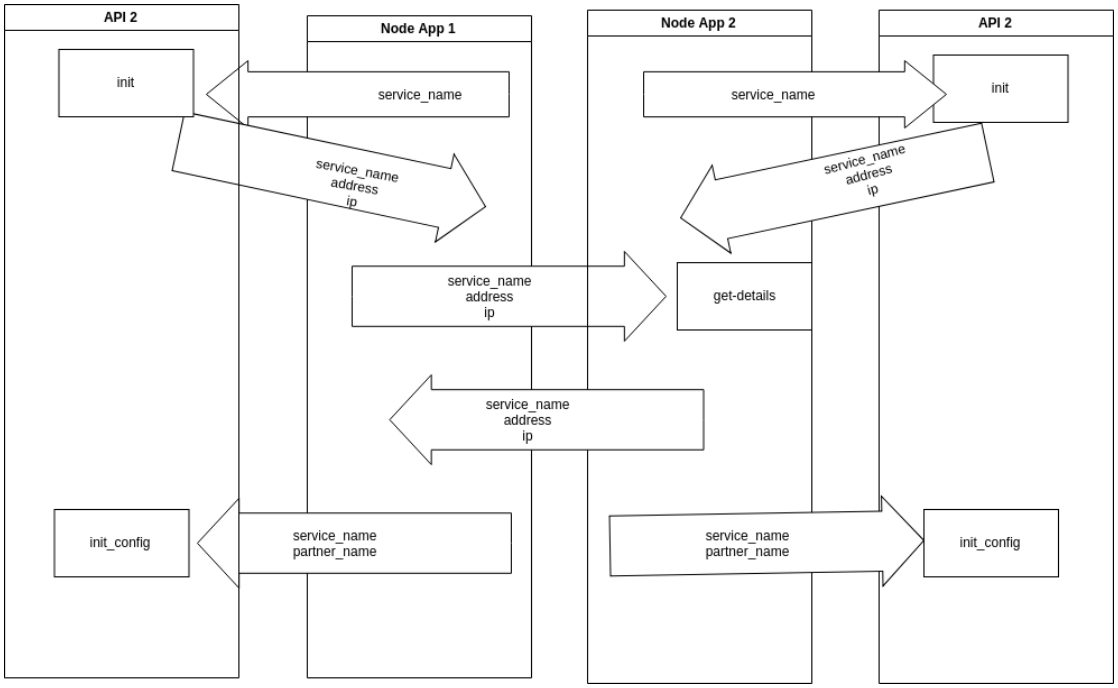

FIGURE 6.3: How Node Apps register with the API

Figure 6.3 is a call flow diagram of how the NodeJS apps register themselves with the API. The same can be demonstrated by using Wireshark and listening on localhost. The filters I used are (tcp.port == 9081) http since I only want to see POST requests for the API running on port 9081 since the POST requests for the other API are identical so there is no point in showing it twice.

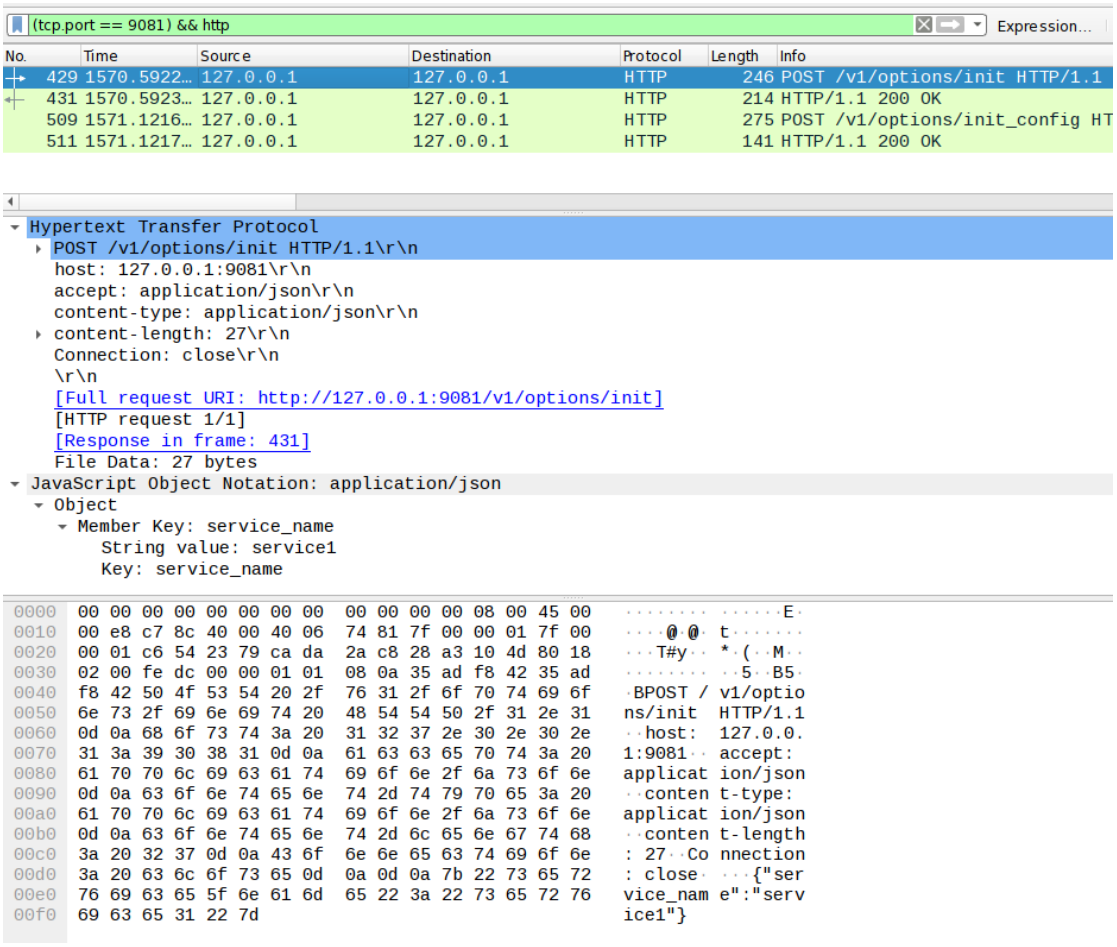

FIGURE 6.4: POST request to init

Figure 6.4 shows how the NodeJs App sent a POST request to the init route with its user service name.

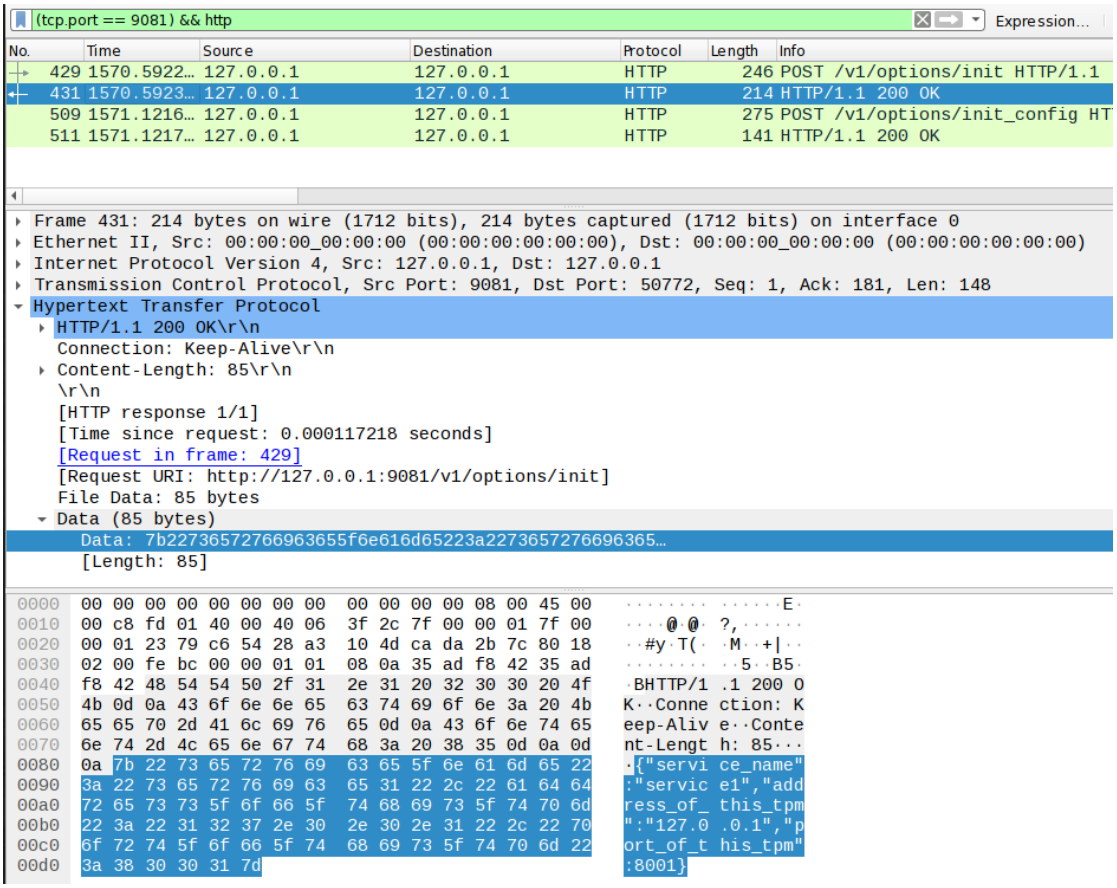

FIGURE 6.5: API response

Figure 6.5 shows how the APIs reply to the previous post request with the service name the API wants the NodeJS App to use, the port of the sync-server and the address of the sync-server. After this the Node App would make a request to the other Node App for the get details route to exchange information. This can also be acquired with Wireshark by changing a filter as seen in figure 6.6.

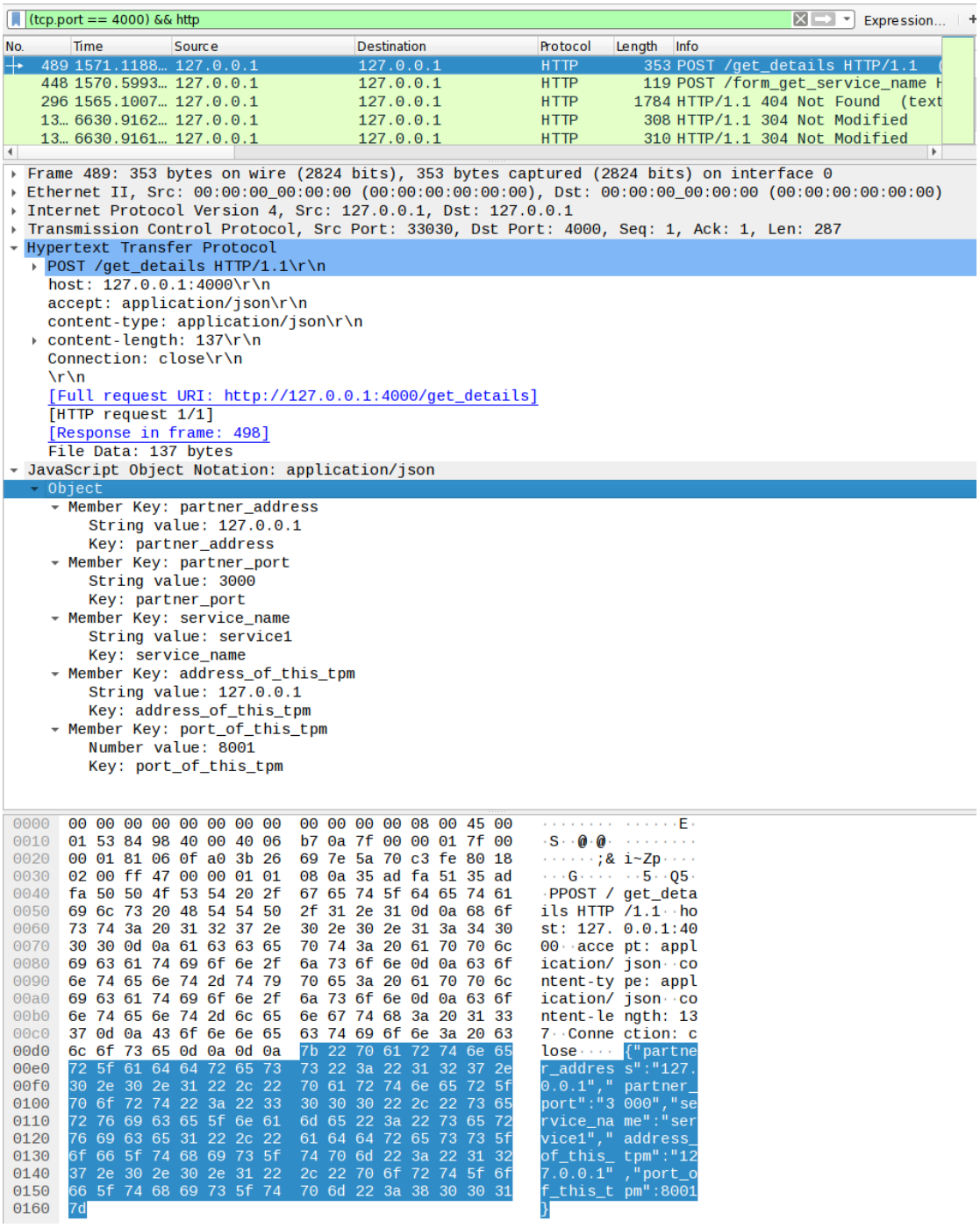

FIGURE 6.6: POST request to other NodeJs app's get details route

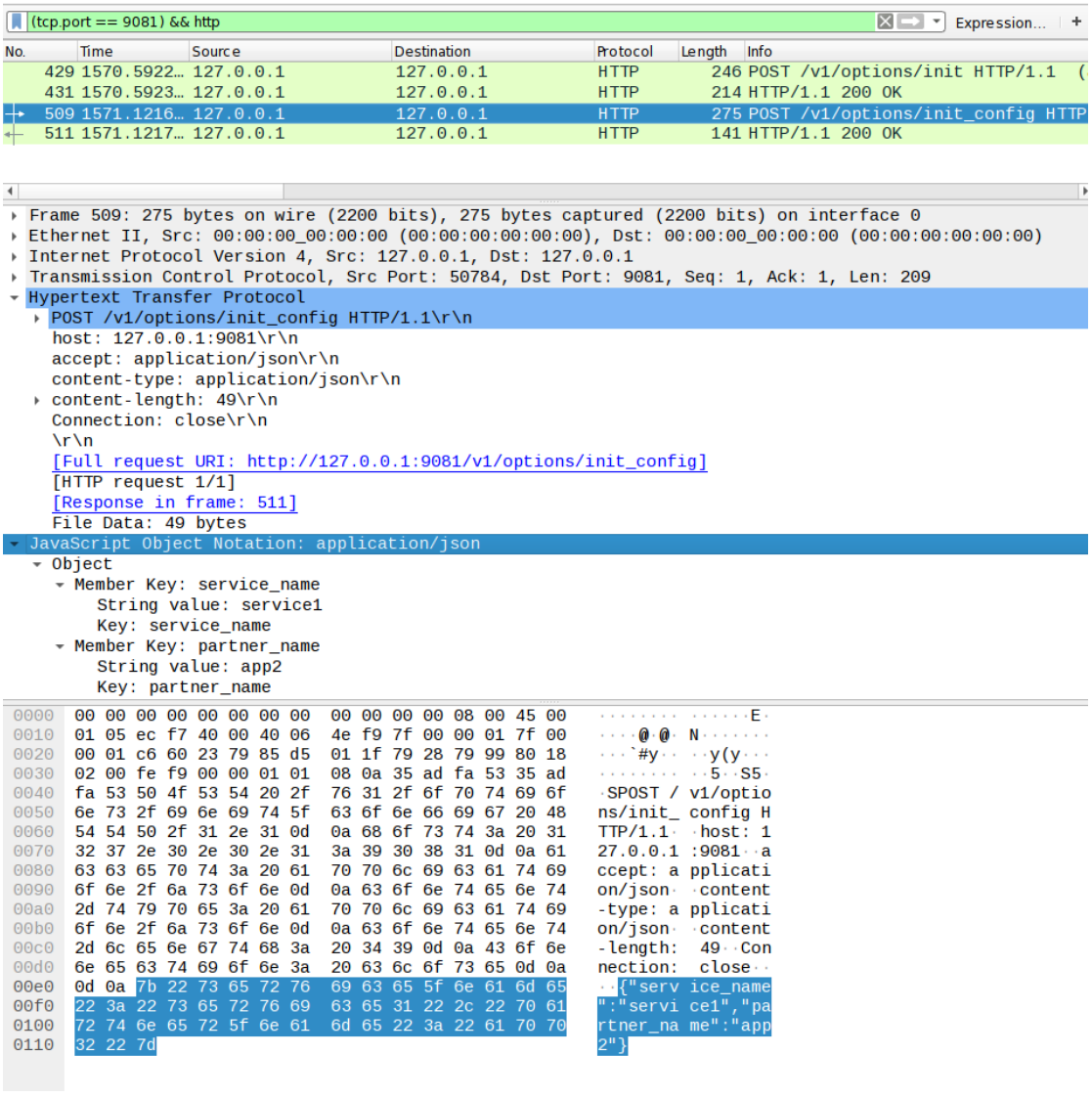

FIGURE 6.7: POST request to init config

Figure 6.7 shows how the NodeJs app updates the API with the service name of the other NodeJs app in this case it is called partner name.

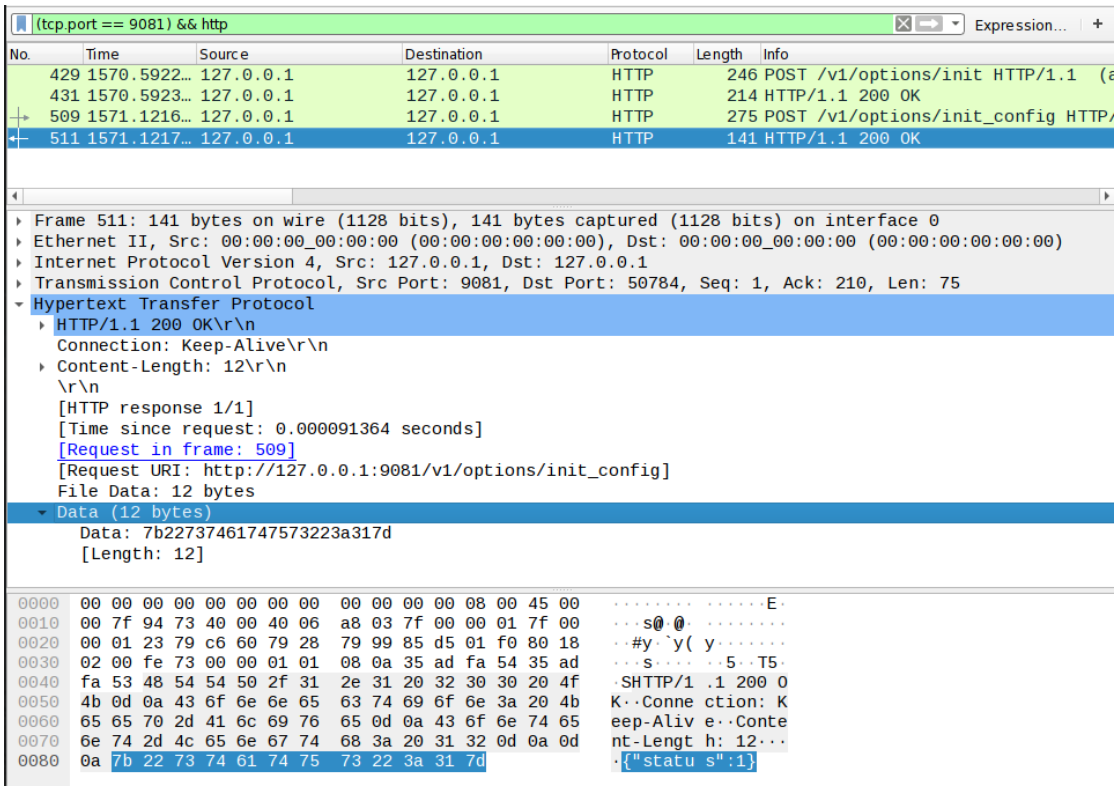

FIGURE 6.8: API response

Figure 6.8 shows the APIs reply to the previous post request with a status 1 which means everything is OK and the update was successfully made. Now the two NodeJs apps are fully registered with the API. The next step is to tell the API to tell the sync-server to start synchronising so that the Apps can send messages to each other using dynamic encryption. For this to occur one of the Apps must simply call the sync route of the API.

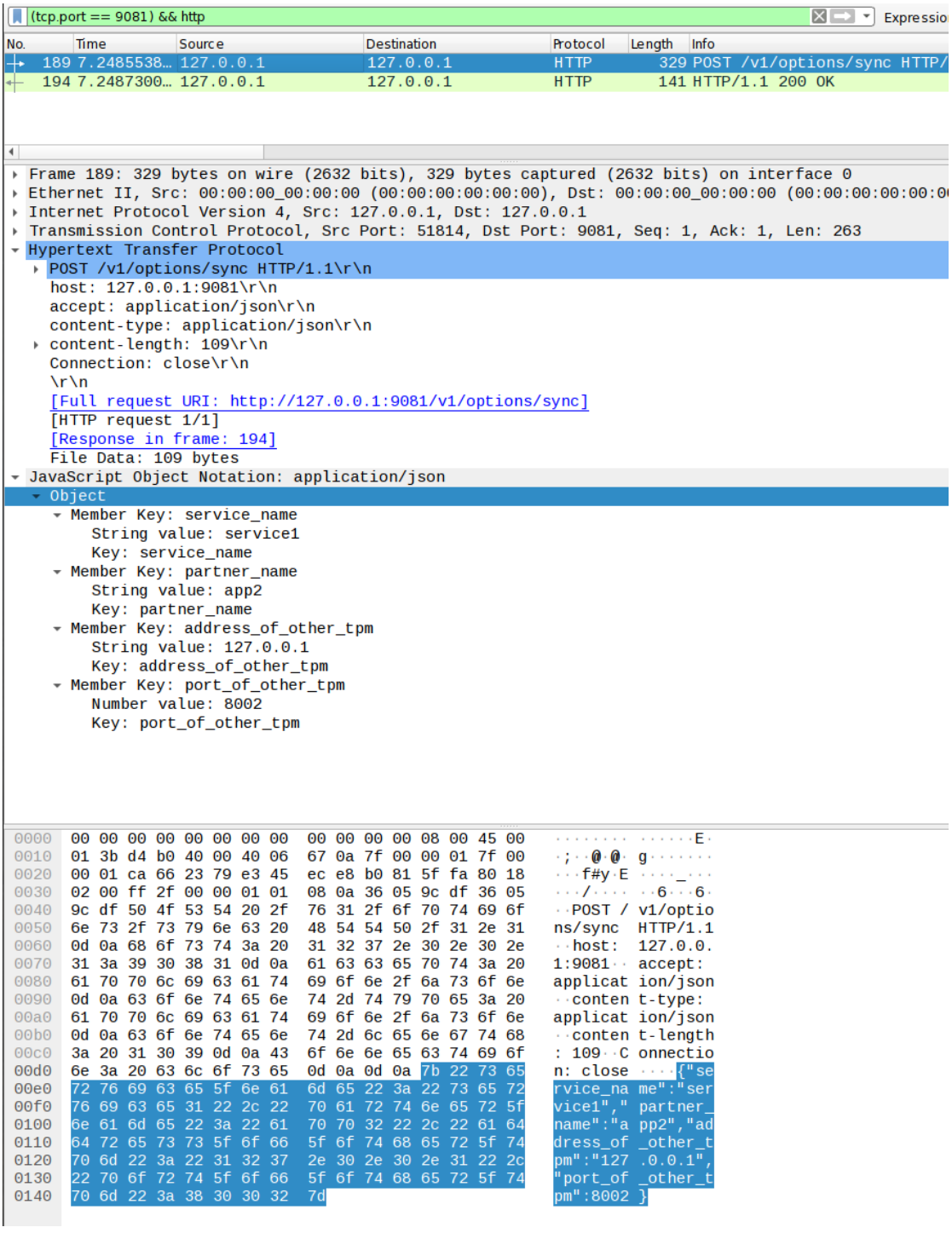

FIGURE 6.9: POST request to sync

Figure 6.9 shows the sync route of the API is called. The Node App sends a lot of details this time including service name, partner name, the address and port of the sync-server that is used by the other API that the other NodeJs App communicates with. This is because it is calling a different function outside of the api service data handler object. This time a function from the definitions.cpp is called.

```
int begin_sync(std::string address, int port, std::string service_name, std
    ::string partner_name){
    try{
        peer::ptr initiating_peer = peer::new_(true, address, port);
        initiating_peer->start(service_name, partner_name);
    }
    catch(std::exception& e){
        return 0;
    }
    return 1;
}
```

Here a new peer is created of connecting type instead of listening type like we looked at last time. This peer will try to connect to the sync-server at the address and port the Node App sent it.

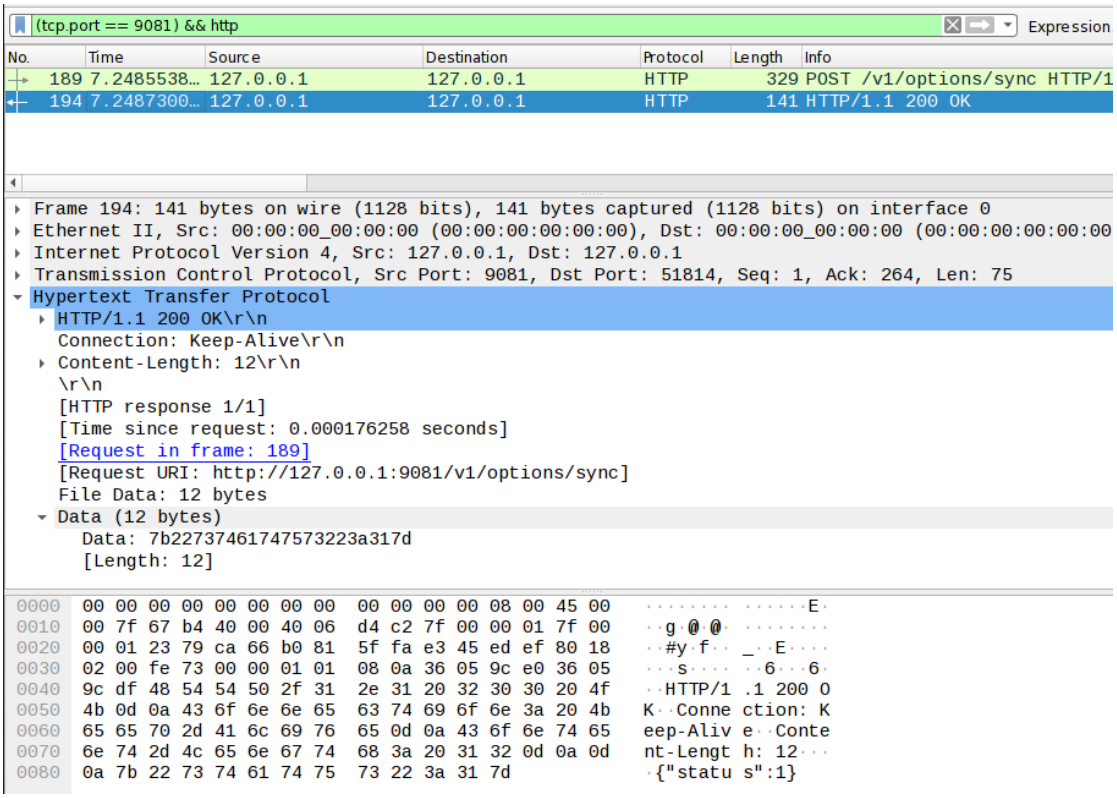

FIGURE 6.10: API response

Figure 6.10 shows the APIs reply to the previous post request with a status 1 which means the request was processed successfully however this does not mean that the sync-server connected to the other sync-server correctly this is because of the asynchronous nature of the sync-server it is impossible to get the result of the connection. Therefore the NodeJs app should query the API to see if the connection was successful and if the two sync-servers are currently syncing this is done by calling the status route in the API

and providing the service name. if the status returned is 1 then the two sync-servers are currently synchronising, if the status is 0 then they failed to connect or some other error occurred.

Encryption / Decryption

Now the NodeJs apps can send messages to the API to be encrypted. How encryption works is it selects the appropriate key from the key store based on the mode provided. The key store looks like this.

```
struct key_store {
    std::string key;
    int uses;
};

struct API_data {
    std::string service_name_;
    std::string service_name_partner_;
    std::vector<key_store> keys_;
};
```

This is an inner struct inside the

```
API_service_data_handler
```

class which contains a vector of API data objects.

Currently DynamiCrypt supports two encryption methods. Encryption mode 1 will simply use the latest key avaiable in the keystore, this mode is handy if you want the fastest mode of encryption available. For encryption mode 1,

```
if(mode == 1){
                //check for any latest key
                if(api_data->keys_.size() == 0){
                    return DYNAMICRYPT_API_WAIT;
                }else{
                    string_key = api_data->keys_.back().key;
                    api_data->keys_.back().uses ++;
                    if(PRINT_API_CRYPT_MESSAGES){
                        std::cout << "got key like this " << string_key <<
    std::endl;
                        std::cout << "key was used " << api_data->keys_.
    back().uses << " times" << std::endl;
                    }
                }

            }
```

The appropriate api data object is selected referring to the service that called the API. The size of the key store is initially checked to see if it contains any keys, if not then the API will tell the Node App to wait. The node app can then query the API again and again for encryption or just use a timeout and wait for 100 milliseconds or so. If there is keys in the key store then the last key that was entered is used for encryption. The uses count of this particular key is also increased, the uses count doesn't really matter for mode 1 but will matter for mode 2 and future modes that are not implemented yet.

Encryption mode 2 would be the more secure one as only keys that have 0 uses are picked this way each message would be encrypted using its own unique key.

```
else if (mode == 2){
                if(api_data->keys_.size() == 0){
                    return DYNAMICRYPT_API_WAIT;
                }
                else{
                    int attempts_to_find_key = 0; //search
    max_attempts_to_decrypt times for the key since decrypt will only
    search for max_attempts_to_decrypt times too
                    int found_key = 0;
                    for(int number_of_keys = api_data->keys_.size()-1;
    number_of_keys >= 0; number_of_keys --){
                        if(attempts_to_find_key == max_attempts_to_decrypt)
    {
                            break;
                        }
                        attempts_to_find_key ++;
                        if(api_data->keys_.at(number_of_keys).uses == 0){
                            string_key = api_data->keys_.at(number_of_keys)
    .key;
                            api_data->keys_.at(number_of_keys).uses++;
                            if(PRINT_API_CRYPT_MESSAGES){
                                std::cout << "got key like this " <<
    string_key << std::endl;
                                std::cout << "key was used " << api_data->
    keys_.at(number_of_keys).uses << " times" << std::endl;
                            }
                            found_key = 1;
                            break;
                        }
                    }
                    if(!found_key){
                        return DYNAMICRYPT_API_WAIT;
                    }
                }
            }
```

This mode is a little bit more involved then the simple get latest key. The Basic premise here is to find a suitable key that has 0 uses. This key is found by working back from the latest key down the vector until either a key is found, either the number of keys runs out or the search limit is reached. in this case the search limit is defined at 10 and stored at this variable.

```
max_attempts_to_decrypt
```

This means that in order to find a key provided the key store is longer than 10 the search will max out at 10 elements from the latest. This limit is set not so much for encryption but rather the decryption, this is because if there was no limit the encryption algorithm might choose a key that is say 200 keys away from the latest, the decryption algorithm will then have a heck of a time trying to find that correct key to use. And just like before if an appropriate key is not found the API will tell the Node App to wait.

After these two algorithms find the appropriate key the next step is to encrypt the data.

```
CryptoPP::byte key[ CryptoPP::AES::MAX_KEYLENGTH ];
gen_key(string_key,key);
std::string for_encode = encrypt(message, key, iv);
```

Earlier I said that the key generated by the tree parity machines will be used to encrypt the message. This is technically not true since that key is actually used to generate the actual key used for encryption. This is because the key generated by the tree parity machines is too long for AES-256 encryption therefore the gen key function generates an appropriate size key without loosing any data, this means that it does not simply cut of the rest of the key that's longer than 32 bytes needed for AES-256. Here is a snippet from that function.

```
void API_service_data_handler::gen_key(std::string string_key, CryptoPP::
    byte* key){
    if(string_key.length() > CryptoPP::AES::MAX_KEYLENGTH){
            int count_first = 0;
            int count_last = string_key.length() -1;
            int number_of_operations = count_last - CryptoPP::AES::
    MAX_KEYLENGTH;
            for(int i = 0; i< number_of_operations; i++){
                string_key[count_first] = string_key.at(count_first) +
    string_key.at(count_last);
                if(count_first == CryptoPP::AES::MAX_KEYLENGTH){
                    count_first = 0;
                }
                count_first ++;
                count_last --;
            }
```

```
        }

        int string_count = 0;
        for(int i=0; i<CryptoPP::AES::MAX_KEYLENGTH; i++){
            if(string_count == string_key.length() -1){
                string_count = 0;
            }
            key[i] = string_key.at(string_count);
            string_count ++;
        }
```

After encrypting the data, the ciphertext is then encoded with base 64 encoding. Encoding is necessary because random bytes tend to mess up the structure of the JSON object.

```
std::string encoded = encode_base64(for_encode);
output = encoded;
```

Finally when sending the ciphertext to the node app the API also generates a hash specifically SHA 256 of the original plaintext. this will help the API to determine if the message was decrypted successfully by comparing hashes. Hashes are one way function so it is save to create a hash of the plaintext. The node app would then essentially just forward the ciphertext and the hash to the other node app for decrypting.

Decryption has algorithms specific for each mode therefore decryption mode 1 will directly be capable of decrypting encryption mode 1 and so on. The decryption is a potentially much more operation heavy operation that encryption. This is because it takes a guess at the key then tries to decrypt it and if that fails takes a guess at another key. This might seen pretty bad but from testing normally it only takes one to three decryption attempts to decrypt the key since the most likely key would be the latest one.

The decryption section of the code initially decodes the data from base 64 then that data is passed to the various decryption algorithms based on the mode.

For decryption mode 1

```
if(mode == 1){
                if(api_data->keys_.size() == 0){
                    return DYNAMICRYPT_API_WAIT; // no keys in key ring
    shouldn't happen with decrypt if used properly
                }
                int number_of_keys = api_data->keys_.size()-1;  // maybe
    change to keys_.size()-1
```

```
            for(int decrypt_loop = 0; decrypt_loop<
max_attempts_to_decrypt; decrypt_loop++){
                //try last key first
                std::string string_key = api_data->keys_.at(
number_of_keys).key;
                if(PRINT_API_CRYPT_MESSAGES){
                    std::cout << "trying to decrypt with key " <<
string_key << std::endl;
                    std::cout << "key was used " << api_data->keys_.at(
number_of_keys).uses << " times" << std::endl;
                }
                CryptoPP::byte key[ CryptoPP::AES::MAX_KEYLENGTH ];
                gen_key(string_key,key);

                output = decrypt(decoded_message, key, iv);
                if(!hash_with_sha_256(output).compare(hash)){ //
decrypted successfully
                    api_data->keys_.at(number_of_keys).uses ++;
                    break;
                }

                if(number_of_keys == 0){
                    //output = "failed";
                    return DYNAMICRYPT_API_FAILED_DECRYPT;
                    break;
                }

                number_of_keys --;

            }
        }
```

This checks if there are keys in the key ring this is not necessary since the tree parity machines generate the same keys at pretty much the same time but it is just in there as a sanity check or to catch some other strange errors. This is because for decryption the key is presumed to exist since that key was used to encrypt in the first place.

Since this is mode 1 we can simply try the latest key first then work down from there. Again how much further down you can go depends on the size of the key store and the limitation put in place to prevent traversing all the keys in case of an error or broken data.

Next the message is decrypted and hashed and the hash is compared to the hash sent by the node app. if the hashes match then the plaintext is returned if not the next key in line is tested. If the hashes match the uses for the key is incremented this is important

because you don't want this API using the same key for encryption as the other API if using mode 2. In the unlikely event that all the keys tried failed to decrypt the message a failure message will be sent to the node App, during testing I never have seen this scenario occur however it is likely to happen if you delay sending the encrypted data for decryption by quite some time like ten or more seconds since the key store will be filled up with new keys and the old key required for decryption will be pushed back past the limit. For this reason it is important for the node apps to send the ciphertext to the other node app as soon as possible.

For decrypting mode 2 quite a long algorithm is used.

```
else if (mode == 2){ // try keys with 0 uses first
                std::string string_key;
                if(api_data->keys_.size() == 0){
                    return DYNAMICRYPT_APIWAIT; // no keys in key ring
    shouldn't happen with decrypt if used properly
                }
                int has_skipped_keys = 0;
                int number_of_keys = api_data->keys_.size()-1;  // maybe
    change to keys_.size()-1

                //std::cout << "number_of_keys at start " << number_of_keys
     << std::endl;
                std::vector<int> skipped_keys;
                for(int decrypt_loop = 0; decrypt_loop<
    max_attempts_to_decrypt; decrypt_loop++){
                    //std::cout << "decrypt_loop at start " << decrypt_loop
     << std::endl;
                    if(number_of_keys == -1){ // needed because of the
    continue which could cause number_of_keys to be -1
                        break;
                    }

                    if(api_data->keys_.at(number_of_keys).uses == 0){

                        string_key = api_data->keys_.at(number_of_keys).key
    ;
                        //std::cout << "key with 0 uses found " <<
    string_key << std::endl;
                    }else{
                        skipped_keys.push_back(number_of_keys);
                        //std::cout << "skipping key at index " <<
    number_of_keys << std::endl;
                        number_of_keys --;
                        has_skipped_keys = 1;

                        continue;
```

```
                }

                if(PRINT_API_CRYPT_MESSAGES){
                    std::cout << "trying to decrypt with key " <<
string_key << std::endl;
                    std::cout << "key was used " << api_data->keys_.at(
number_of_keys).uses << " times" << std::endl;
                }
                CryptoPP::byte key[ CryptoPP::AES::MAX_KEYLENGTH ];
                gen_key(string_key,key);

                output = decrypt(decoded_message, key, iv);
                if(!hash_with_sha_256(output).compare(hash)){ //
decrypted successfully
                    api_data->keys_.at(number_of_keys).uses ++;
                    return output;
                }

                if(number_of_keys == 0){
                    break;
                }

                number_of_keys --;

            }
            //code runs here only if decryption was unsuccessful
            if(has_skipped_keys){ //check for skipped keys
                for(int i = 0; i < skipped_keys.size(); i ++ ){
                    string_key = api_data->keys_.at(i).key;
                    if(PRINT_API_CRYPT_MESSAGES){
                        std::cout << "trying to decrypt with key " <<
string_key << std::endl;
                        std::cout << "key was used " << api_data->keys_
.at(i).uses << " times" << std::endl;
                    }
                    CryptoPP::byte key[ CryptoPP::AES::MAX_KEYLENGTH ];
                    gen_key(string_key,key);

                    output = decrypt(decoded_message, key, iv);
                    if(!hash_with_sha_256(output).compare(hash)){ //
decrypted successfully
                        api_data->keys_.at(i).uses ++;
                        return output;
                    }
                }
                // if key not decrypted then code is continued here
therefore
                //output = "failed";
```

```
            return DYNAMICRYPT_API_FAILED_DECRYPT;
        }else{
            //output = "failed";
            return DYNAMICRYPT_API_FAILED_DECRYPT;
        }

    }
```

The reason this is quite long is because of the introduction of skipped keys. Because for encryption mode 2 we only use a key with 0 uses. Therefore to speed up decryption we can test all the keys within the limit that have 0 uses since keys with 0 uses are most likely to decrypt the message for mode 2. If the keys with 0 uses fail to decrypt then the keys with more than 0 uses are that were skipped are tested, this should not happen but due to network latency it can be possible where both of the node apps send an encrypt request with mode 2 at the same time and the API manages to use the same key for both, this is very unlikely however just in case it does the code can take care of this scenario, and adding code here doesn't decrease the performance at all since it is just an if statement wouldn't even be called because of the break if the keys with 0 uses successfully decrypted the message.

Now that we have an idea of how the encryption / decryption operates its time to demonstrate it in action. I will use the quick setup python script to register the node apps quickly. The quick setup simply "fills out the forms" via code rather than hand. The script looks like this and is included in the Github repository.

```
#!/usr/bin/env python3.7

import requests
import time

Service_one_name = "service1"
Service_two_name = "app2"

Service_one_address = "127.0.0.1"
Service_one_port = 3000

Service_two_address = "127.0.0.1"
Service_two_port = 4000

Service_one_API_address = "127.0.0.1"
Service_two_API_address = "127.0.0.1"

Service_one_API_port = 9081
Service_two_API_port = 9082
```

```
url1 = "http://" + Service_one_address + ":" + str(Service_one_port) + "/
    form_get_service_name"
data = {'Service_name':Service_one_name, 'API_port': Service_one_API_port,
    'API_Address': Service_one_API_address}

r = requests.post(url = url1, data = data)


url1 = "http://" + Service_two_address + ":" + str(Service_two_port) + "/
    form_get_service_name"
data = {'Service_name':Service_two_name, 'API_port': Service_two_API_port,
    'API_Address': Service_two_API_address}

r = requests.post(url = url1, data = data)

time.sleep(0.5)

# now send info to parnter
url2 = "http://" + Service_one_address + ":" + str(Service_one_port) + "/
    form_send_to_partner"
data = {'port': Service_two_port, 'address': Service_two_address}
r = requests.post(url = url2, data = data)

# can now press sync in the browser
```

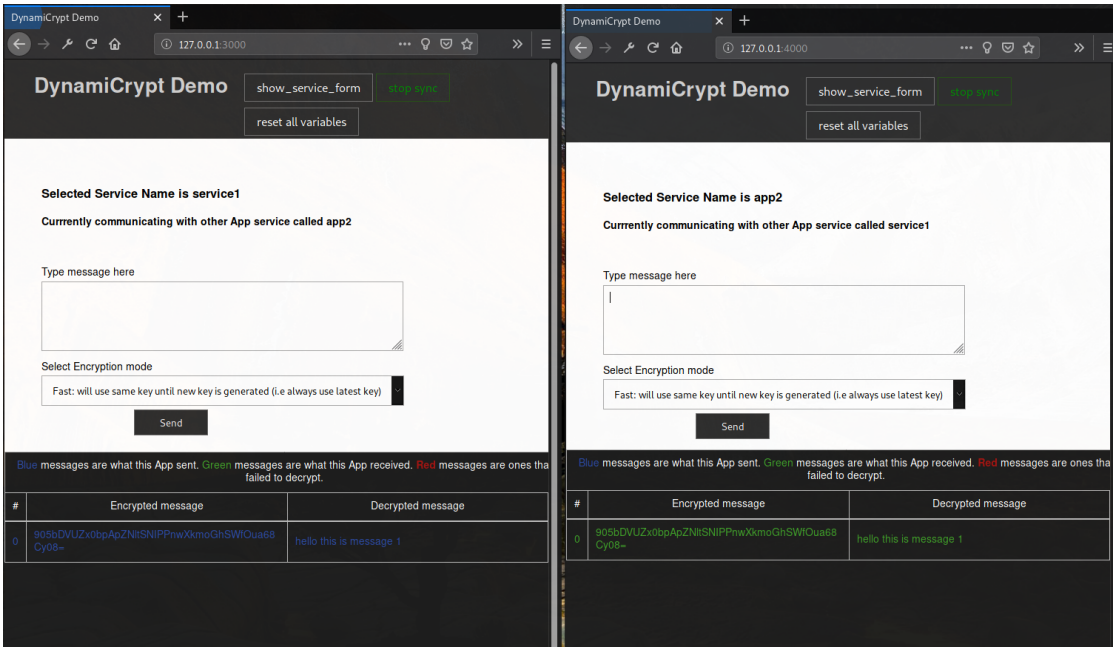

FIGURE 6.11: Encryption Node App view

Figure 6.11 shows the two node Apps after a message "hello this is message 1" was sent using the text box interface of the left App. As you can see the right node App

decrypted the text successfully. I will use Wireshark once again to show the data sent between the Node Apps and the API.

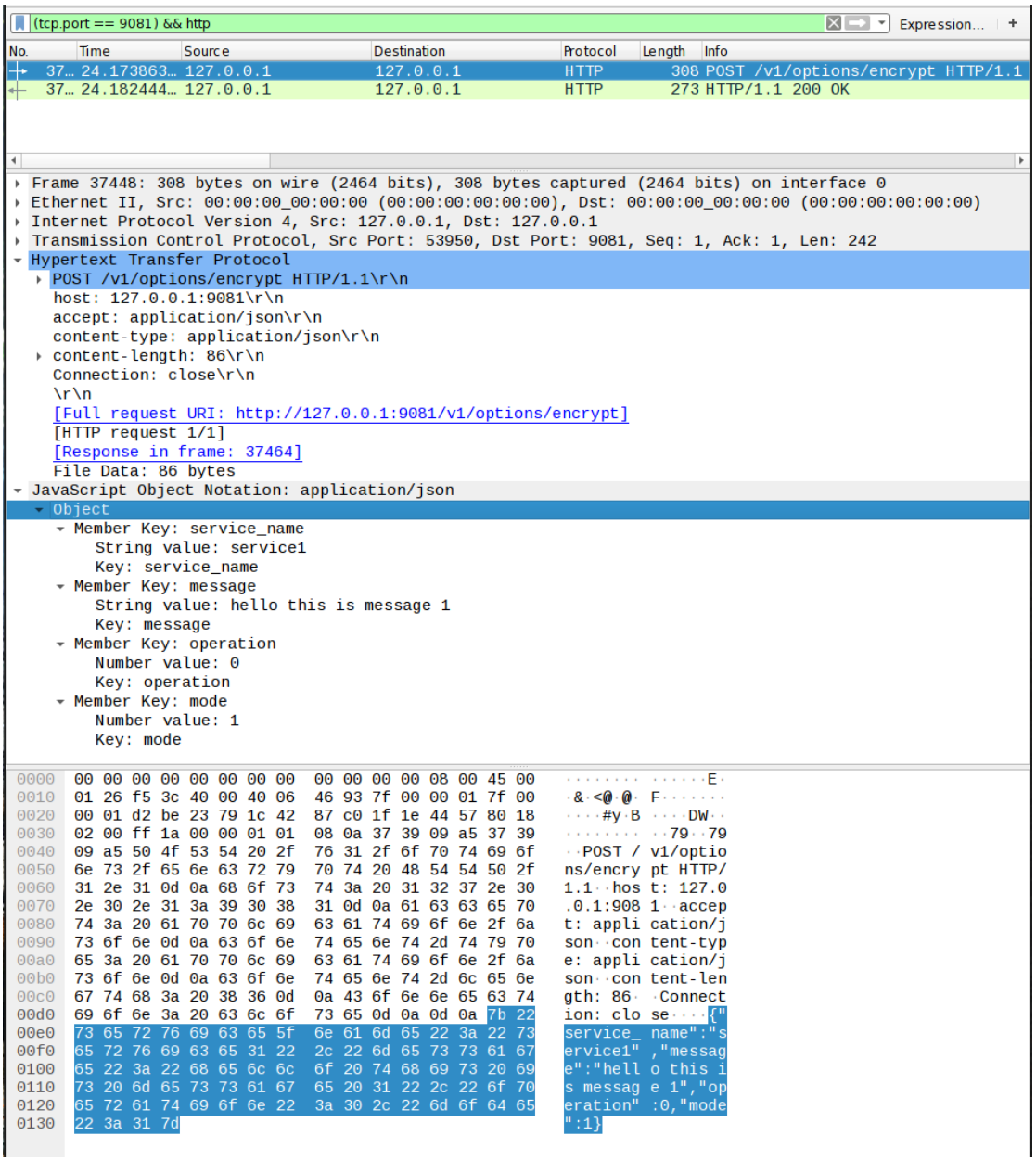

FIGURE 6.12: Node sends encryption message to the API

Figure 6.12 shows the node app at port 3000 sending a post request to the encrypt route of the API. The data it sends are as follows, service name so the API can identify which key store is appropriate for this service, message the plaintext that will be encrypted, operation this just means whether to encrypt or decrypt because the same route is used, in this case it is 0 but for decryption it would be 1. And finally the encryption mode, mode 1 was chosen in this case.

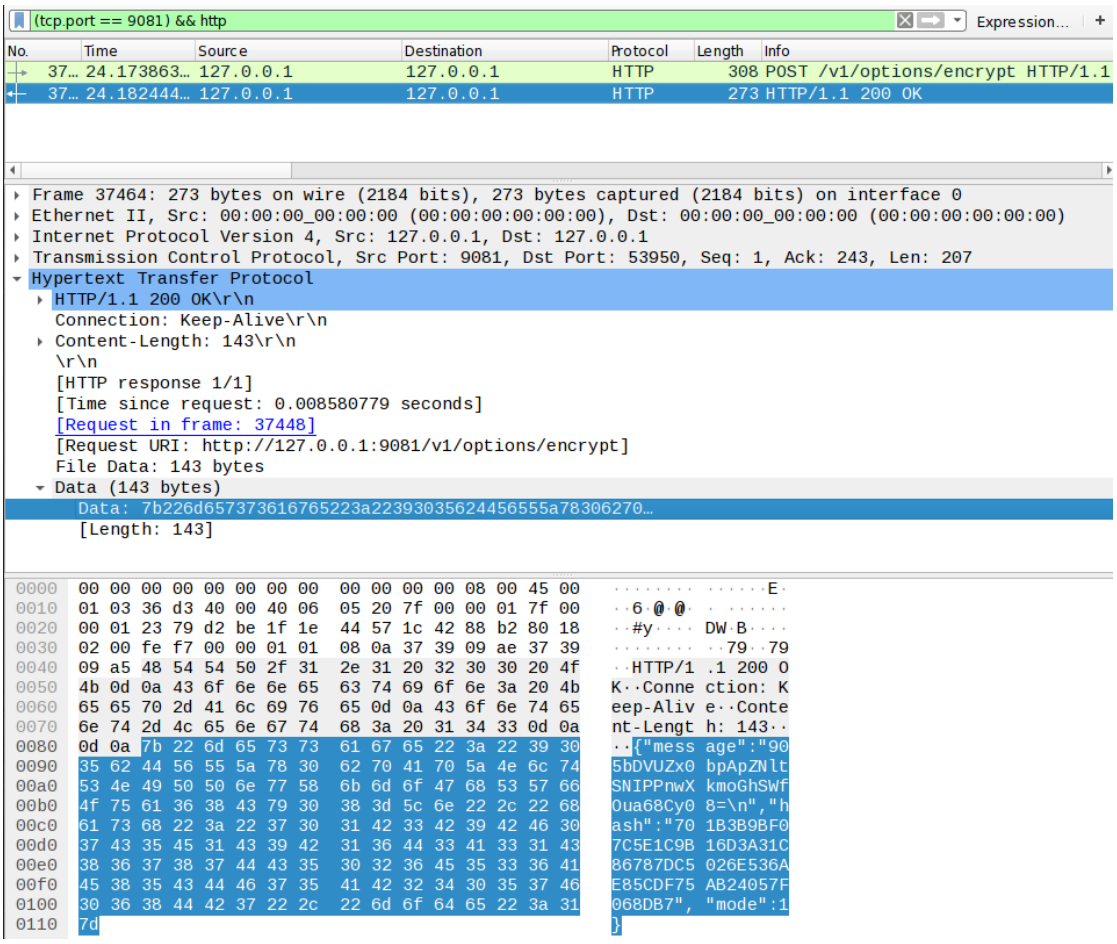

FIGURE 6.13: API responds with ciphertext and hash

Figure 6.13 shows the response of the API, the API replies with the ciphertext as the message, followed by the hash of the plaintext and the mode to be used for decryption. This data will then be forwarded to the other node App by posting it to its decrypt route as seen in figure 6.14

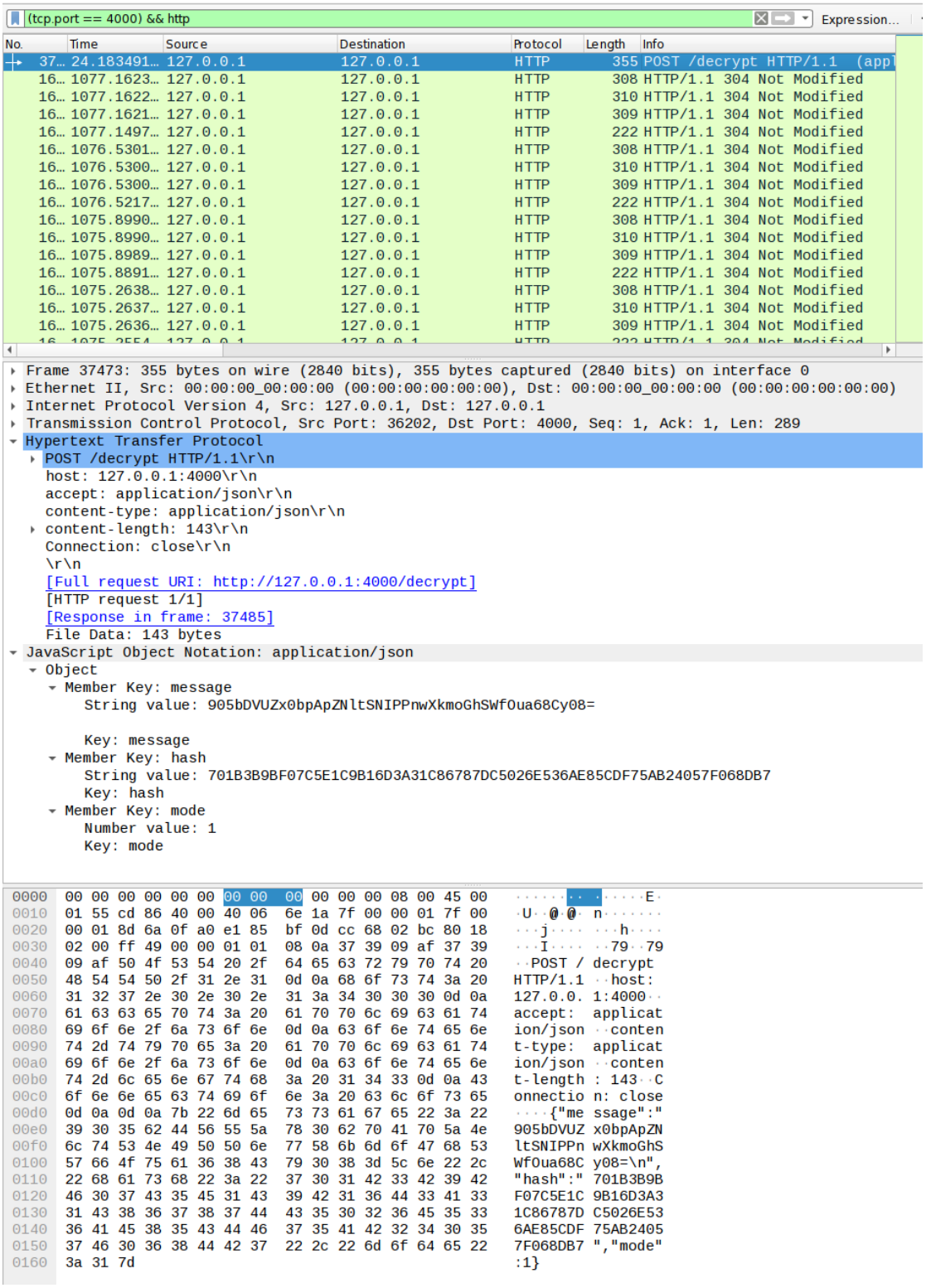

FIGURE 6.14: Node App at port 3000 sends ciphertext to Node App at port 4000

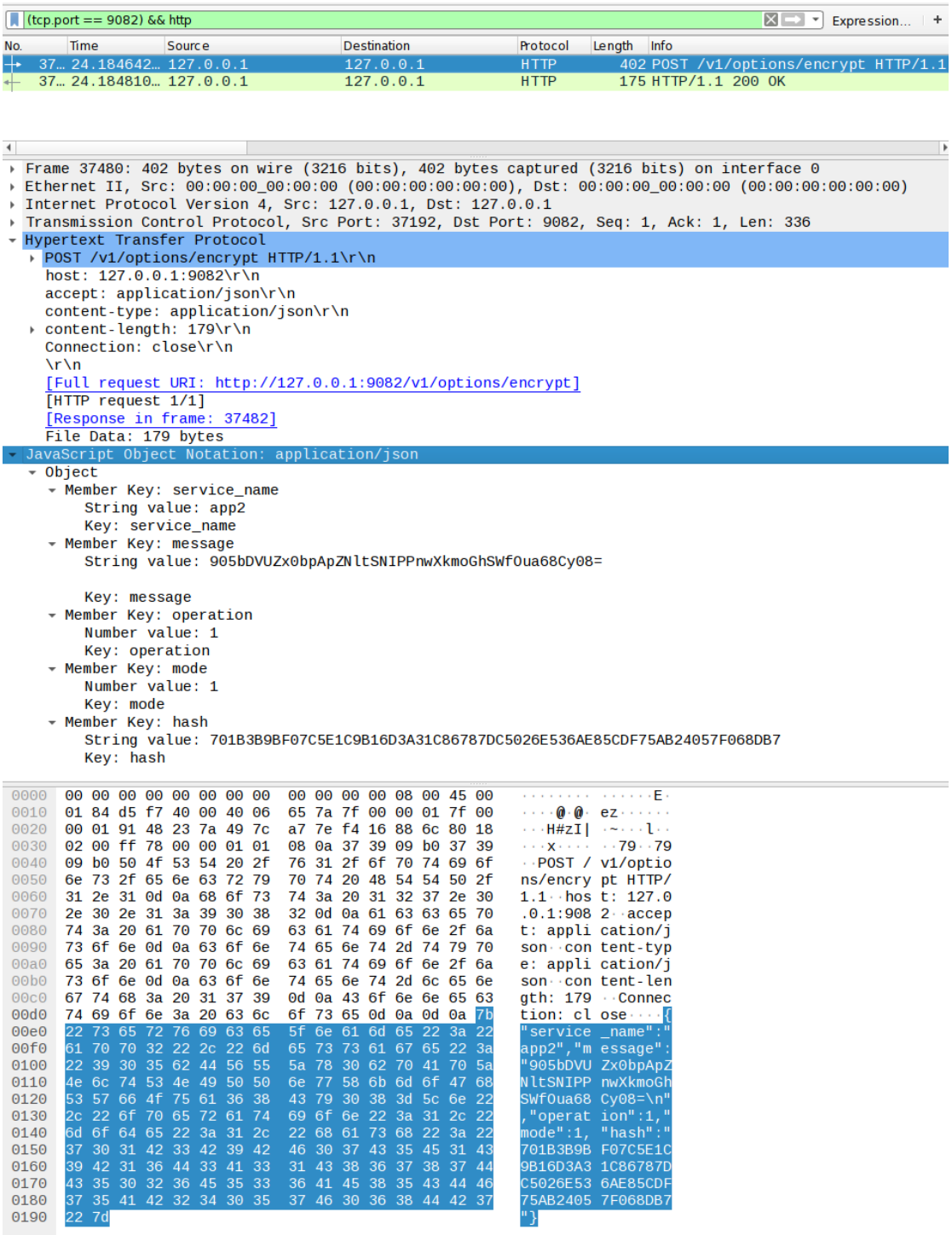

FIGURE 6.15: Node sends decryption message to the API

Now the node app on port 4000 needs to decrypt the ciphertext so it sends over its service name, ciphertext, operation is 1 this time for decryption, mode is 1 again and finally the hash of the plaintext forwarded from the other API to the API it is using on port 9082. This can all be seen in figure 6.15

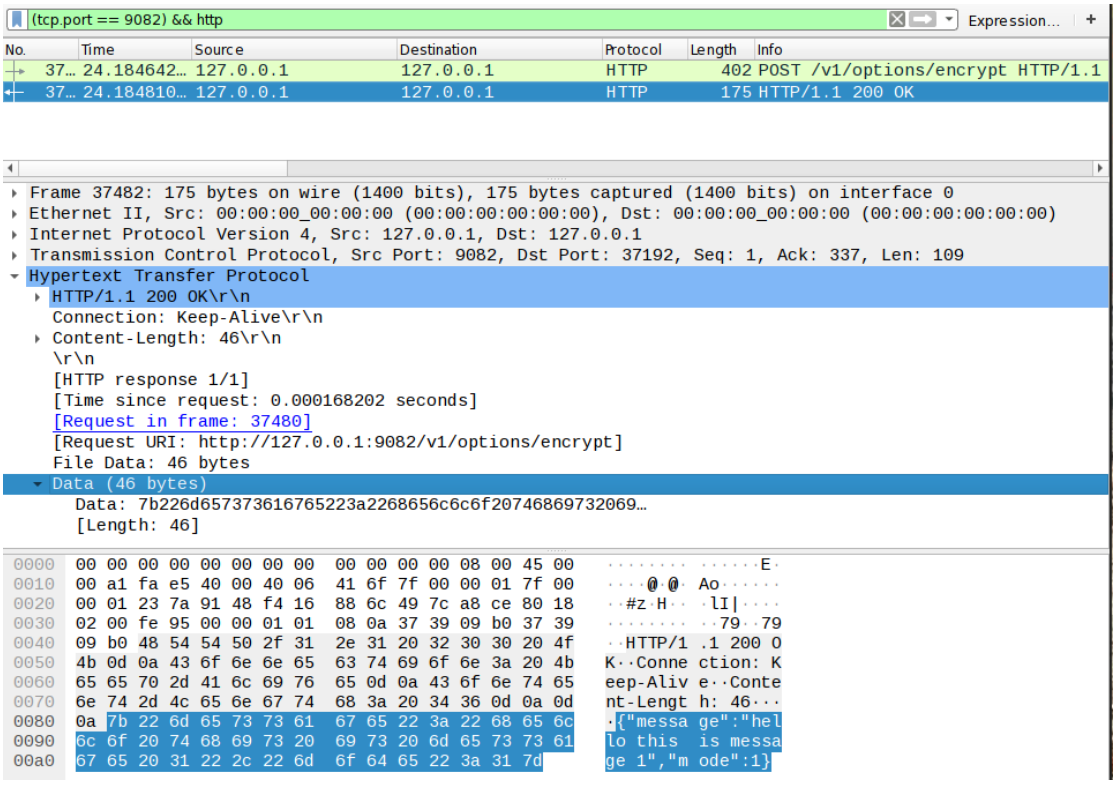

FIGURE 6.16: API responds with plaintext

Figure 6.16 is a response from the API, the message was decrypted successfully and is simply sent back to the node app in plaintext the mode is also attached but this is not required for anything other than statistics.

Next I would like to demonstrate the difference between the encryption modes using the node apps. Basically the point here is that if sending the same message i.e the same plaintext over and over in a non dynamic encryption environment the cipher text that will be produced will always be the same. You can see the following demonstration in video form which I mentioned before at this link https://www.youtube.com/watch?v=LsR4XsGrDCYt=70s

Therefore since this project is all about dynamic encryption the ciphertext will be different every time if using the same plaintext. Note that it is guaranteed to be different every time if mode 2 is used since that will use a unique key for each message, mode 1 on the other hand is built for speed and therefore will use the latest available key therefore if frequent identical plaintexts are sent then there is a possibility that some of the ciphertexts will be the same.

To test this properly a human would be too slow to type out the message each time so once again I will use a script which will basically submit the form on the browsers behalf. This python script is called send message.py and is once again located inside the GitHub repository. The script looks like this.

```
#!/usr/bin/env python3.7

import requests
import time

Service_address = "127.0.0.1"
Service_port = 3000

message = "hello -> using mode 1"
encrypt_mode = 1  # 1 for fast. i.e. encrypt using the latest key. 2 for
    secure only use 1 key once.

how_many_times = 10

url1 = "http://" + Service_address + ":" + str(Service_port) + "/encrypt"
data = {'message':message, 'encrypt_mode': encrypt_mode}

for i in range(0,how_many_times,1):
  print("sending")

  r = requests.post(url = url1, data = data)
  time.sleep(0.6)
```

As you can see this will send the message "hello -¿ using mode 1" to the encrypt route of the node app on port 3000 (yes both the node app and the API have an encrypt route, only realised this might be confusing just now). This message will be sent 10 times and after each message is sent there will be a break for 0.6 seconds. Firstly I will use encrypt mode 1.

1	pgFyApjKD6KRjR2IT1M5Zcr3OLV dMwSHOkkUfL/JQSw=	hello -> using mode 1
2	JOY6oekMFvn4VYWZj6KSbsK+t M+f0yk+ooSclxsrQdA=	hello -> using mode 1
3	BVvPzId6xcTlq4syOqZk1r69NYqD uyqrV6J0uEXGR7U=	hello -> using mode 1
4	hepMYKaXSt9Jb/xK8ANtNnY/UrZt kvvn3SXMSGWlOHo=	hello -> using mode 1
5	hepMYKaXSt9Jb/xK8ANtNnY/UrZt kvvn3SXMSGWlOHo=	hello -> using mode 1

1	pgFyApjKD6KRjR2IT1M5Zcr3OLV dMwSHOkkUfL/JQSw=	hello -> using mode 1
2	JOY6oekMFvn4VYWZj6KSbsK+t M+f0yk+ooSclxsrQdA=	hello -> using mode 1
3	BVvPzId6xcTlq4syOqZk1r69NYqD uyqrV6J0uEXGR7U=	hello -> using mode 1
4	hepMYKaXSt9Jb/xK8ANtNnY/UrZt kvvn3SXMSGWlOHo=	hello -> using mode 1
5	hepMYKaXSt9Jb/xK8ANtNnY/UrZt kvvn3SXMSGWlOHo=	hello -> using mode 1

FIGURE 6.17: Encryption / Decryption mode 1 part 1

FIGURE 6.18: Encryption / Decryption mode 1 part 2

Figures 6.17 and 6.18 show the output of the encryption and decryption table in the NodeJs apps after running the script. Here the plaintext is all the same and as you can see on the right node app they have decrypted successfully. The purpose of this test is to examine the cipher text and see which ones if any used the same key. message 1 used a key unique to itself. message 2 used a key unique to itself. message 3 used a key unique to itself.

however message 4 and 5 have the same ciphertext therefore they have used the same key this is because by the time message 5 was encrypted no need key was generated yet and since this is mode 1 it just used the latest key available which happened to be the same one as message 4.

message 6 used a key unique to itself.

message 7 and 8 once again used the same key as each other.

message 9 used a key unique to itself.

message 10 used a key unique to itself.

Given that only two plaintexts were encrypted using the same key twice this is actually a very good result for mode 1. In my previous testing normally you would see three messages in a row encrypted with the same key. This time the keys were generated quicker than expected. How keys are generated will be covered in the sync-server section after this API section.

Now we will have a look at using mode 2 for encryption to use mode 2 I simply modified two variables in the script.

```
message = "hello -> using mode 2"
encrypt_mode = 2
```

FIGURE 6.19: Encryption / Decryption mode 2 part 1

FIGURE 6.20: Encryption / Decryption mode 2 part 2

Figures 6.19 and 6.20 show the output of the encryption and decryption table in the NodeJs apps after running the script with the modifications. By looking at the cipher-texts part of the table of the Apps you can see that they are all different despite the same plaintext being used, therefore each message was encrypted using its own unique key and decrypted successfully as can be seen on the right app.

This means that my algorithm works perfectly for both modes of operation.

As we can see both modes demonstrated dynamic cryptography. Mode 2 is in my opinion is more secure as each message no matter what is encrypted using a unique key. However there is a possibility that you will run out of suitable keys if sending loads of data frequently. If this happens you will just have to wait a little bit for new keys to be generated and encrypt again. Mode 1 therefore is perfect if you would like to send loads of data at regular intervals since there will be no issue with running out of keys, you will still reap the benefits of dynamic encryption since it takes around one second or so to generate new keys (testing time of key generation will be seen later).

This will conclude the explanation and testing of aspects related to the API. If I were to explain everything related to the API it would take another 50 pages or so to go through everything therefore I selected the most impact full aspects of the API based on the outcomes of the project and left out most of the technical aspects. This section should have provided a good understanding of most of the more important operations of

the API, it is recommended to browse through the GitHub repository for a full exposure of the API.

6.3 Sync-server

This section of the Document will cover the sync-server that is responsible for managing the Tree parity machines. As it was mentioned before the sync-server uses a custom peer to peer network where there are essentially two types of peers. One type is the one that waits for another peer to connect to it and the second type is the one that connects to the waiting peer. This is because there is no need to use a full fledged peer to peer network for the purposes required for DynamiCrypt there is no need for all the other features that come with a proper peer to peer network like peer discovery and such.

Since I used Boost Asio for the sync-server it is slightly different in the way the API was written since that used Pistache for networking. With boost asio a call back asynchronous architecture was used, Boost Asio also supports futures and promises however they were not covered deeply in the Book I used to learn Boost Asio. Nevertheless call backs are perfectly fine and efficient.

The class that contains all of the interactions with sockets and directly with the network is the peer class. There is actually no such a thing as Sync-server code wise, I just refer to everything else that is not the API and not the Tree Parity Machine as Sync-server. This is because there are many peer objects stored in a vector which is global and there are functions for creating new peers, updating current and deleting unused peers inside definitions.cpp, which is not a class its just another source file from which some other class can call functions from if they included the appropriate header definitions.hpp.

As mentioned before there are two types of peers the only difference between these two is in the start function as seen here.

```
void peer::start(std::string service_name, std::string partner_name){
  { boost::recursive_mutex::scoped_lock lk(read_lock);
    peers.push_back( shared_from_this());
  }
  started_ = true;
  service_name_ = service_name;
  if(sock_using_ep){ // this makes the connection so write straight away
    //endpoint_(ip::address::from_string(ip_address_), ip_port_);
    endpoint_ = boost::make_shared<boost::asio::ip::tcp::endpoint>(boost::
    asio::ip::address::from_string(ip_address_), ip_port_);
    sock_.async_connect(*endpoint_, MEM_FN3(on_connect,_1,service_name,
    partner_name));
```

```
  } else { // this will listen to connection so read.
    do_read();
  }
}
```

As you can see the variable sock-using-ep determines which type of peer this is. This is set when creating a peer object and basically means is the socket that this peer will use be of type endpoint or not. If so then the address and port of another peer on the network will be loaded up and an asynchronous connection is made to the other peer.

```
sock_.async_connect(*endpoint_, MEM_FN3(on_connect,_1,service_name,
    partner_name));
```

Because call backs are used a callback function must be provided which runs when the peer establishes a TCP connection with another peer. This call back function is on-connect, the service name and partner name are also passed in to be verified by the other peer (more on this later). If the peer is of type waiting then it simply waits for a peer to connect to it on the do-read function.

```
void peer::do_read(){
  { boost::recursive_mutex::scoped_lock lk(read_lock);
    async_read(sock_, boost::asio::buffer(read_buffer_), MEM_FN2(
    read_complete,_1,_2), MEM_FN2(on_read,_1,_2));
  }
}
```

This read function performs an asynchronous read operation this time there are two call back functions read complete and on read, this is required because Boost asio doesn't know when all of the data is sent because you can configure it anyway you want. read complete basically scans the buffer for a newline character as that is what is set to determine the end of the TCP message and returns a boolean.

```
size_t peer::read_complete(const boost::system::error_code & err, size_t
    bytes){
  if ( err) return 0;
  bool found = std::find(read_buffer_, read_buffer_ + bytes, '\n') <
    read_buffer_ + bytes;
  return found ? 0 : 1;
}
```

When this returns 1 Boost Asio then knows that the message has been fully read and calls the on read function which can then access the read buffer and perform calculations on the message.

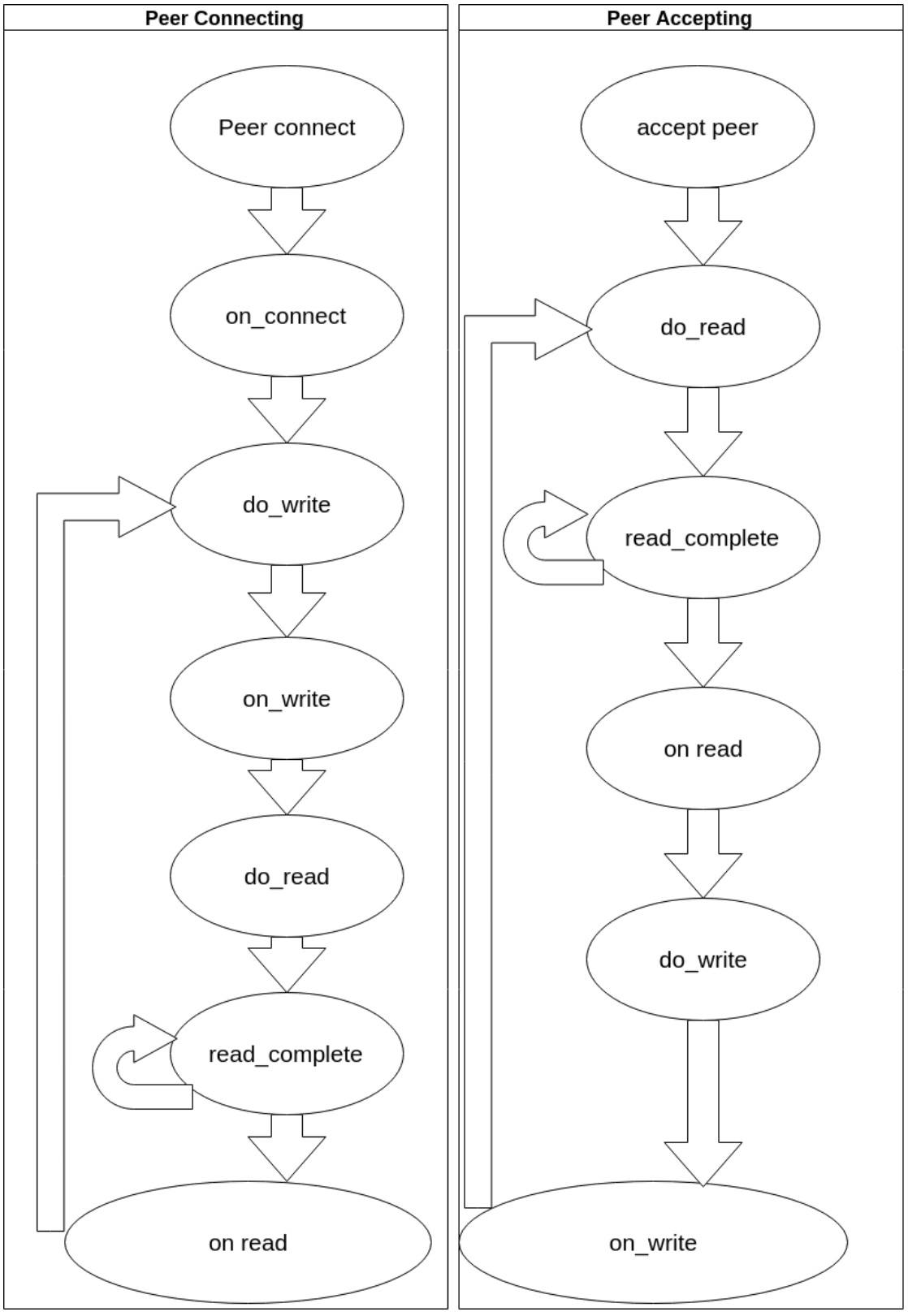

FIGURE 6.21: Asynchronous flow diagram of peer

Before continuing with the rest of the operations a quick overview of the asynchronous code flow is presented in figure 6.21. The two boxes represent the two different peers and

the order of the asynchronous operations that they follow. The connecting peer after connecting to a reading peer prepares a packet and and calls the do write operation. After the do write operation finishes writing all of the data to the socket on write is called which then calls do read, then read complete gets called multiple times until a newline character is detected, then on read is called and there are other functions called from on read that call do write and the cycle continues.

The accepting peer as referred in the diagram is the waiting peer. It is important for the connecting peer and waiting peer to not read or write to the socket at the same time, so you will notice that every time the connecting peer is writing the accepting peer is reading and visa-versa. The waiting peer starts of its cycle by reading since there is no data on the socket while there is no peer that is about to connect it simply does nothing. Boost Asio in the background does check periodically for any data in the socket but this behaviour is hidden from the developer. After a peer connects to the waiting peer it immediately starts checking if the message was fully sent by repeatedly calling read complete when a new byte arrives. When all the data is received on read is called which creates a new message and calls do write when the message is fully written on write is called which once again calls do read and the cycle once again continues.

Currently there are four types of messages or at least four different categories of messages. Messages are constructed in the following fashion message-type + tab character + data + tab character + data + ... + new line character.

Messages of message-type 1 are used for sending synchronisation data used to synchronise tree parity machines.

Messages of message-type 2 are used for when the peer initially connects to another peer this is the first message in the conversation.

Messages of message-type 3 are essentially a reply to message-type 1 with more information needed to setup the tree parity machines.

Messages of message-type 4 are for resetting tree parity machines if a key is found, or if the tree parity machines reached the maximum iteration count

Bellow is a snippet of all the different types of messages and the function from which they are called from.

Note the << operator in C++ just adds data to a buffer so ss << "1" << variable << "string" would be the same as ss = "1" + variable + "string" in python for example.

```
// called from std::string TpmNetworkHandler::sync_tpm_message_one(int
    tpm_id)
ss << "1\t" << tpm_id << "\t" << tpm_networks_[index].id() << "\t" <<
    tpm_networks_[index].iteration() << "\t" << random_input_vector << "\t"
     << tpm_result << "\t" << 1 << "\n";


// called from std::string TpmNetworkHandler::sync_tpm_message_one_advanced
    (int tpm_id, std::vector<std::string> & parsed_msg)
ss << "1\t" << tpm_id << "\t" << tpm_networks_[index].id() << "\t" <<
    tpm_networks_[index].iteration() <<  "\t" << random_input_vector << "\t
    " << tpm_result << "\t" << message_type_to_process  << "\t" <<
    tell_machine_to_update<< "\t" << old_input_vector << "\t" << key_hash
    << "\t" << random_input_for_key << "\n";


// called from void peer::on_connect(const error_code & err, std::string
    service_name, std::string partner_name)
ss << "2\t" << id << "\t" << tpm_handler.get_iteration(id) << "\t" <<
    service_name << "\t" << partner_name << "\n";

// called from void peer::on_init(std::vector<std::string> & parsed_msg)
ss << "3\t" << id << "\t" << tpm_handler.get_iteration(id) << "\t" <<
    tpm_handler.get_partner(id) << "\n";


// called from void peer::on_sync(std::vector<std::string> & parsed_msg)
    and void peer::on_linking(std::vector<std::string> & parsed_msg) also
    uses the same parameters
ss << "4\t" << tpm_id << "\t" << tpm_handler.get_iteration(tpm_id) << "\t"
    << 0 << "\t" << 0 << "\n";


// called from std::string TpmNetworkHandler::sync_tpm_message_one_advanced
    (int tpm_id, std::vector<std::string> & parsed_msg)
ss << "4\t" << tpm_id << "\t" << tpm_networks_[index].iteration() << "\t"
    << 0 << "\t" << 1 << "\n";
```

Chapter 7

Discussion and Conclusions

In this chapter, you should expand upon (and initially reflect upon) the discussion and conclusion of the research phase of the project. The expectation here is that you should discuss the results presented in the previous evaluation section of the project in their totality (i.e. as a whole) from which you will then draw clear conclusions both on the quantitative and qualitative aspects of the overall project. This chapter should be a about 2000 words long (5 pages of text - 1600 words of discussion and 400 words of conclusion). This may vary depending on quality. The conclusion section of this report should conclude the project.

Some suggested sections (the nature of this chapter should be discussed in detail with your term 2 supervisor):

7.1 Solution Review

Discuss how well your solution solves the problem, based on your results from the evaluation chapter.

7.2 Project Review

Discuss how well you addressed the project, and what you might do differently if you were to do it again. Make sure to identify how you handled any problems that arose during the project. Identify key skills that you learnt during the project, and clearly describe how you applied these, and how you might apply them differently if you were to do a similar project.

7.3 Conclusion

Enumerate the main conclusions you have got in terms of background, problem description and the solution approach you have come up with. Detail your primary and any secondary conclusions from your project.

7.4 Future Work

Discuss any proposals for completion of the project, or for enhancements, or for re-design of your solution or software. Enumerate all the things you would have wanted to do should you have more time to work on this project.