# Coordinate Routing in the Lightning Network

Shivanand C Kohalli
Student Number: 4751116

Delft University of Technology

# Coordinate Routing in the Lightning Network

Master's Thesis in Embedded Systems

Distributed Systems group
Faculty of Electrical Engineering, Mathematics, and Computer Science
Delft University of Technology

Shivanand C Kohalli

23rd September 2019

**Author**

Shivanand C Kohalli

**Title**

Coordinate Routing in the Lightning Network

**MSc presentation**

Sep 30, 2019

**Graduation Committee**

| | |
|---|---|
| Dr.ir.J.A. Pouwelse | Delft University of Technology |
| Dr. S. Roos | Delft University of Technology |
| Dr. M. Nasri | Delft University of Technology |

## Abstract

Blockchains such as the Bitcoin facilitate the online transaction between users without an intermediate financial institution and can serve as a global currency. However, at the moment, blockchain solutions are suffering from low transaction throughput and high delays. For instance, the bitcoin network processes at most 7 transactions per second. Blockchain second layer solutions aim to solve this transaction scaling problem with the idea of creating off-chain transactions using payment channels. For a payment channel, the blockchain network is only required to initialize and terminate the channels. Otherwise, they can finalize transactions between pairs of users instantly without using the blockchain network. The Lightning Network is one such payment channel solution that creates an overlay network on top of the blockchain and routes payments across users. Currently, the Lightning network uses source routing to route payments from a source to a destination. However, source routing does not scale to large networks.

In this thesis, we begin by establishing the requirements for routing in the Lightning Network. Next, we analyze generic payment routing algorithms based on the set requirements and understand their limitations. Further, we design a routing algorithm for the Lightning Network that builds upon the key ideas found in the previous work. We, in particular, provide design solutions for computing transaction fees and communicating errors in a privacy-preserving manner. These problems were not addressed in any earlier works. Finally, we implement our algorithm in an active Lightning Network codebase. We evaluate it with regard to security, privacy, and performance requirements and also compare the same with the existing solution.

# Acknowledgments

I would like to thank my supervisor, Stefanie, for providing a wonderful opportunity to work on this research topic. Thank you, for steering me in the right directions when I faced obstacles, for all the weekly meetings and the valuable feedback. I also want to thank Oguzhan for the cryptography discussions we had.
I made many good friends in Delft during my studies, would like to specially thank Amitabh, Umeer and Bhavya for being a source of motivation when needed.
I would like to express my heartfelt gratitude to my parents, for providing unconditional support to all my decisions and believing in me. This dream would not have been possible without them.
While I was pursuing my Masters in Netherlands, I lost a dear one, my grandpa (*Ajja*), back in my hometown. *Ajja*, you have always been an inspiration to me, I will always try to inculcate the values learned from you. This work is dedicated to you.

Shivanand C Kohalli

Delft, The Netherlands
23rd September 2019

# Contents

# List of Figures

# Chapter 1

# Introduction

## 1.1 Blockchain

Blockchain is a technology which promises to revolutionize the current payments system. In the past decade, many prominent blockchain protocols like Bitcoin [49], Ethereum [61] and Ripple [23] have come into existence. People have already started to use this technology for their financial needs. In the month of July 2019, the Bitcoin network witnessed at least 270,000 transactions per day [2] and its net market cap as an asset class was worth over $200 Billion.

The blockchain can be described as a decentralized, immutable public record of transactions. Every transaction in a blockchain network is broadcast to all the participating nodes in the network. The nodes validate this transaction and add it to a pool of transactions called a block. At the core of any blockchain protocol, a consensus algorithm decides the next block that will be added to the blockchain. Subsequently, the next block will be linked to the previous record of blocks forming a virtual chain of blocks, hence the name blockchain. Anyone can read and verify the correctness of the data on the blockchain but cannot tamper or delete it. Such an architecture has a vital implication that even untrusted entities can directly interact with each other without requiring the presence of a trusted intermediary. These properties are relevant to financial institutions, banks and even common people who would want to avoid the middleman costs for different operations.

In spite of the attractive properties the blockchain provides, there are still some hurdles to overcome for widespread adoption of this technology. Performance of the blockchains is one such factor that requires significant improvements. For instance, the bitcoin blockchain currently can handle around 7 transactions per second [28]. On the contrary, Visa which facilitates electronic fund transfers all over the world can processes around 1700 transactions per second on an average [20]. This is not only pertaining to Bitcoin, but a general issue of blockchains [38]. This is an important bottleneck that needs to be resolved for blockchains to be considered viable mechanisms for day-to-day payments.

According to recent works, the solutions to reduce the bottleneck in blockchains can be classified into two general approaches namely:

1. Layer-one solutions: These solutions focus on improving the core components of the blockchain. This maybe dealing with the consensus protocols [49, 26, 43, 24], modifying the block sizes or sharding [44, 37].

2. Layer-two solutions: Also called as off-chain solutions, are built on top of the blockchain network, but still rely on the securities of the underlying blockchain. The main idea is to create transactions using payment channels [53, 31, 30] or state channels [48, 17] between the nodes.

The topic of interest in this thesis is the payment channels as they promise near instantaneous transactions and reduced fees allowing blockchain to scale.

## 1.2 Payment Channels

In a payment channel two parties establish a private channel between them. This channel will be governed by a set of rules, which allows both the parties to update the current balances in the channel. To open a channel, both the nodes need to lock funds as collateral on the blockchain. The locked funds can only be spent with the permission of both the nodes. Once a channel is established, both the nodes can update the balance in the channel as many times they desire, according to the preset rules. To update the channel balance no broadcast is made to the blockchain network. Later, the two nodes can terminate their channel by broadcasting their latest balance to the blockchain. With payment channels, only the opening and closing of a channel is broadcast to the blockchain. Thus, if payment channels are used for most of the transactions, it can lead to drastically reducing the transaction load on the underlying blockchain.

When a payment channel is established between two nodes, they can send and receive funds between them as many times on that channel. However, if a channel is not established with some node they need to either open a channel or send funds directly over the blockchain. Opening a payment channel with all such nodes is not practical because:

- Opening and closing a channel will cost the nodes. As both these activities requires the blockchain, a mining fee will be deducted for the operations.

- The sender needs to wait until the channel is opened for doing the first payment. A channel is considered open when the blockchain records in a block that both the nodes have some funds locked. Also, that particular block must have sufficient confirmations. These activities take time according to the underlying blockchain.

A payment channel network aids to solve this problem. In a payment channel network, funds can be routed across multiple nodes from a source to the destination if there is a path of channels between them. Thus, there is no need to open payment channels with everyone, only a viable route needs to be found between nodes.

Some promising designs for the payment channel networks have been proposed [53, 17, 31]. In this thesis we will focus on the Lightning network [53] as a payment channel network solution. The motivation for this choice is that Lightning network is designed to work on Bitcoin-like blockchains and it already has an active mainnet deployed [9, 4, 6].

### 1.2.1 Lightning Network

The Lightning network leverages the smart contract functionality of Bitcoin-like blockchains to create a network of user-generated bi-directional channels to send payments back and forth across the network. Clearly, such a network can aid bitcoin to scale to a much higher transactional throughput. The mainnet of the Lightning network has around 9000 nodes with over 30000 channels open and a total capacity of about 820 Bitcoins ( $9 Million) in August 2019 [11]. The growth in the capacity of the network from September 2018 can be seen in the Figure 1.1.

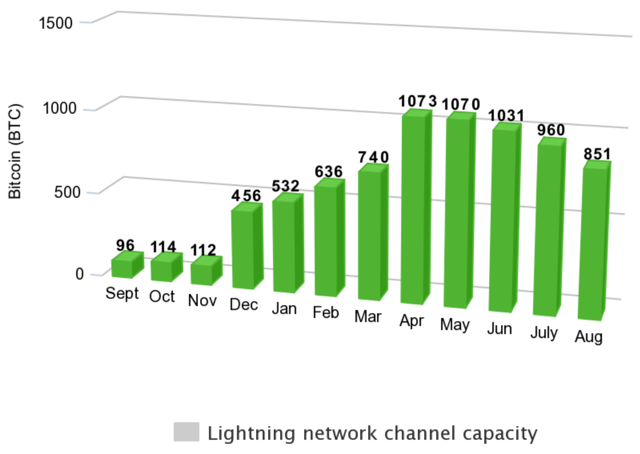

Figure 1.1: Growth in the Lightning network [18]

In Lightning, payments can be routed across multiple nodes from a source to the destination if there is a path of channels between them. A viable route having enough capacity to transfer the funds needs to be found. As Lightning is a distributed network, there is no central source to determine these routes. The nodes need to cooperate among themselves to find such routes. The Lightning network white paper[53] focuses on the payment algorithm for off-chain transactions but does not throw light into how routing should work. Later, the specifications for the Lightning were developed terming them as BOLTS (Basis of Lightning Technology Standards)[10]. According to the BOLTS, the nodes use source routing to route payments between them. Source routing is a routing technique where the source node decides the whole path of a packet in the network to reach the destination. To do so, the source adds the whole path information in the packet so that the intermediaries can decode and forward according to the instructions.

## 1.3 Motivation

To accomplish source routing in the Lightning Network, every node maintains a local view of the complete network. The nodes refer to this snapshot locally to determine the routes. This technique is indeed advantageous in discovering the most efficient routes. It allows the source to discover all the possible routes and determine the best one that has the lowest overall routing fees. However, a major downside is that to maintain the local snapshot of the network, every node needs to broadcast regularly its state and the required channel states to the whole network. This might be acceptable in a small network but if the network keeps growing as shown in the Figure 1.1 it will be highly inefficient to maintain a local snapshot of the network as even a small change in the state needs to be broadcast. The Lightning network is still in its initial stages, but steadily growing everyday. Thus, other efficient routing techniques need to be explored which can handle this scaling issue as the network evolves.

## 1.4 Problem statement and research objectives

The motivation to introduce a new routing algorithm in the Lightning network is discussed in Section 1.3 which translates into the following problem-statement for the thesis:

*Design a payment routing algorithm for the Lightning network that builds upon key ideas in previous work, but takes the specifics of Lightning into consideration.*

This problem expands to the following research objectives for the project:

1. Determine the characteristics of the Lightning network and formalize a set of requirements for routing payments.

2. Study and analyze the generic routing algorithms for payment channel networks and determine their limitations with respect to the established requirements.

3. Select a routing algorithm that satisfies most of the requirements and does not have inherent issues which cannot be addressed. Determine the limit-

ations of the chosen algorithm and establish the ones that will be resolved through this thesis.

4. Define the goals, the evaluation criteria, and hence the designs to resolve the identified problems in the routing algorithm.

5. Implement the new routing algorithm in an active Lightning network codebase.

6. Evaluate the performance of the routing algorithm based on various metrics on an emulation network.

## 1.5 Contributions

This thesis answers the problem-statement specified in Section 1.4 by solving the individual objectives. The contributions can be summarized as answers to these individual research objectives:

- The Lightning network is studied in detail and its characteristics for routing payments are derived. With the help of these characteristics and the properties of the Lightning network, the requirements for routing payments are proposed.

- A thorough literature study is performed to learn the state of the art routing algorithms in payment channel networks. The following routing algorithms are analysed: Flare [54], Flash[60], Spider [58], Ant [39], Silent-Whisphers [45] and SpeedyMurmurs [56]. The limitations of each of these algorithms are established as per our requirements.

- After analysis, SpeedyMurmurs is chosen as the generic routing algorithm as it satisfies most of the requirements when compared with the rest. The algorithm is scalable, efficient, provides formal security guarantees and cost-efficient as well. The shortcomings of SpeedyMurmurs are also discussed and specifically, two of them are chosen to be fixed upon in this thesis.

  - How the source node will acquire the routing fees and CheckLock-TimeVerify (CLTV) expiry before initiating routing?
  - How the routing nodes in the network can send errors securely to the source node?

- To solve both the issues, first, the security goals are defined which act as guidelines the solutions must follow. The security goals are defined with respect to confidentiality, integrity and privacy. Next, the detailed designs of the solution are discussed. Finally, the design solution is evaluated with respect to the set security goals.

- The generic SpeedyMurmurs algorithm along with the proposed design changes is implemented and integrated into an active Lightning codebase (LND [9]). The LND is made to use our routing algorithm for forwarding payments instead of source routing. We also design a system and define all its components to set up an emulation network environment. We use this environment for running different test scenarios.

- We evaluate SpeedyMurmurs and source routing's performance based on the success ratio, success volume, path lengths, transaction delays, network stabilization, stabilization overhead and transaction overheads.

## 1.6 Thesis Outline

The contents of this thesis are outlined as follows: Chapter 2 gives the relevant background on the Bitcoin and Lighting network. Chapter 3 proposes the requirements for routing in the Lightning network, while Chapter 4 gives details on the current state-of-the-art routing algorithms in payment channel networks. Chapter 5 proposes the complete design of our routing algorithm. The system setup to evaluate the performance of the new routing algorithm is described in Chapter 6 and the results of these evaluations are presented and discussed in Chapter 7. The thesis draws its conclusions in Chapter 8 and provides directions for future work.

# Chapter 2

# Background

**Conventions**

The following conventions are used in the document:

- Source/sender/initiator node and destination/receiver/recipient node are the first and the last node respectively on the payment path.
- Hops are the intermediate nodes on a payment path
- A peer is node that has an open channel with a node.
- Erring node: The node which has incurred some error.

In this chapter we give the relevant background to understand the thesis. We first explain about the Bitcoin network. Further, the Lightning network is explored in detail. We discuss multi-hop payments, the process of routing in the Lightning network and also the onion encryption scheme.

## 2.1 Bitcoin

Bitcoin is a distributed system running on a peer-to-peer network. The goal of the network is to collectively maintain a global state. This global state is known as the ledger or blockchain itself. Peers can send and receive currency (cryptocurrency) by broadcasting a transaction onto this network. A transaction typically comprises an input and an output state. The output state embodies the cryptocurrency amount to be sent and a script which defines the conditions to spend that amount. The input state points to the output of an older transaction and also fulfils the conditions to spend the referred output.

Transactions are broadcast to the entire network with the intention of it being included in the ledger. All the receiving nodes verify if the transaction is valid by checking its input and output states. They verify that the

- referred output is not spent already

- the value the transaction wants to transfer is smaller than that of the referred output.

The order of the transactions received and validated by the peers may be different due to the distributed environment of the system. Two or more transactions can claim the same output, and its validity will depend on the order in which the transactions were received. This will lead to inconsistencies within the ledger between the peers. Bitcoin resolves this problem by electing a leader (a miner), who will be responsible to decide the next set of transactions (called a block) that will be going into the ledger. The transactions that are included in a block and gets published to the network are said to be confirmed. Once a transaction has enough block confirmations over it (generally 6), then the payment can be considered irreversible. The leader election and thus the block addition happens through the consensus protocol[49]. This process is time-consuming and happens at an average interval of time of about 10 minutes. Thus, the long delays for a transaction to be confirmed in the Bitcoin network.

## 2.2 Lightning Network

The Lightning network is a payment channel network solution. It aims to reduce the transaction load on the blockchain by creating payment channels between pairs of users. Let us consider an example that Alice and Bob want to transact using the Lightning network. Both of them decide to invest 1.0 BTC to their payment channel. The combined amount of 2 BTC will be the capacity of the channel. They, then cooperate to commit their decided funds to a multi-signature (multisig) address. Agreement of both the parties would be necessary to spend from a multisig address. This initial transaction is called as Funding transaction. This transaction is broadcast to the blockchain. After the necessary block confirmations over the funding transaction, a channel is established between them. Now, if Alice would like to send 0.5 BTC to Bob, she will create a new transaction, spending from the Funding transaction. 1.5 BTC will be assigned to Bob and 0.5 BTC to Alice and it will have relevant digital signatures of the counterparties. This transaction is called Commitment transaction and it is not broadcast to the blockchain. If any of the parties would like to pay each other again, they create a new Commitment transaction as earlier and revoke the previous Commitment transaction. The revocation is important because it is possible that either party can broadcast an older transaction which is beneficial to them. Thus, Alice and Bob can instantly send funds between them without broadcasting on the Bitcoin network. When any of them want to withdraw their funds from the channel, they broadcast their latest commitment transaction to the blockchain. If the nodes try to broadcast any older states of the channel, the counterparty can detect such misbehaviour and penalize it by taking all the funds in the channel.

### 2.2.1 Multi-hop payments

Lightning Network achieves multi-hop payments through Hashed TimeLock Contracts (HTLC). HTLC is a conditional contract made to the payee that, the payee will receive the funds if it produces a cryptographic proof before a certain timeout. Otherwise, the funds are returned to the payer. Every hop in the payment route (including source, excluding destination node) extends an HTLC contract to their successors. In this way, a hop receives an HTLC payment contract of a certain amount on one of its channels. It then forwards a contract with a little lesser amount(a small routing fee is deducted) onto the next hop in the path. This process is extended until the payment reaches the recipient.

To suffice an HTLC contract, the payee needs to provide a preimage (S) of a certain hash value h(S) within a timeout. The S and h(S) are generated by the recipient and only h(S) is shared with the sender initially. When the recipient receives the HTLC contract, it reveals S to its previous node and claims the HTLC output. Similarly, the previous node reveals S to its previous node to claim its HTLC output. This process extends back until the preimage is revealed to the source node, hence fulfilling the transfer. If any hop does not cooperate to disclose S, the counter-party waits until the timelock expires. It further closes the channel and broadcasts the latest channel state, to claim its funds.

The timeouts in the HTLC are enforced through the bitcoin's scripting language opcode CheckLockTimeVerify(CLTV). The CLTV allows a transaction to be locked until a definite number of bitcoin blocks are mined. The CLTV expiry in the HTLC contracts increases from the destination to the source. This ensures that all the hops have enough time to claim their funds back if someone becomes unresponsive. Figure 2.1 shows a simple example of a multi-hop payment where Alice sends Dave 0.5 BTC via Bob and Charlie. It can be noticed that Bob and Charlie deduct a routing fee(0.01 BTC) before forwarding payment to their next hops. The CLTV expiry(should be expressed as block height, for simplicity showed in days) are also in decreasing order from Alice until Dave.

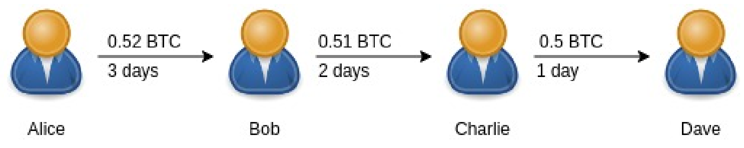

Figure 2.1: An example of a multi-hop payment

In the next section, we will look in detail the route discovery mechanism in the Lightning network.

## 2.3 Routing in the Lightning Network

The payment process begins with the sender node requesting an invoice for the payment from the recipient. The invoice contains the necessary information for the sender to make the transaction. The recipient generates the invoice and it must have the following information:

- blockchain: The blockchain network for which this invoice is generated, for example Bitcoin mainnet, Bitcoin testnet or Litecoin mainnet.
- amount: The amount of bitcoins to be paid
- payment hash: Hash of the pre-image that can settle the HTLC contracts.
- time-stamp: The time at which the invoice was generated.
- expiry: The time after which the invoice is considered invalid.
- CLTV expiry: The CLTV value that has to extended to the recipient node.

The invoice is protected by the recipients digital signature to prevent any tampering. Apart from these compulsory parameters, an invoice can also include parameters such as a description and the hints for the routing.

After receiving the invoice, the sender computes a valid route having enough capacity to transfer the required amount. To compute such a route, the source node requires the knowledge of all available public payment channels with its properties such as routing fees and its capacity. Then using any efficient pathfinding algorithms it constructs a path from the source node to the desired destination. To aid every node with such a local payment channel network graph, the BOLT's [10] propose a gossip protocol between the nodes.

## 2.4 Gossip protocol

The gossip protocol describes the specifications for node discovery and channel discovery. When the nodes receive any gossip message, they validate the relevant signatures in the message. Only if validation is successful, they process the message and broadcast it to their peers. Otherwise, they discard it to stop spreading corrupt messages in the network.

### 2.4.1 Node discovery

As the name suggests the Node discovery allows discovery of different nodes in the network so that connections can be established with them by opening channels. The gossip message to establish node discovery is:

- **node_announcement**: The message mainly has the internet address of the node so that other nodes can start interacting with it.

  The message has the following fields (shown as size_in_ bytes: field_name):

  [64:signature]

  [2:flen]

  [flen:features]

  [4:timestamp]

  [33:node_id]

  [3:rgb_color]

  [32:alias]

  [2:addrlen]

  [addrlen:addresses]

  The internet address here can be an IPv4/IPv6 address or even a Tor [19] onion service address.

### 2.4.2 Channel discovery

Allows nodes to discover all the new public channels that are created across the network. There are three messages designed to facilitate this functionality:

- **announcement_signatures**: The two nodes operating a channel need to express to each other their willingness to make their channel public. They do so by sending this message to each other. The message has the following fields (shown as size_in_ bytes: field_name):

  [32:channel_id]

  [8:short_channel_id]

  [64:node_signature]

  [64:bitcoin_signature]

  The digital signatures in this message are generated from the channel_announcement message. Both the nodes craft the channel_announcement message first but do not send to each other. Subsequently, generate generate their signatures for channel_announcement and populate it in the announcement_signatures message. Only the announcement_signatures message is sent to the node at the other endpoint of the channel.

- **channel_announcement**: The purpose of this message is, to broadcast to the entire network about the existence of a new public channel. Before broadcasting, the nodes at either end of the channel wait for announcement_signatures message from each other. When they receive this acknowledgement of making their channel public from each other, they can complete the construction of channel_announcement message. This message proves to all other nodes in the network, the presence of a channel between node_1 and node_2. In order to do this, they have to

1. prove that the funding transaction pays to bitcoin_key_1 and bitcoin_key_2
2. prove that node_1 owns bitcoin_key_1
3. prove that node_2 owns bitcoin_key_2

Using the short_channel_id in this message, the nodes can verify that the funding transaction pays to bitcoin_key_1 and bitcoin_key_2. The last two can be verified by the digital signatures corresponding to the bitcoin keys.

The message has the following fields (shown as size_in_ bytes: field_name):

[64:node_signature_1]

[64:node_signature_2]

[64:bitcoin_signature_1]

[64:bitcoin_signature_2]

[2:len]

[len:features]

[32:chain_hash]

[8:short_channel_id]

[33:node_id_1]

[33:node_id_2]

[33:bitcoin_key_1]

[33:bitcoin_key_2]

This message is initially broadcast to all the neighbours of both the nodes operating the channel. The neighbours which receive this, verify the contents of the message. If the verification succeeds they again broadcast it to their neighbours and this process repeats. In this manner, the existence of a channel is made public to the entire network.

- **channel_update**: After a channel has been announced, now both the nodes can broadcast their channel-specific parameters. The channel_update message is adopted for that.

  The message has the following fields (shown as size_in_ bytes: field_name):

  type: 258 (channel_update)

  data:

  [64:signature]

  [32:chain_hash]

[8:short_channel_id]

[4:timestamp]

[1:message_flags]

[1:channel_flags]

[2:cltv_expiry_delta]

[8:htlc_minimum_msat]

[4:fee_base_msat]

[4:fee_proportional_millionths]

[8:htlc_maximum_msat]

This message as well is broadcast to the entire network. A node operating the channel, can update the channel parameter values by again broadcasting a new message.

## 2.5 Onion Encryption

This section explains the onion encryption scheme proposed in the BOLTS [10]. The source node after finding a viable path uses onion encryption to forward instructions to the individual hops on the path.

The nodes in the network have the information about all the public channels present in the network due to the gossip mechanism. Following are the properties of the channels that are made public to the entire network.

- Channel ID: The unique identification for every channel created.
- ChainHash: Identify the blockchain on which this channel was opened (Hash of genesis block)
- Node Keys: The public keys of both the nodes involved in channel formation.
- Bitcoin Keys: The bitcoin public keys of both the nodes involved.
- Authentication proof: Set of signatures to prove the legitimacy of the channel
- Capacity: The total capacity of the channel
- CLTV expiry delta: The time gap required between the timelocks (CLTV expiry) of the incoming and the outgoing HTLC.
- Base Fee: The minimum routing fees to forward a payment of any amount.
- Proportional Fee: The routing fees to be deducted proportional to the amount. Usually expressed in millionths of a satoshi(smallest unit of the bitcoin currency) transferred.

With this information the nodes construct a graph of channels with properties associated with it. When it wants to initiate a payment it searches for the best path available to the destination from this graph. The best path is the one which

- has the capacity to support the amount being transferred
- has the minimum fees compared to rest of the paths and satisfies the first requirement

With the knowledge of the path the source node constructs an onion packet to the destination. This scheme is built upon the Sphinx [29] cryptographic message format. This onion packet has the instructions for every intermediate hop in the route as to where and how much funds to transfer. The instructions for individual hops can only be decrypted by that specific individual hop. The hops only gets to know who their predecessor and successor are, without discovering any knowledge about the initiator or the recipient of the payment. The source node uses a specific scheme within the Elliptic-curve cryptography (ECC) to encrypt instructions for the individual hops. We first discuss the ECC scheme and then explain the onion packet construction.

### 2.5.1 Encryption/Decryption scheme

Elliptic Curve Integrated Encryption Scheme (ECIES) is one of the widely known encryption schemes based on ECC. The onion encryption scheme uses ECIES as a building block to encrypt and decrypt messages. ECIES is a framework which allows the implementer to choose different functions for its operation. The ECIES has the following functions:

- Shared secret generation (SS): A function used to generate a shared secret between the involved parties. Diffie-Hellman key agreement protocol [32] is an example of such a function.
- Key Derivation Function (KDF): Generate sets of keys for different operations, such as encryption and Message Authentication Code (MAC) generation. The input to this function would be the shared secret generated in previous stage and optional parameters. For example PBKDF2 [42] or Srypt [51].
- Encryption(ENC): A symmetric encryption algorithm such as the AES [55] to encrypt the plain text.
- Message Authentication Code (MAC): Function to generate and verify the authentication codes. For example HMAC-SHA-2 [35] is a MAC generation function.

The Figure 2.2 shows the process of encrypting a plain text and generating a MAC code for it. During decryption the process will remain similar, except, the

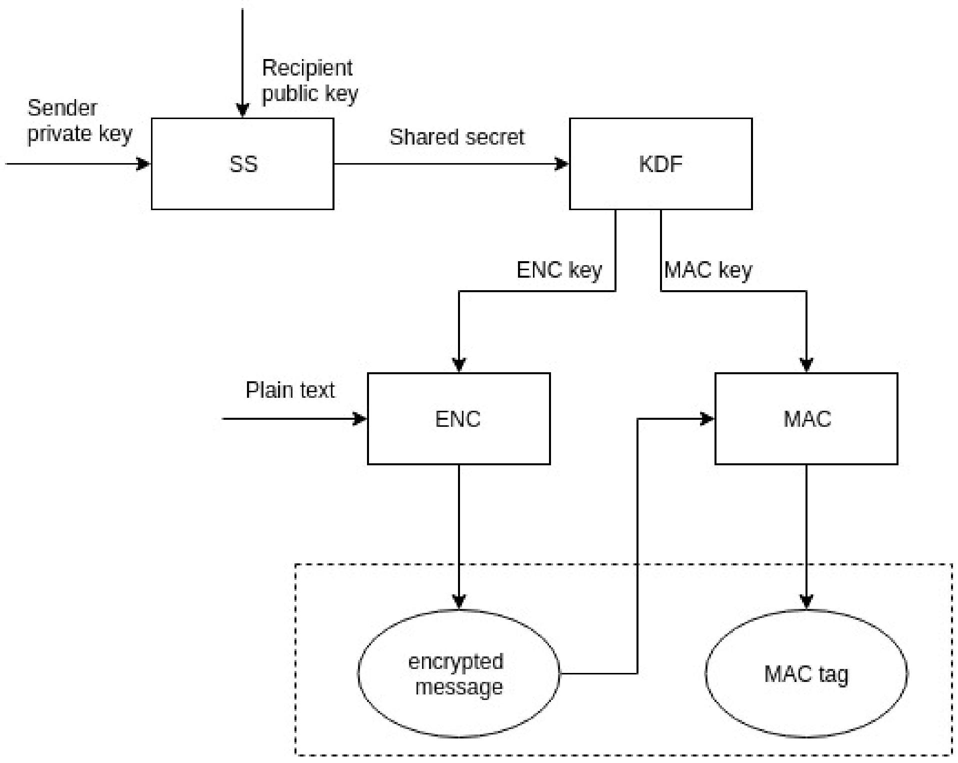

Figure 2.2: ECIES scheme encryption process

recipient will use its private key and the senders public key to generate shared secret and decrypt the message.

In the Lightning network the choices for these functions are shown in Table 2.1.

| **Function** | **Choice** |
|---|---|
| Shared secret generation | ECDH |
| KDF | HMAC-SHA 256 |
| Encryption | ChaCha20 |
| MAC | HMAC-SHA 256 |

Table 2.1: ECIES function selection

- Elliptic-curve DiffieHellman (ECDH): It is a variant of the standard Diffie-Hellman algorithm [32] for the elliptic curves. ECDH establishes a shared secret between two parties, which have an elliptic-curve (EC) key pair. If Alice and Bob want to establish a shared secret between them, Alice uses her EC private key and Bob's EC public key as an input to the ECDH algorithm. Similarly, Bob uses his EC private key and the EC public key of Alice as inputs to the ECDH. The ECDH algorithm ensures that both Alice and Bob end up with the same secret. This secret sometimes is directly used as a key, or multiple keys can be derived from it.

- HMAC-SHA 256:The keyed-hash message authentication code is used as the key derivation function and also for generating the MAC code.
- ChaCha20: It is a stream cipher [25].

### 2.5.2 Onion packet construction

The intention of the onion packet is to instruct every hop about:

- Next hop on the path
- The amount they should forward
- The CLTV expiry they must apply in the HTLC script

#### Structure of the packet

The structure of the packet has the following fields as shown in Table 2.2

| **Size(bytes)** | 1 | 33 | 20*65 | 32 |
|---|---|---|---|---|
| **Field** | Version | Public key | Hops data | HMAC |

Table 2.2: Structure of the onion packet

- A byte for the version
- Public key which is an ephemeral key generated by the source node
- Forwarding data for 20 intermediate hops. For this version the size is restricted to 20 hops and the size of the packet is fixed even though there may be less than 20 hops. This is to prevent the hops learning the length of the route.
- HMAC to verify the integrity of the packet

The structure of the *Hops* data field is as shown in Table 2.3. It has a Realm field which defines the structure of Per hop field. HMAC is again to verify the integrity.

| **Size(bytes)** | 1 | 32 | 32 |
|---|---|---|---|
| **Field** | Realm | Per hop | HMAC |

Table 2.3: Structure of Hops data

The *Per hop* field has the forwarding instructions that every individual hop must obey in the path. Its structure is as shown in Table 2.4 and has the following fields:

| **Size(bytes)** | 8 | 8 | 4 | 12 |
|---|---|---|---|---|
| **Field** | Short Channel ID | Amount to forward | Outgoing CLTV | Padding |

Table 2.4: Structure of Per hop field

- Short Channel ID: It is the outgoing short channel ID on which a node must forward the payment to. It is a unique identification for a channel. It is comprised of the block height at which the funding transaction of the channel is created, the transaction index of the funding transaction within the block and the transaction position indicating transaction output. It is strictly not necessary for a hop to use the same channel mentioned in the *Per hop* field. The hop can forward on a different channel, however, the next hop must be the same(multiple channels can be opened between two peers). If it fails to adhere to this rule and forwards to an unintended hop, that hop will not be able to decrypt the instructions in the onion packet. Eventually, the whole payment will fail.

- Amount to forward: This is the amount to forward to the hop having the Short Channel ID opened with. The node can calculate the fees it receives from this transaction by
  $Fee = Incoming\ amount - Amount\ to\ forward$
  Every hop must deduct the fee according to the advertisement made during gossip. Failing to do so would mean the destination node will receive a different amount than it is expecting. Thus, it will fail the transaction and none of the hops will receive the routing fees.

- Outgoing CLTV value: It is the CLTV value the HTLC contract must have when forwarding. The node must ensure that this value satisfies the CLTV expiry delta which it had advertised over its channels.

- Padding: This field is reserved for future use

To craft an onion packet the source node performs the following actions:

- Shared secret generation: The sender knows the public keys of all the intermediate hops, due to the gossip mechanism. The sender generates shared secrets for every hop including the destination node using ECDH. The input to the ECDH function is an ephemeral private key and the node public key of a hop. Ephemeral key pair is used instead of the sender's node keys to prevent knowing the sender's identity. The ephemeral public key will be a part of the onion message as shown in Table 2.2.

- Key Derivation: Three keys are derived from the shared secret for the following purpose.

  - To encrypt the *Per hop* information

  - Generating the HMAC's for verifying integrity of *Per hop* and overall onion packet
  - For encrypted error reporting back to the source node from an intermediate hop

  The keys are generated by calculating the HMAC-SHA256 over the shared secret.

$$Key = HMAC(shared\_secret\_value)_{key=key\_type}$$

  The keys for this operation are predefined as shown in Table 2.5(Predefining the keys will not compromise the security of the HMAC algorithm, as the input to the HMAC is a secret).

| **key_type** | **key** |
|---|---|
| Encryption (Per hop) | 0x72686F |
| MAC generation | 0x6d75 |
| Error encryption | 0x756d |

Table 2.5: Keys for different operations

- Encryption: The forwarding instructions are filled in the *Per hop* field for all the hops. They are encrypted individually with the generated encryption keys. Each field is placed in the onion packet according to the Sphinx [29] construction.

- MAC generation: The HMAC's are computed for the individual *Per hop* field and added to the *Hops data* field. Thus, completing the onion packet construction.

The intermediate hops receive the onion packet and they peel their individual onion layers to decrypt the forwarding instructions. They use their node private key and the ephemeral public key received as a part of the onion message to derive the required keys for decryption. They validate the individual values in their *Per hop* field so that there are no discrepancies and it can be fulfilled. The integrity of the message is also validated by computing the HMAC and verifying it against the received one. If all the validations are successful, they extend an HTLC contract with the next hop and forward the onion packet to it.

Finally, the onion packet reaches the recipient. The recipient also performs the same steps as the hops to decrypt the payload. It validates that it is being paid the exact amount and being extended the HTLC expiry as requested in the invoice. If validations are successful, it settles the HTLC by revealing the preimage with the previous hop. The HTLC settlement should continue until the sender, thus fulfilling the transfer.

**Analysis of the onion routing**

We analyze the onion routing scheme according to confidentiality, integrity, availability and privacy. Generally, an information security system is governed by three goals which are confidentiality, integrity and availability, often called as the CIA Triad. The Confidentiality deals with providing access to particular set of information only for authorized users. Integrity is about verifying that any data has not been tampered by an unauthorized party. The Availability is about ensuring that information is available when required by any authorized party. We also consider privacy goals to be important in systems such as the Lightning network. Privacy goals aim to protect the personal node information such as the identity of the nodes sending and receiving payments and the individual channel balances.

- Confidentiality: The forwarding instructions for a hop can only be decrypted by that individual hop only. There are two layers of message encryption done to achieve this. The first layer is described in BOLT 8 (Encrypted and Authenticated Transport), which is for secure communication between any two Lightning nodes. The second layer of encryption is applied through the onion routing process by the sender.

- Integrity: The intermediate hops can authenticate that the received onion packet is not tampered and sent by some node in the network. However, it cannot authenticate that it was sent by the source node. If the source had used its node key in the construction of onion packet instead of an ephemeral key then this scheme would have provided end to end authenticity. Although, doing so would have revealed the sender identity which is not desired. A malicious intermediate hop can generate a new ephemeral key and construct an onion packet with relevant HMAC's for its desired route and forward the packet. The hops will not be able to differentiate between the packet sent by the malicious node and the actual source node as they cannot associate the ephemeral key to any node in the network.

- Availability: Multiple nodes can collude to lock the funds of specific nodes. Due to the gossip protocol, all the nodes know the entire channel network topology. Malicious nodes can act as a sender and a receiver and transfer a payment on a particular route. Later, the malicious recipient when it receives an HTLC request from its previous node will turn unresponsive. Thus, funds will be locked until the timeouts occur, however, no funds will be lost.

- Privacy: From the onion message, the hops do not get to know who the sender and the receiver of the payment is. They also don't learn the path lengths. However, with timing analysis, the hops can estimate its distance from the recipient. The hop can calculate the difference between the time it got the onion message and the time when its HTLC was settled. By dividing this time with the average time to settle an HTLC between nodes it can approximate the hop distance with the receiver. Further, it knows the complete network topology, thus, it can pinpoint the few nodes at the discovered hop radius which should be the recipient.

### 2.5.3 Encrypting Errors

The intermediate nodes can incur different errors while forwarding the payment. It is important that this error is reported back to the source node as it can take appropriate action. The errors are also encrypted to prevent leaking any sensitive information. To send an error message the hops cannot use a similar onion packet format like the one crafted by source node. The hops do not have information about the route back to the source node. It can only send the error message to its immediate peer who had forwarded the onion packet to it. The format of the error message is as shown in Table 2.6.

| **Size(bytes)** | 32 | 2 | Failure Length | 2 | Pad length |
|---|---|---|---|---|---|
| **Field** | HMAC | Failure Length | Failure Message | Pad length | Padding |

Table 2.6: Structure of Error Message in Lightning

The keys for encrypting the message and computing HMAC are done in a similar manner as while forwarding the onion messages. Every receiving hop encrypts the message again but does not update the HMAC field. Finally, the error message reaches the sender. It knows the node public keys of all the hops. It decrypts the multiple layers of encryption until the computed HMAC matches with the received HMAC.

#### Analysis of the error encryption scheme

- Confidentiality: The error message can only be decrypted by the sender.
- Integrity: The HMAC in the message field validates the integrity of the message. Due to the use of the node keys in encrypting the message, it also authenticates that the error is sent by a particular hop.
- Privacy: It does not leak any information about the sender, receiver or the path lengths. However, probing attacks are possible to find out the balance present in the channels [41].

# Chapter 3

# Requirement Formalization

In this chapter, we understand the routing characteristics of the Lightning network and then formalize a set of requirements for routing.

## 3.1 Characteristics of the Lightning network

Routing is critical for the Lightning network to fulfill its promise of multi-hop payments. With efficient routes, most of the payments can be delivered off-chain, reducing the burden on the blockchain. Though routing is well-studied and a classic problem in the field of computer networks, there are essential differences between routing in a computer network and a payment channel network like Lightning. The job of a routing algorithm in a computer network is to route data from the source to destination. While routing so, the link bandwidth capacities do not change dynamically. Also, the link bandwidth is not regarded as private information. However, the characteristics are different for the Lightning network:

- As the payments are routed over a link(channel), the link bandwidth(channel balance) also change accordingly.The links can also become unidirectional over the time.
- The actual bandwidth over a link is considered as a private information and only the total capacity(initial funding) of the channels can be revealed.
- The nodes in the network can become online/offline anytime. New channels can be opened/closed at any moment. These activities can give rise to new and better routes or invalidate some old routes in the network.

## 3.2 Requirement for routing

Considering the characteristics of the Lightning network as discussed in Section 3.1 we summarize the following as the requirements a routing algorithm must satisfy.

- **Effective route discovery:** The algorithm must be able to find routes between nodes such that there is a high probability that the payment will be successful.
- **Efficiency:** The memory required to store any routing information, the communication and computations for finding routes must be low.
- **Scalability:** Even though the number of nodes in the network is increased(at least 100,000), the algorithm must still hold its effective route discovery and efficiency requirements.
- **Cost efficient:** Among the viable paths, the routing algorithm should be able to find the ones lower transaction fees.
- **Fee and TimeLock predictability:** The nodes should know the fees for a route beforehand making a payment. Thus, the sender must be able to decide whether it would accept the fees for that route and make a payment or look up for a different route. This also holds good for TimeLock, the sender should be able to gather the CLTV expiry for a route before initiating a payment.
- **Privacy:** The routers (intermediate nodes) should not get to know who the sender or the receiver of the payment is. The routers should also not know the actual value being transferred to the receiver.
- **Security:** Even in the case of any adversaries lying about their routing state/information the sender must be guaranteed that his funds will not be lost in such scenarios.

Some of these requirements are also discussed in various works [54, 58, 40, 56], we have summarized the most essential ones for the Lightning network here. Next, we review and analyze the generic routing algorithms for payment channel networks and determine their limitations with respect to the established requirements. If there are any inherent issues with the algorithm, we do not further discuss all the requirements for it.

# Chapter 4

# Literature review

The similarities between routing in a payment channel network and a general network, has led the current work to be based on well studied concepts. Interestingly, one of the algorithms [39] is inspired by the behaviour of insects. The Table 4.1 categorizes the algorithms according to the underlying concepts they are based on:

| **Concept/Technology** | **Algorithms** |
|---|---|
| Flow-based [22] | Spider [58], Flash [60] |
| Distributed Hash Tables [47] | Flare [54] |
| Behaviour of ants [33] | Ant [39] |
| Landmark routing [59] | SilentWhisphers [45] |
| Network Embeddings [50] | SpeedyMurmurs [56] |

Table 4.1: Underlying concepts of different payment routing algorithms

## 4.1 Spider

Spider [58] derives inspiration from the packet-switched networks for routing in payment channel networks. Its goal is to have an equal incoming and outgoing payment flows through every node, thus keeping the network balanced. The main idea is to break up the payment into transaction units and sends them across routes that also keep the channels balanced. Spider does not consider channels becoming imbalanced overtime as a separate problem but tries to solve it along with routing. It does so by giving higher preference to routes which rebalance channels and also does on-chain transactions for rebalancing. Spider also tries to introduce a congestion control algorithm which decides the rate at which the transactions units must be sent for the payments.

Evaluations [58] with a real-world dataset show that spider is effective in finding routes in the network. To do so, just like Lightning, every node requires the complete payment channel network map. Maintaining a global map of a dynamic network is not an efficient solution as the network scales. Other important analysis of the algorithms are listed below:

- The nodes also require link weights to solve the optimization problem to find routes. Revealing the link weights is considered as private information.

- As the algorithm proposes for on-chain rebalancing, latencies for such transactions will be higher.

- Spider also does not allow the routers to decide their routing incentives. This may not be acceptable for individuals who want to maximise their profits.

For all these discussed issues we do not proceed ahead with Spider.

## 4.2 Flash

Flash [60] is a flow-based algorithm which aims to find routes having the right trade-off between optimal paths and the probing overhead. The idea is to recognise the unique characteristics of transactions in a payment channel network and apply distinct routing strategies for them. The authors studied the transactions in the Bitcoin and the Ripple networks and categorised the transactions into mice and elephant payments. Mice payments are comprised of small values and the elephant payments are of huge values. Flash applies differential treatment to both of these payments. To route an elephant payment, it adopts a modified max-flow algorithm to find $k$ paths and find maximum flow through them. Next, it solves an optimization problem over the $k$ paths to minimize the routing fee. For routing the mice payments, Flash finds the top $m$ shortest paths to the receiver and starts sending the full payment along one of them. If the payment is not successful, it probes the corresponding path for capacity and sends a partial payment on it. It continues this iteration with the remaining paths until the full payment is transferred.

One of the reasons for development of payment channel networks was to support micro-payments. An on-chain micro-payment transaction would bear a high fee when compared to the transaction amount itself. Thus, in such a network having sufficient liquidity to support an elephant payment may be difficult. A user may prefer to do an on-chain transaction for an elephant payment instead. Flash provides algorithm for routing in off-chain networks based on the on-chain transaction characteristics. A study of off-chain transaction characteristics also needs to be done to understand whether it would behave the same as on-chain characteristics. The flow-based approach used to finding routes with sufficient capacity will not be scalable as it will have a high probing overhead for larger networks. Also, every node needs to maintain the complete channel network topology, which makes the solution inefficient.

## 4.3 Flare

Flare [54] uses the key ideas of the Kademilia Distributed Hash Table [47] for finding routes in payment channel networks. The algorithm proposes the nodes in the network maintain routing information of their neighbouring nodes within a certain hop distance. Also, every node connects with a set of nodes as beacons in the networks. The beacons are chosen such that they are nearer to individual nodes in terms of node addresses. After this setup, a node will have local visibility due to the information stored about the neighbours. Furthermore, they will have an extended outreach in the network because of the beacons. To find a route between two nodes, both the corresponding routing tables are combined to find an intersecting pathway. If there is no intersection, the sender will iteratively try to fetch routing information from the nodes closest to the receivers address until it can find a beacon which has a pathway to the receiver. The Figure 4.1 shows an example of the neighbouring and beacon nodes for two nodes A and B. It also shows a path from A to B (through X and Y nodes)when both of the nodes do not have any common nodes in their neighbourhoods.

The algorithm is better than in the Lightning with respect to memory usage, as only neighbourhood channel information needs to be stored. However, if a node is not present in the neighbourhood the path lengths can get longer as shown in their results [54]. This will result in routes which are costly. The algorithm does not discuss how will it adapt itself to the dynamics of a payment channel network. The authors perform simulations in a static environment where the balances in the channel do not change when a payment is executed. Thus, further evaluations are required to comment about its efficiency and scalability in a dynamic setting.

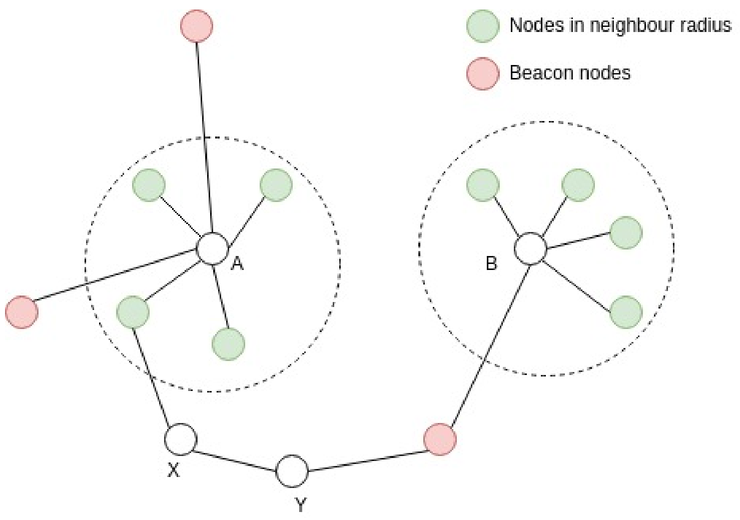

Figure 4.1: Example of neighbouring and beacon nodes in Flare

## 4.4 Ant

The authors of the Ant routing algorithm [39] derive inspiration from algorithms the ants use to coordinate with each other to find their food. The authors cite that the process of natural selection has driven to prevail only the top-performing algorithms among the ants. This was an important reason for them to study their behaviour and analyze if it applies to the payment channel networks. The main idea of their proposed algorithm to find routes is that, initially the sender and the receiver initiate a certain gossip in the network. Eventually, this gossip from both the nodes is detected by an intermediary, thus discovering a path. To begin the gossip, the sender and the receiver exchange secrets between them. Subsequently, they generate a seed from this secret termed as a pheromone seed which only differs by the starting bit between both the nodes. Both of them broadcast their respective pheromone seeds to their immediate neighbours and wait for a reply from them about the discovery of a path. The intermediate nodes receive these pheromone seeds and they check whether they also have the corresponding matching seed. If there is no match they broadcast again to their immediate neighbours. In case of a match, they communicate this information back along the respective paths until it reaches the sender and the receiver.

The Ant [39] algorithm would allow finding routes in a payment channel network. However, to achieve this, there is intensive communication in the network to propagate the pheromone seeds for every transaction. This makes the algorithm highly inefficient for larger networks. The authors also do not claim about scalability and performance of the algorithm and imply that further simulations are required to evaluate them.

## 4.5 SilentWhisphers

SilentWhisphers [45] leverages from the traditional landmark routing technique and adapts it to a distributed environment. The idea is to use a well known intermediate node called as landmarks, to find routes between any two nodes. To explain further, the information about a certain set of nodes termed as landmarks are made public to all the nodes in the network. The landmark node computes the shortest distance from itself to all the nodes, as well as, the shortest distance from all nodes to itself. A Breadth-First-Search algorithm is executed by every landmark to find these distances. Thus, the best route between two nodes in such a setting would be, the shortest path from the sender to the landmark combined with the shortest path from the landmark to the receiver. The Figure 4.2 shows an example of a discovered route (thicker connecting links) between two nodes A and B in SilentWhisphers. The algorithm further proceeds to provide a privacy-preserving method to calculate the available bandwidth in the discovered routes. It does so by designing a secure multi-party computation protocol where the landmarks act as the computation party.

As every path must traverse through a landmark, the discovered path lengths can get longer which will result in costly routes. Performance evaluation with a real-world data set also reveals that the algorithm suffers from low effectiveness in a static environment (link balance do not change) when compared rest of the algorithms [56]. Every transaction requires a multi-party computation to determine the bandwidth in a route. This is a costly operation in terms of communication and will be inefficient for larger networks.

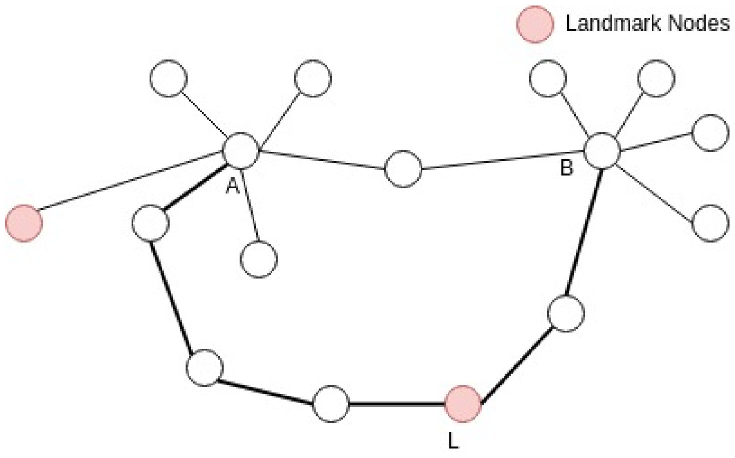

Figure 4.2: Example of a route between two nodes in SilentWhisphers

## 4.6 SpeedyMurmurs

SpeedyMurmurs [56] is an embedding-based routing protocol that tries to overcome the shortcomings in the SilentWhispers. The idea is that unique coordinates are assigned to every node in the network according to their position in a spanning tree and routing decisions are based on these coordinates. To elaborate, the nodes initially cooperate with each other to build a rooted distributed spanning tree in the network. Subsequently, they start assigning coordinates to themselves in a greedy approach. Coordinates are in the form of vectors with the root node having assigned the empty vector. Every node receives its coordinates from its parent node which will be the parent's coordinates appended with an enumeration index. The Figure 4.3 shows an example of the assignment of such network embeddings. The nodes are also aware of the coordinates of all of its neighbours. For a routing decision, a node calculates the distance between its neighbours and the destination node. It will then route the payment on the shortest path towards the destination that has sufficient balance. The links in the spanning tree are not considered during routing, which allows to find shorter routes.

Evaluations [56] of this algorithm with a real-world dataset with over 90000 nodes show that it has a success ratio of about 90% in a static setup(where link bal-

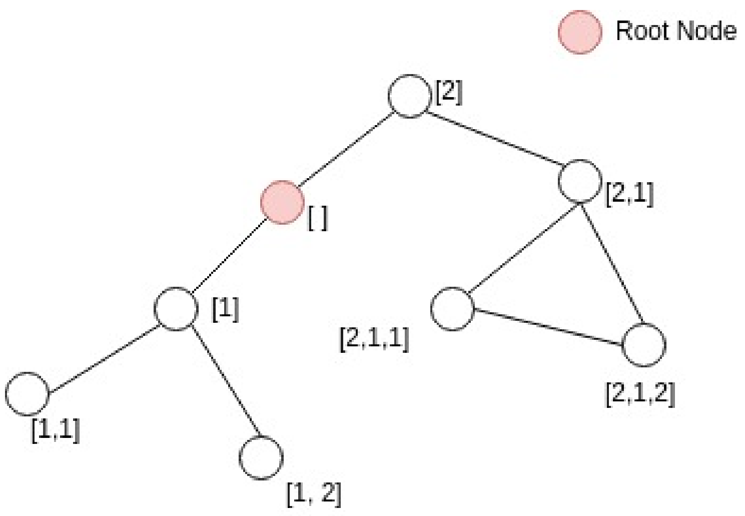

Figure 4.3: Example of the assignment of network embedding in SpeedyMurmurs

ance do not change). In a dynamic setup(link balance change according to transactions) its effectiveness remained similar to or below that of SilentWhisphers [45]. The algorithm can handle the dynamic topology changes in the network well, as it only affects a part of the spanning tree and not the whole network. Thus, in a dynamic setting, the messages required to stabilize the network are low which makes it efficient. SpeedyMurmurs strives to find the shortest path in the spanning tree which also leads to cost-efficient routes. The algorithm also provides formal privacy guarantees for value, sender and the receiver. However, it does not satisfy the fee and timelock predictability requirement as the algorithm is designed for generic payment channels and not specific to the Lightning.

Considering all the algorithms reviewed in this section, the SpeedyMurmurs is scalable, efficient, finds shorter routes and has formal privacy guarantees. Rest of the reviewed algorithms could not satisfy all of these requirements. Thus, it would be a suitable candidate to be investigated more to make it adaptable to the Lightning network.

# Chapter 5

# Routing in SpeedyMurmurs

This chapter presents the entire design of the routing algorithm. We first establish the problems we will be solving in SpeedyMurmurs. Further, we discuss in detail the design goals, the actual design, their security and privacy analysis to solve the particular problems. Finally, we also compare our solution with that of the Lightning Network.

According to the current specifications of the Lightning Network [10], every node maintains a local topology of the entire channel network through a gossip protocol. To make any payment routing decisions, the nodes refer to this local network topology. SpeedyMurmurs [56], on the other hand, does not require the nodes to learn the entire channel interconnection topology. The algorithm starts with the nodes coordinating with each other to construct a distributed spanning tree between them. Further, proceeds to establish their individual coordinates depending on their position in the spanning tree. These coordinates are broadcast to the neighbouring peers so that they are updated with the latest coordinate information. The next node on the path towards the destination is calculated based on these coordinates. Thus, the nodes running SpeedyMurmurs will hold knowledge about only its immediate peers.

Unlike source routing in the Lightning Network, the nodes running SpeedyMurmurs cannot precompute an entire path to the destination. It can only forward the payment to one of its immediate peer that will eventually lead towards the destination node. The sender also cannot calculate the Fees and CLTV over the route as it does not have the luxury of the entire network topology. However, to start forwarding the funds the sender requires this information. This leads to an interesting question with respect to the SpeedyMurmurs algorithm, which is

*How the sender can calculate the Fees and the CLTV over the entire route before initiating a payment?*

Finding a solution to this problem is critical to initiate payments.

## 5.1 Gathering Fee and CLTV

Before proposing any design solution for this problem, it is important to define the security goals. These goals will act as a guideline our design solution must obey. We will define our goals according to confidentiality, integrity, and privacy. The emulation network (explained in Chapter 6) in this thesis work is relatively small to do evaluations related to the Availability. Thus, Availability goals will not be considered in this thesis. In the later Section 5.1.4 a thorough analysis of the design according to these goals is made.

### 5.1.1 Security Goals

- **Confidentiality**: This is a query message to the intermediate nodes on the path. Only the intermediate nodes should know the contents of this query.
- **Integrity**: End to end integrity of the message is required. The source/destination node should be able to detect any inconsistent/malicious tampering of this query message.
- **Privacy**: For any payment transaction the following information must not be revealed to the intermediate nodes on the path because of our design:

  1. The sender and the receiver identity, which is the respective node public keys.
  2. The exact hop distance to the sender or the receiver.
  3. The actual invoice being transferred to the recipient.

  The following information is also desirable to remain private, as it can be used as an attack vector similar to explained in [41] to reveal undesirable information.

  1. Intermediate nodes identity: The identity, which is the public keys for the intermediate nodes should not be revealed in the route even to the sender or the receiver. The exception being the immediate peers from which the nodes receive the funds, and the peers to which they forward the funds to. The Figure 5.1 shows an example payment path, wherein the public keys of Hop2 and Hop3 must not be revealed to the sender nor the receiver.

Our intention is to uphold these security goals when designing the routing components in the Lightning network. Some of these goals are violated at different layers [41] of the protocol for which orthogonal work is being carried out.

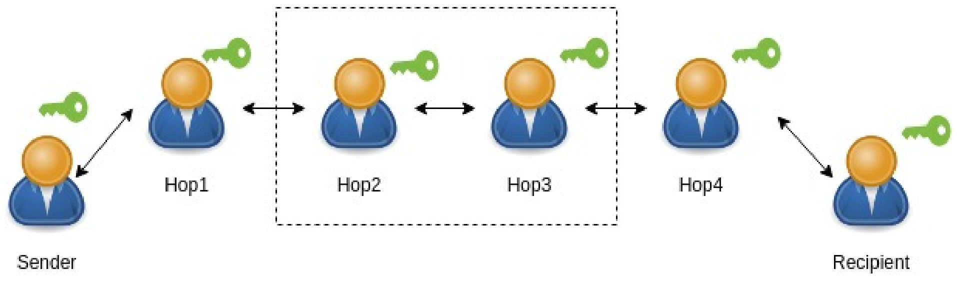

Figure 5.1: Intermediate node public keys that must be hidden from sender and the recipient

### 5.1.2 Adversary Model

We also make the following assumptions about the adversary.

1. The adversary is computationally bound. The adversary can be a node in the Lightning Network or otherwise as well.

2. It can weaken the security goals by deviating from the protocol.

3. The adversary does not have control over significant nodes in the network. It can add its own nodes in the network (bound by its computation and funding capacity), however, it cannot control/corrupt any nodes at will.

4. In this thesis, we are not considering adversaries that want to perform a Denial-of-service attack (for example flooding or replay attacks). We also overlook attacks based on network traffic analysis and timing analysis.

### 5.1.3 Design

According to our design the routing in SpeedyMurmurs must proceed in two phases:

1. **Query:** In this phase the sender will aggregate the fees and CLTV values over as many routes as it wants to split the amount into. SpeedyMurmurs can incorporate multiple spanning tree construction between the nodes. Thus, there can be multiple routes to a destination in which the payment can be split into.

2. **Forwarding:** In this phase the sender and the intermediate nodes starts forwarding the payments towards recipient based on the information collected in Query phase.

We will first elaborate the Query phase and in section 5.2 the Forwarding phase is discussed. The following descriptions will be made considering payment is made through a single path. In the case of Atomic Multi-Path payments [21], our design will still hold good, only, it needs to be applied to the involved multiple paths.

### Query phase

The main idea in the Query phase is that, the initiator will send a query packet on the route towards the destination. The intermediate nodes will update the query packet and finally return it to the initiator.

The sending of funds to a recipient begins with a request to the recipient for an invoice. The invoice has critical information for the sender as described in the Section 2.3 to initiate a payment. Along with the usual information in the invoice, now the recipient will also add the following parameters in it:

- Probe ID: A 4-byte randomly generated ID which will associate a particular payment only with this ID. A node can receive multiple payments at a time. It needs an identifier to differentiate between various payments. 4 bytes provide enough randomness to handle the number of payments in the network (For example, TCP uses 4 bytes for its sequence numbers).

- Destination Address: Every node running the SpeedyMurmurs algorithm has a unique coordinate associated with them depending on their position in the spanning tree. These addresses are only broadcast to the immediate neighbours of every node. The sender may or may not be an immediate neighbour with the recipient. Thus, the recipient needs to explicitly inform its coordinates to the sender.

We distinguish the behaviour of the algorithm based on whether a node is the sender, an intermediate hop or the recipient. They are discussed in detail below.

## Sender

The sender after receiving the invoice starts constructing the Query message as shown in Table 5.1. It performs the following actions:

- Fill the ProbeID and Destination address as it had received from the invoice. To prevent the intermediate nodes from discovering path lengths we initialize the CLTV field with 0. The Amount field cannot be initialized to 0 as every hop would be relying on this value(and Destination address) to find the next hop. If we initialize the field with the value received in the invoice, it will lead to discovering the hop distance to the receiver, violating our privacy goals (The hops can subtract the values it receives in the downstream and upstream message to estimate distance. This will be clear when explaining the behaviour of the hops). Thus, we increase the received invoice amount by a small random value and add it in the Amount field. If increased by a large value, SpeedyMurmurs may not be able to find a viable route for that large amount eventhough there existed a route to send the invoice amount. At the moment we have configured the random value to be a random number between 0 to 0.2% of the payment. The reason being, the Lightning nodes have a default fee of 0.01%, and for a route of 20 nodes, the total fee would be approximately 0.2%. Both the CLTV and the Amount field will be appropriately updated first by the recipient according to the invoice when it receives this query.

- Generate two pairs of Elliptic-curve ephemeral public-private key pairs. Add one of the public keys in the query message. Communicate the other public key directly to the recipient. These keys are a part of the design to return encrypted errors back to the source node. The necessity for generating two key pairs and the design for returning the routing errors are elaborated in the Section 5.3.
- Populate its digital signature over the SHA-256 hash of query message.
- Find the next hop according to the SpeedyMurmurs algorithm and forward this message to it.

| **Size(bytes)** | 4 | 8 | 4 | 80 | 33 | 64 |
|---|---|---|---|---|---|---|
| **Field** | ProbeID | Amount | CLTV | Destination | PubKey | Signature |

Table 5.1: Structure of Query Message to collect Fee and CLTV

The digital signature asserts the fact that the query message is being sent by a hop with whom the receiving node has an open channel with. The receiving node can verify the same as it has the knowledge of the previous hops public key. The Algorithm 1 gives the pseudocode of the operations done by the source node to send the query message.

**Algorithm 1** Query message initiation by the sender

```
1: procedure SendQueryMessage
2:     invoice = RecieveInvoice()
3:     keyPair1, keyPair2 = GenerateEphemeralKeyPairs()
4:     m = GenerateEmptyQueryMessage()
5:     m.Amount = invoice.Amount + Random(0,invoice.Amount)*0.2%
6:     m.CLTV = 0
7:     m.PubKey = keyPair1.PubKey
8:     m.ProbeID = invoice.ProbeID
9:     m.Destination = invoice.Destination
10:    m.Signature = GenerateNodeSignature(m)
11:    SendEphPubKeyToRecipient(keyPair2.PubKey)
12:    SendMessageToNextHop(m)
```

### Recipient

This query message gets forwarded by intermediate nodes and eventually reaches the recipient. A node can determine that it is the recipient by comparing the

ProbeID with the ID's of the open invoices it still has and also the destination address in the message. It then performs the following actions:

- Verify the digital signature in the message that it has been signed by an immediate peer with whom it has at least a channel open. If the verification fails discard the message and send an error according to section 5.3.

- Fill the Amount and the CLTV field as it is present in the invoice generated by itself.

- Change the public key in the query message to the one it had received from the sender node initially. The details of why this activity is important is described in section 5.3.

- Add its digital signature and return this message to the peer from which it had received this message.

The Algorithm 2 gives the pseudocode for processing the query message by a recipient.

**Algorithm 2** Query message processing by the recipient

**INPUT:**

$m$ - query message
$openInvoices$ - list of open invoice
$myAddress$ - node coordinate

```
procedure ProcesQueryMessageRecipient
    if m.Destination != myAddress then
        ProcesQueryMessageHop($m, openInvoices$)
        Exit()
    if m.ProbeID not in openInvoices then
        Exit()
    else
        invoice = the invoice corresponding to m.ProbeID
    if VerifySignature($m$) $= false$ then
        SendError()
        Exit()
    m.Amount = invoice.Amount
    m.CLTV = invoice.CLTV
    m.PubKey = invoice.PubKey
    m.Signature = GenerateNodeSignature(m)
    SendMessageToPrevHop(m)
```

### Hops

The intermediate hops receive the query message twice for a particular payment. First, during the upstream, when the message is being propagated towards the receiver. Second, during the downstream, when the message is sent back towards the sender node. The hops can detect the message is an upstream/downstream message by checking the ProbeID. The message is upstream if the node has not received a query message with this ProbeID. Otherwise, it is a downstream message if the node had earlier received an upstream message with this ProbeID. The hops perform the following actions when they receive a query message:

- Verify the digital signature in the message. Check that the digital signature belongs to the immediate hop which sent this message. If the verification fails discard the message and send an error.
- If it is an upstream message, update the signature field with its digital signature in the message and forward it to the next hop according to the SpeedyMurmurs.
- If it is a downstream message, update the Amount and CLTV field. To update the Amount, it calculates the fee required to forward the value present in the message and adds this fee to that value. CLTV is updated by adding its channel CLTV delta to the value present in the message. It updates the signature field with its digital signature and sends the message to the node from which it had received the upstream version of it.
- In the case of any other errors it discards the message and sends an error message according to section 5.3.

The Algorithm 3 gives the pseudocode for processing the query message by an intermediate hop.

The downstream message finally reaches the sender travelling hop by hop through the intermediate nodes. The sender will now know the total amount it must transfer to satisfy the invoice. This total amount will include the routing fees of the hops in the path. The sender will also get to know the CLTV expiry value to begin creating the HTLC contracts with the next node. Knowing both of this information for a particular route, the sender can decide whether it wants to go ahead with this route or query another one.

The Figure 5.2 shows an example of the above design. There are four nodes in the route including the sender and the receiver. Only, the Amount field of Table 5.1 is considered in this example for simplicity. Let us assume the sender wands to pay 5000 satoshis to the receiver. Let the intermediate nodes have uniform fees as in Table 5.2:

The sender constructs the query message, with the Amount field having the actual invoice amount (5000 here) plus a random value (8 here). The intermediate

**Algorithm 3** Query message processing by the hops

**INPUT:**

$m$ - query message

$openInvoices$ - list of open invoice

```
1:  procedure PROCESQUERYMESSAGEHOP
2:      if VERIFYSIGNATURE($m$) = $false$ then
3:          SENDERROR()
4:          EXIT()
5:      if m.ProbeID not in openInvoices then                    ▷ upstream
6:          next_node, error =GETNEXTHOP($m$)
7:          if error != null then
8:              SENDERROR()
9:              EXIT()
10:         m.Signature = GENERATENODESIGNATURE(m)
11:         SENDMESSAGEUPSTREAM($m$)
12:     else                                                     ▷ downstream
13:         if HASBANDWIDTH(m.Amount, next_node) == false then
14:             SENDERROR()
15:             EXIT()
16:         m.Amount = m.Amount + CALCULATEFEE($m.Amount, next\_node$)
17:         m.CLTV = m.CLTV + GETCLTVDELTA($next\_node$)
18:         m.Signature = GENERATENODESIGNATURE(m)
19:         SENDMESSAGEDOWNSTREAM(m)
```

| Fee Type | Fee in satoshi |
|---|---|
| Base fee | 1 |
| Proportional fee | 1 per 1000 satoshi |

Table 5.2: Routing fees of intermediate nodes

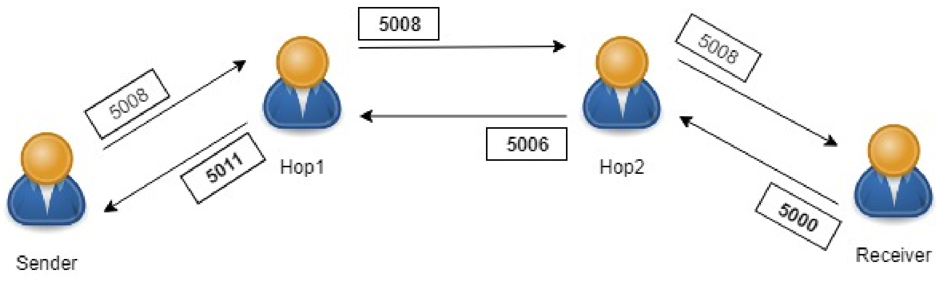

Figure 5.2: Sender acquiring the total amount to be paid on a route

nodes find the next hop according to the destination address having the link bandwidth of at least 5008 satoshis. When the query reaches the receiver, it replaces the value in the Amount field with the actual invoice amount and sends it downstream. Hop2, when it receives the query again, it calculates the fees for transferring 5000 satoshis and updates the Amount field accordingly (routing fees = 1 + 5000/1000 = 6 satoshis in this case). Hop1 also does a similar job to update the query and sending the message back upstream. Finally, the sender receives the completely updated query message which instructs it to pay 5011 satoshis to settle an invoice of 5000 satoshis.

### 5.1.4 Design analysis with respect to security goals

- **Confidentiality**: The confidentiality goal is achieved through encryption of messages before forwarding to the nodes on the path. Our design does not add an explicit layer of encryption like the onion routing in Lightning Network. We rely on the underlying encrypted transport between the nodes as described in the BOLT 8 Encrypted and Authenticated Transport [10]. This query message is propagated along the path to the destination. Thus, only the intermediate nodes on the path can decrypt this message. It is quite possible that a malicious node on the path can decrypt this message and share or distribute the contents of this message. This holds for any information system where an authorized user, if not detected as an adversary can access confidential information and play with it. Detecting such adversarial activities in the network is out of the scope of this thesis. However, the contents of the query message are such that the privacy goals which we have set cannot be weakened by such activity.

- **Integrity**: The digital signature in the message field validates that this message is sent by a node with whom it has at least an open channel with. The signature also assures the integrity of this message for individual nodes. A node with malicious intent can tamper with the contents of the message before forwarding. We analyze below the effects of tampering the individual fields:

  - Probe ID: A malicious intermediate node can update the Probe ID with some random ID. The next hops on the path cannot determine that such activity has taken place. They sincerely forward the upstream/downstream message. When this message reaches the recipient or the sender they can detect this tampering. The sender and the recipient know the Probe ID's they are involved with. Thus, when they receive a message with an unknown Probe ID they will simply discard the message and try to query a different route. With such tampering, an adversary does not gain any incentive other than denying service.
  - Amount: This field can be tampered in differnt ways, let us consider all the cases

**Random value in the amount field:** A malicious node can randomly increase or decrease the Amount field, without calculating its actual fee and propagate the message. The rest of the intermediate nodes will not be able to differentiate that such an activity has taken place. They, would honestly update the Amount field according to their channel fee rate and propagate. The source node when it receives the updated query can detect such discrepancies as it knows the actual value to be transferred. If the received Amount deviates a sane threshold value of the actual amount, it can conclude a malicious behavior by one/more intermediate nodes. Then it can drop that path and try a different route.
**Increase in the Amount field by a small value:** An intermediate node can calculate the actual fee required to transfer the received amount and bump it up with some delta value. The intermediate nodes have all the rights to do it, as they are the ones who control the routing fees through them. The source node should decide whether it agrees with this routing fees or would like to try a different route.
**Decrease in the Amount field by a small value:** An intermediate node can decrease the Amount field in the following ways:

$$Amount + fee - \delta \tag{5.1}$$

$$Amount + \delta \tag{5.2}$$

$$Amount - \delta \tag{5.3}$$

In the case of 5.1 and 5.2, if the nodes behave as per their advertisement during the actual payment routing then the payment will be successful(as the recipient will receive the exact amount as requested in the invoice). If they dont behave accordingly and deduct a higher fee then the payment will fail. The destination node, as it knows the invoice can easily detect that it is being forwarded a reduced amount which it will reject. In case 5.3, the payment is bound to fail as the destination node will not agree to any amount lesser than the invoice.
In all the cases discussed about the Amount field, an adversary cannot weaken the security goals we have held. It can only try to deny service to some of the nodes in the network.

- CLTV: The analysis for this field will remain the same as that of the Amount field.
- Destination address: An adversary can tamper the destination field in the message. It can update it with an actual coordinate of a node in the network or a random coordinate address. In either of the cases the honest hops try to forward the message towards the tampered destination node address. If the message reaches a destination, the node will

discard it straight away. The destination node can detect it as it is not aware of an open invoice with a Probe ID in the received message. It is also possible that the message will be discarded by an intermediate node itself, when it cannot find the next node on the path.

  - PubKey: A detailed analysis for this field is provided in the Section 5.3.

- **Availability**: The Availability goals are not considered to be in the scope of this thesis.

- **Privacy**: Let us recall the privacy goals we had set and assess our design according to them. We intended that the intermediate nodes must not learn the following information:

  - The sender and the receiver identity, which is the respective node public keys: In our design we never reveal the node public keys of the sender and receiver to the intermediate nodes.
  - The exact hop distance to the sender or the receiver: Preventive measures are taken so that the intermediate nodes do not discover the hop distance. While the message is travelling upstream, the CLTV fields is initialized to zero and the Amount field has a random value. The fields are correctly populated only during the downstream propagation of the message. This ensures the intermediate nodes cannot estimate hop distances by calculating the difference in the downstream and upstream message values. However, the simulation results in SpeedyMurmurs [56] show that the path lengths are usually short. This allows the intermediate nodes to estimate the sender or receiver being in a shorter hop radius, but not the exact radius.
  - The actual invoice being transferred to the recipient: If the sender uses only one path to transfer the funds the intermediate nodes can estimate the invoice being settled. However, if the sender uses Atomic multipath transactions and splits the payment disproportionately the intermediate nodes cannot estimate the invoice amount.
  - Intermediate nodes identity: We do not gather the intermediate node keys before initiating payment as done currently in the Lightning network. Our design works without knowing the intermediate node identities.

## 5.2 Forwarding payments

After the query phase, the sender will know the appropriate fee and CLTV over a route. If it is satisfied with the fees over the route, it can start forwarding the payment to the next hop. The sender and the intermediate nodes craft the message as in Table 5.3 which it will send to the next hop on the path. The message comprises of the amount a node is forwarding, the CLTV it is applying, an ephemeral public key for returning errors and the coordinates of the destination node. Simultaneously with this message the nodes also craft and send the HTLC contracts according to the description in the BOLT 3: Bitcoin Transaction and Script Formats [10].

| **Size(bytes)** | 8 | 4 | 33 | 80 | 4 | 64 |
|---|---|---|---|---|---|---|
| **Field** | Amount | CLTV | Eph PubKey | Dest Address | ProbeID | Signature |

Table 5.3: Structure of forwarding message

The intermediate nodes after receiving the message and the scripts should forward the payment to the next node after deducting the routing fee from the amount. The deduced routing fee must be the same which it had advertised during the query phase(the same holds good for CLTV as well). The hops will obtain the routing fee incentive only if the whole payment is successful. If any of the hops become greedy and deduce a higher routing fee then the recipient will detect it and discard the whole payment. Thus, to earn the routing fee the nodes have to behave honestly in the network.

### 5.2.1 Comparison with the Lightning network

The Lightning network uses the gossip protocol and the onion encryption scheme to find a route and forward funds. SpeedyMurmurs, on the other hand, does an on-demand query of a route, and then forwards payments accordingly without the onion routing.
Below we compare both these techniques according to Confidentiality, Integrity and Privacy.

- Confidentiality: In both techniques, only the intended recipient can decrypt any of the messages sent.

- Integrity: The digital signatures and HMACs in the various message fields assure the integrity of the messages to the intermediate nodes. The onion encryption has a stronger sense of an end to end integrity when compared to our design. Any tampering to the onion packet(other than the ones intended in the protocol) by an intermediate node will lead to detection by the subsequent node. In our design, such tampering will be eventually detected by the sender or the receiver. The consequences of the lowered integrity can be that, it can lead to increased network congestion due to the propagation of the tampered messages until the source or the destination.

- Privacy: The gossip protocol in the Lightning network reveals all the nodes in the network, the number of open channels they have, the capacities of each of the channel(not bandwidth) as well. In our design, we only get the channel information of the corresponding peer with whom it is open.

The size of the onion message is larger when compared to the messages used in our design. However, we have a probing overhead before forwarding payments. Thus, it would be interesting to see a comparison between the algorithms regarding the actual transaction overhead they make in a network(Results are in Chapter 7).

## 5.3 Returning Error Messages

While forwarding the payments, the nodes can face many errors such as insufficient channel balance, the next hop being inactive, insufficient fee or the amount is below a minimum transfer value. Communicating this error to the source is critical because it can take suitable actions according the received errors. As SpeedyMurmurs does not use onion routing for forwarding payments, we cannot use the same technique used in Lightning network to return errors. We need a different design that can comply with the SpeedyMurmurs. This leads us to the following question:

*How the routing nodes in the network can send errors securely back to the source node?*

Before proposing any design solutions let us again set the security goals we want to achieve. As described in section 5.1 we will segregate our goals based on Confidentiality, Integrity and Privacy.

### 5.3.1 Security Goals

- **Confidentiality:** The contents of the error message may leak undesirable private information to the intermediate nodes. For example, a node returning an error that it does not have a sufficient channel balance can reveal that nodes channel balance is below a threshold. Thus, it becomes important that only the initiator of the payment should be able to decrypt the error message and not all the hops in the path. The confidentiality goal is also in accordance with the current standards described in BOLTS [10].

- **Integrity:** Preserving the integrity of the error messages is critical. Without integrity checks an adversary can try to tamper with the message and convey fake information to the initiator. Thus, we want the initator to be able to detect any tampering of messages.

The Privacy goals remain the same as described in Section 5.1.1.

### 5.3.2 Design

As we are operating in a distributed network and there are no pre-shared secrets/keys between the nodes, we will be using public-key cryptography to achieve our goals. We will explain our design in two stages:

1. Encryption/Decryption
2. Key Distribution

**Encryption/Decryption**

For encryption and decryption of message, we will use the same scheme (Elliptic Curve Integrated Encryption Scheme (ECIES) explained in Section 2.5.1) as that being used in the Lightning network. In our design, the choices for the different functions in ECIES will be the same as that of the Lightning network as shown in Table 2.1. We do not intend to modify the encryption technique as there is no need for that.

- Shared secret generation: ECDH operation is used to generate a shared secret between the erring node(the intermediate node which has incurred some error) and the source node. The erring node requires a public key from the source as another input to the ECDH. The source should not reveal its node public key as this can lead to revealing the origin of the payment. Thus, the source node will generate an ephemeral EC key pair and communicate the respective public key to the erring node as explained later in the Key distribution phase. Similarly, the source also needs the corresponding public key from the erring node for the ECDH operation. The erring node as well should not reveal its node public key because of the privacy goals we have set. It will also generate an ephemeral EC key pair and has to communicate the public key to the source node. Thus, the sender and the erring nodes need to use their respective ephemeral private keys and the other partie's ephemeral public key as an input for the ECDH. The format of the error message is shown in Table 5.4. The erring node starts populating the message fields by filling in its ephemeral EC public key in it.

| **Size(bytes)** | 33 | 32 | 2 | Length | 2 | Pad Length | 4 |
|---|---|---|---|---|---|---|---|
| **Field** | PubKey | HMAC | Length | Message | Pad Length | Pad | ProbeID |

Table 5.4: Structure of Error Message

- Key derivation function: HMAC-SHA 256 is the key derivation function used here. We require two keys for our operation, one for encrypting the errors and the other for generating the MAC tag. HMAC-SHA 256 takes as input the generated shared secret and a key. The keys for different operations are predefined in the BOLTS [10] and the required ones are as follows:
  - Encrypting errors: 0x756d
  - MAC generation: 0x6d75

  Thus, with the KDF the erring and the source node will have two sets of keys which they can use for encryption/decryption and message authentication.

- Encryption: With the encryption key generated using KDF, the erring node encrypts the plain error message using the ChaCha20 [25] algorithm and populates it in the message field. Similarly, the source node can decrypt with the same encryption key it will generate when it receives the error message.
- Message Authentication Code: Again, HMAC-SHA 256 is chosen to derive the MAC tag for the encrypted message. The input key for the HMAC-SHA 256 is derived during the KDF stage. The erring node calculates the HMAC over the entire message and updates it in the message field. After generating the MAC tag, all the fields in the message as shown in Table 5.4 are populated and ready to be sent to the source node. When the source node receives this message it can check the message integrity as it can also derive the required MAC key for verification.

Thus, with the ECIES scheme the erring node can encrypt the errors and the source node decrypt this message. The Algorithm 4 gives the pseudocode for the operations required to generate an error message. Next, we will describe how to distribute the required keys between the intermediate nodes and the source node.

**Algorithm 4** Generate Error Message

**INPUT:**

$SourcePubKey$ - ephemeral public key received in the upstream/downstream routing message

$error$ - The error message to be sent in plain text

```
1: procedure GenerateErrorMessage
2:     keyPair = GenerateEphemeralKeyPair()
3:     secret = GenerateSharedSecret(SourcePubKey, keyPair.PrivKey)
4:     encKey, macKey = KeyDerivationFunction(secret)
5:     encMess = EncryptErrorMessage(error, encKey)
6:     macTag = GenerateHmac(encMess, macKey)
7:     m = PopulateErrorMessage(encMess, macTag, keyPair.PubKey)
         return m
```

## Key Distribution

The error encryption/decryption operation begins with the shared secret generation between the erring node and the source node. To generate the shared secret either party require the corresponding public keys of the each other. As explained earlier these keys will be ephemeral public keys generated by both the nodes. In the Lightning network, the source node knows the node public key of all the intermediate nodes, thanks to the gossip protocol. Later, the source communicates its ephemeral public key in the onion message to the intermediate nodes. This is how the key distribution problem is solved in the Lightning network. The same solution cannot be applied in SpeedyMurmurs because:

- No gossip protocol to collect the public keys of all nodes in the network.
- No onion routing for forwarding payments.

Thus, in SpeedyMurmurs we need a design to solve the key distribution problem. Following are the problems we want to solve:

1. Communicate an ephemeral public key from the source node to the intermediate nodes.
2. Communicate an ephemeral public key from the erring node to the source.

We are designing this solution to return errors from intermediate nodes if any during two occasions. It is expected that our design will be consistent during both these following occasions:

1. Gather Fee and CLTV: Here, the query message traverses first from the source to destination (upstream), and then back to the source (downstream) as explained in Section 5.1.
2. Forwarding payments: The forwarding message only traverses from the source to the destination(upstream) as explained in Section 5.2.

A simple solution as shown in the Figure 5.3 would be that the source node will embed its ephemeral public key(shown in green colour) in the upstream message. This message would travel hop by hop to all the intermediate nodes. The erring node would use this received public key for encryption of error message. Then along with the encrypted message the erring node would have sent its ephemeral public key(shown in blue colour) back to the source node.

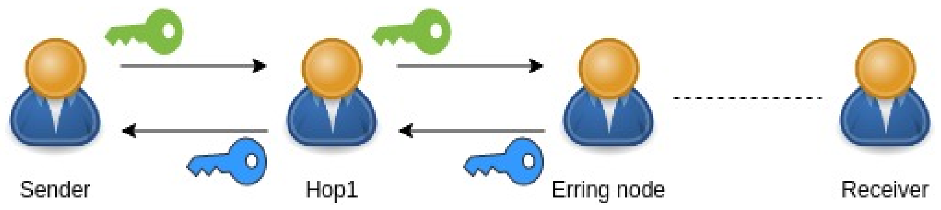

Figure 5.3: Key distribution in a non-adversarial scenario

This solution would cleanly work in a non-adversarial scenario. However, if there are nodes with malicious intents they can easily launch a classic Man-in-the-middle attack as shown in Figure 5.4. An intermediate adversary can replace the key with its own public key(shown in red colour) and forward the message towards the destination. The erring node cannot detect such tampering and would encrypt the message using the adversarial public key. The adversary now can read the contents of the error message sent by the erring node and also can forward

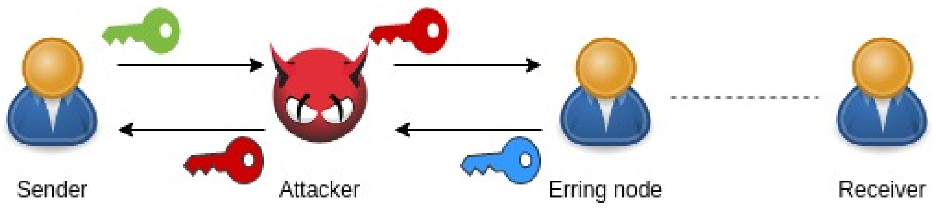

Figure 5.4: Key distribution in an adversarial scenario

a fake encrypted message back to the source node. This will clearly violate our confidentiality and integrity goals.

The problem with the simple solution was that we allowed both the public keys (from source to erring and from erring to source) to traverse through the same set of intermediaries. Thus, an adversary gets an opportunity to replace the keys in both the direction and remain undetected. Our solution will improve if we do not traverse both the keys on the same set of intermediaries and use a different route(assuming only one adversary present in a route). Let us consider both the instances of message traversal and see how we can use different routes to send the keys.

- **Upstream:** The Figure 5.5 shows the process that will be explained. Upstream is when a message traverses from the source to the destination. This can occur during the fee gathering phase or while forwarding payments as explained in Sections 5.1 and 5.2 respectively. In both of these messages as shown in Table 5.1 and Table 5.3 the source node will add its ephemeral public key (green colour). The erring node uses this key in the message for encrypting its error. Now, instead of sending the error message(which includes the ephemeral public key (blue colour) of erring node as shown in Table 5.4 ) back on the same route, we can utilise the other contents of the forwarded message to dispatch the error along a different path. Both the upstream messages as in Table 5.1 and Table 5.3 have a coordinate address of the destination node. Thus, the erring node can send the error message towards this destination address instead towards the source. The path to the destination from the erring node can never have an intersection with the intermediate nodes that had already received the upstream message. This property is guaranteed due to the underlying spanning tree construction in the SpeedyMurmurs. The destination node when it receives the error message can send it to the source node which can decrypt the message and also validate its integrity.

- **Downstream:** The Figure 5.6 shows the key distribution process during the downstream. Downstream is when a message traverses from the destination back to the source node. This can occur during the fee gathering phase as explained in section 5.1. In this case, the erring node cannot send the error message towards the destination as done during the upstream case. It also

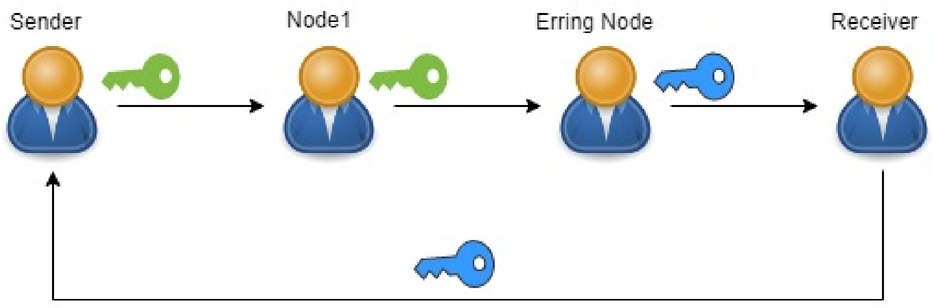

Figure 5.5: Ephemeral key distribution during the upstream traversal of message

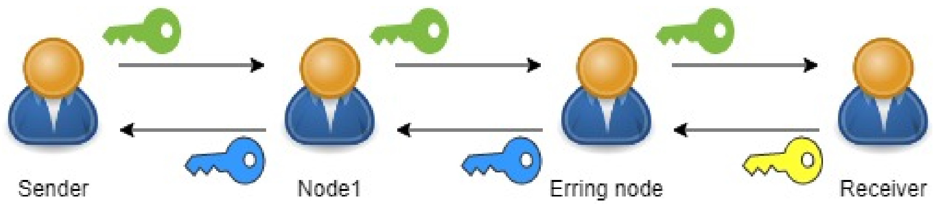

Figure 5.6: Ephemeral key distribution during the downstream traversal of message

cannot send towards the source node as the adversaries between source and the erring node could have replaced the public key sent by the source node during the upstream message. This problem can be solved if we send a different public key during the downstream and the upstream message to the intermediate nodes. Before initiating the query request the source node shares an ephemeral public key with the destination node. The destination node when it receives an upstream message, updates the message with a new public key (yellow colour) it would have received from the source node. The erring node uses the updated key to encrypt the error and send its own ephemeral key (blue colour) back towards the source. The adversaries (if any) between the source and the erring node will not be able to decrypt the message even though they have replaced the public keys during the upstream message.

In the above design it can be seen that the cooperation of the intermediate nodes is required in forwarding the error message. The incentive for them is that their cooperation helps to improve the overall payment effectiveness of the network.

### 5.3.3 Design analysis with respect to security goals

- Confidentiality: The error message sent by the erring node can only be decrypted by the source node. Only the source node has the corresponding private keys for the public key with which the message was encrypted. Even in case of a Man-in-the-middle attack the source node can detect such tampering of the message. However, the adversaries can succeed in decryption of the message if multiple of them collude on the path (at least one adversary must be before and after the erring node). The adversary before the erring node would change the public key in the upstream message. When the erring node forwards the error towards destination, another adversary can hand over this message back to the former one.

Though such a coordinated attack is presumable, the possibility of such an attack depends on whether the adversaries can be on the same payment path. This in turn depends on:

  - Position of the adversaries in the spanning tree of the network
  - The position of the source and the destination for a particular payment
  - The invoice amount of the payment

Thus, the adversaries have to plan their position in the spanning tree well and hope that they will be included in the payment path. Planning their position in the spanning tree is also not easy, as the nodes don't have access to a global map of the spanning tree.

- Integrity: Detecting tampered messages is important as it helps prevent undesirable situations to the source node. The source node can validate the integrity of the error message by calculating its HMAC and checking it against the received HMAC in the message. If they do not match it can discard the error message. If there is only one adversary on the path, tampering of any fields in the error message can be detected by validating HMAC. If multiple adversaries coordinate they can successfully tamper the message without the source node detecting it. However, as explained earlier it is not simple for adversaries to be on the same payment path.

- **Privacy**: Let us again recall the privacy goals we had set and assess our design according to them. We intended that the intermediate nodes must not learn the following information:

  - The sender and the receiver identity, which is the respective node public keys: In our design we never reveal the node public keys of the sender and receiver to the intermediate nodes.
  - The exact hop distance to the sender or the receiver and the actual invoice being transferred to the recipient: No field in the error message can help to discover both these information.
  - Intermediate nodes identity: We generate ephemeral public keys to encrypt errors to prevent revealing intermediate node identities.

### 5.3.4 Comparison with Lightning network

Again, we compare our design with that of the Lightning network according to Confidentiality, Integrity and Privacy:

- Confidentiality: In both the designs only the source node can decrypt the error message. However, multiple nodes can collude to weaken this guarantee in our design(however, it is not easy as explained during the analysis). Such an attack will not succeed with the onion routing.
- Integrity: The source node can validate the integrity of the error message in both the designs. The same drawback exists in our design as discussed in Confidentiality if multiple nodes collude.
- Privacy: In the Lightning network, the source node gets to know who sent the error message. This has led to probing attacks [41] to discover channel balance of a node. Our design does not reveal the identity of the erring node, thus such probing attacks are not possible.

## 5.4 Root Election and Spanning Tree construction

The main components in the SpeedyMurmurs can be divided into the following:

1. Root Election: The elected node will be the root node in the spanning tree.
2. Spanning Tree construction: A distributed spanning tree that will be built across all the nodes that have open channels.
3. Coordinate gossip: To establish their individual coordinates, the nodes send and receive their tentative coordinates with their neighbours.
4. Query phase: To query a route for knowing the fees and the CLTV to be applied.
5. Payment forwarding: The Speedymurmurs decides which is the next hop a node should forward payments to.
6. Returning errors: Mechanism to return encrypted errors during the Query or Payment forwarding phase, back to the source node.

We have discussed in detail the design of the last three components in the earlier sections of this chapter. The root election and the spanning tree algorithm are also the core components of the Speedymurmurs. In this thesis, we have not considered deliberating over the design of these algorithms, to meet the same security and privacy goals as that in Section 5.1.1. Such a design requires a separate thesis of its own. As all the components need to be implemented to evaluate SpeedyMurmurs, we only consider scalability as the factor when choosing the Spanning tree protocol. The Radia Perlman spanning tree protocol [52] has been a fundamental block in the modern Ethernet. The algorithm transformed the original limited-scalable Ethernet to a protocol that can manage massive networks today. It allows to construct a rooted spanning tree in any graphs, thus, it satisfies the requirements for 1st and 2nd components which we need to implement. The next section explains the Radia Perlman algorithm to construct a rooted spanning tree and how we adapt it to according to the Lightning network.

### 5.4.1 Spanning tree algorithm

The algorithm was initially developed for the bridges in a Local area network to prevent broadcast storms. In the original algorithm, every bridge competes to become the Root Bridge (RB) and then tries to find the shortest paths to RB from all its ports. The RB in the network will be the one with the numerically lowest MAC address. For all the operations required the bridges communicate among themselves with messages called as Bridge Protocol Data Units (BPDU). The BPDU comprises mainly of the Bridge ID (Bridge priority + MAC address), Root Bridge ID (The bridge which it thinks is the Root Bridge), Cost to Root (Distance or Cost to the Root Bridge). Initially, every Bridge initializes itself as the RB and cost to Root as 0 and participates in the algorithm. Every bridge keeps sending this BPDU messages to its connected bridges at regular interval of times. They process these received BPDU's periodically such that, if it receives a better BPDU (which has a lower Root ID or a lower cost to Root) it updates its Root according to this BPDU and propagates this information down to its other connected entities. The port on which the Bridge receives the best BPDU (the best path to the Root) will be labelled as the Root port. The port which is the best path for a LAN segment to reach the RB will be labelled as the designated port. This is determined by if the port has sent out BPDU along it and has not received any better BPDU on that port. The ports which are neither Root ports or Designated ports will be Blocked ports. Blocked ports break the loops in the network as no traffic will be sent or received along them. Finally, the RB will be the one having all its ports labelled as the Designated ports. If a new node enters the system, it will start itself as RB and again there will be an exchange of BPDU's in the network.

The explained algorithm is adapted and implemented for the Lightning network. The algorithm remains the same only some terminologies will be adjusted for the distributed network.

- Bridges -> Individual nodes in the Lightning network
- Bridge ID -> Node ID. (Generated by SHA-256 hash of the public key of the Lightning nodes.)
- Cost to Root -> Number of hops required to reach the Root node.
- BPDU -> Hello Message. The Table 5.5 shows the fields in the Hello message.

| **Size(bytes)** | 4 | 4 | 4 |
|---|---|---|---|
| **Field** | Node ID | Root Node ID | Cost to Root |

Table 5.5: Structure of spanning tree Hello message

When a node opens a new channel with another node, or rejoins the network after being offline it broadcasts the Hello messages to its peers. Processing of these Hello messages is done at regular intervals of time(currently 5 seconds) and the best path found is advertised to the other connected peers. While processing these Hello messages, the paths originating from the node are marked as Root, Designated or Blocked according to the algorithm. When there is no more exchange of Hello Messages in the network, it signals that the Spanning tree construction is completed. The final elected Root node will be the one, which has all its paths/ports labelled as Designated.

### 5.4.2 Coordinate gossip

The nodes advertise their coordinates to their peers in the following two cases:

1. When they join the network by opening a channel, announcing themselves as the Root node.
2. Once their coordinate address is updated when they find a better path to the Root node.

The Table 5.6 shows the structure of the gossip message the nodes exchange.

| **Size(bytes)** | 4 | 80 |
|---|---|---|
| **Field** | Node ID | Coordinate Address |

Table 5.6: Structure of the coordinate gossip messages

These messages are also sent and processed in periodic intervals of time(5 seconds). With the termination of the spanning tree construction, the coordinate gossip will also terminate as nodes will be sure of the Root port node.

To conclude, this chapter gave an elaborate design of each and every component of SpeedyMurmurs. In the next chapter we explore a system to implement and evaluate our design.

# Chapter 6

# System Setup

Having proposed the design in Chapter 5, we want to evaluate the performance of the routing algorithm according to different metrics. The aim is to emulate the Lightning network with many interacting nodes. Subsequently, execute payments to evaluate the routing algorithm. The mainnet and the testnet of the Lightning network cannot be used for obvious reasons to run our tests. Thus, we need to explore the different solutions offered by the Lighting network implementations to execute tests as per our aim. Also, we need to decide the exact Lightning network implementation we would implement the routing algorithm in. In this chapter, we explore and finalise a suitable system to run our evaluations.

We require that the emulation system will at least fulfill the following requirements:

- The Lightning and the Bitcoin nodes follow the standard set of rules as described in their respective white papers [53, 49]. As, we want to emulate the Lightning network.

- The Lightning nodes should support opening and closing of channels. They should allow single and multi-hop payments.

- Funding to any bitcoin addresses should be rather quick and computationally inexpensive. This saves us time and the energy that would require for mining the Bitcoin blocks for running our tests.

To establish such a network for our study, we first describe the general architecture of the different software components the Lightning network interacts with. Later, analyze each individual components and understand which solutions suit our requirements. Finally propose the network architecture on which we will do our evaluation study.

## 6.1 General Network Architecture

Figure 6.1 shows the architecture of the different components the Lightning Node interacts with. Generally three components interact with each other on one host:

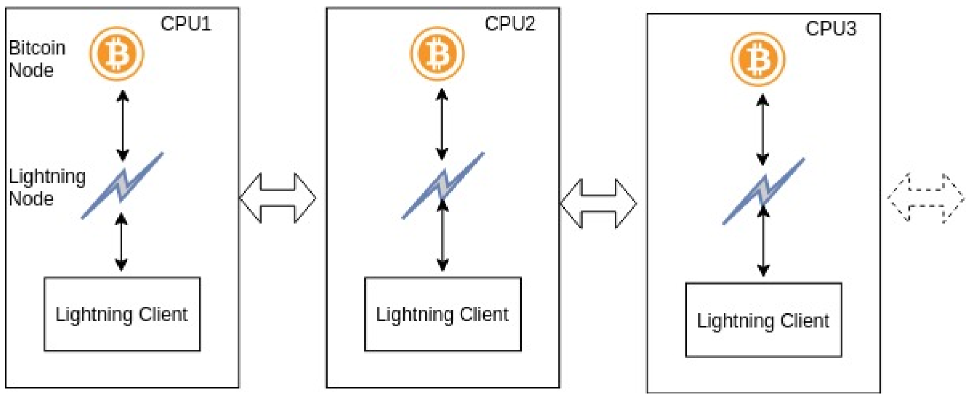

Figure 6.1: Architecture of components interacting in Lightning

- Lightning Node: Implements the Lightning protocol [53] and participates in the peer-to-peer Lightning network.
- Bitcoin Full Node: A full node can verify all the the activities happening on the blockchain according to the Bitcoin rules. The Lightning nodes require access to the verified blockchain data to manage its channel states.
- Lightning client: An interface for interacting with the Lightning node for activities such as sending payments and accessing node channel balances.

There may be variations of this architecture, such as multiple Lightning nodes connecting to the same Bitcoin node, but this holds good in general. Each of these components has different implementations providing various features and capabilities. We analyse each of these and select the ones that match our requirements.

### 6.1.1 Lightning Node

We are designing an algorithm for the Lightning network. It makes sense that the algorithm is evaluated by implementing it in one of the Lightning codebases, instead of simulating it. The Table 6.1 shows the active implementations of the Lightning network maintained by different entities. Four of them comply with the BOLT [10] standards and one do not.

After a brief study of all these implementations, we chose LND [9] as the most suitable one to implement and evaluate our design solution. Following are the reasons why LND was chosen:

| Implementation Name | Developers | Language | Bolt Compliance |
|---|---|---|---|
| LND [9] | Lightning Labs | Golang | Yes |
| C-Lightning [4] | Blockstream | C | Yes |
| Eclair [6] | ACINQ | Scala | Yes |
| Ptarmigan [16] | Nayuta | C++ | Yes |
| LIT [12] | MIT-DCI | Golang | No |

Table 6.1: Existing Implementations

- The LND provides a local simulation framework, wherein as many LND nodes can be spawned on a local machine. The spawned nodes can interact with each other in the localhost to form the payment channel network. Simulating the lightning network on a local setup is crucial as it is not feasible to test it on the main network.
- LND has the highest adoption rate when counted the number of nodes running the the software in the main network.
- Has a docker support, which can be used to spawn a required number of nodes during testing and evaluation.
- It has a responsive developer community that can be reached out for queries.
- The code has decent comments for understanding. Initial documentation is available to get acquainted with the testing environment.

**Lightning Network Daemon (LND)**

The LND complies with the BOLT standards, and their developers(Lightning Labs [8]) are active contributors to the BOLT's. LND can interact with the Lightning network that is deployed over the Bitcoin or the Litecoin [13] network. LND requires a backend full node to be always connected to manage its channels. For Bitcoin, it can connect with the BITCOIND [1] or BTCD [3] full nodes. It can also connect to a light client called Neutrino [15]. The LND in its current development state has the following capabilities:

1. Create and close channels between nodes
2. Manage all the channel states
3. Maintain a fully authenticated and validated network channel topology
4. Finding paths within the network to route payments.
5. Sending onion-encrypted payments over the network, and forwarding incoming payments

6. (Autopilot) Can automatically create and manage channels.

7. Provides an experimental feature to connect with a Bitcoin light client (Neutrino). It helps clients to reduce bandwidth and storage required for running a Bitcoin node.

SpeedyMurmurs can be developed and integrated with the first two capabilities to send multi-hop payments. The next three (3-5) capabilities will help us evaluate the existing source routing algorithm.

### 6.1.2 Bitcoin Full Node

The Lightning nodes need access to the activities happening on the blockchain to manage their channels. The full nodes provides access to the validated blockchain information to the Lightning nodes. For our study we need to open channels between Lightning nodes by executing an on-chain funding transaction initially. This means the Bitcoin addresses corresponding to the Lightning nodes must have some token balance to fund them. The Bitcoin addresses can be funded in the following ways:

1. The tokens are sent to that address on-chain from some other address.

2. The full node mines Bitcoin and the reward payout is made to that address

The first activity takes time and the second one requires a lot of resources to accomplish. These are the problems we are trying to avoid in the off-chain solutions. The full node implementations provide a regression testing(regtest) mode to save the developers time and resources for such activities. In a regtest test mode, the mining difficulty is drastically reduced so that the bitcoin node can instantly mine and create new blocks. Additionally, the full node will not interact with any other peers and creates and maintains a private blockchain on a single computer. Thus, the full node need not sync with the global blockchain saving us time, memory and bandwidth. The regtest mode is suitable for our study as we can instantly make payouts to the bitcoin addresses by mining blocks. Both BTCD [1] and BITCOIND [3] support the regtest mode. We choose BTCD as the backend full node, however, BITCOIND could have been adopted as well.

### 6.1.3 Lightning Client

The client software allows users to interact with their Lightning nodes. Every Lightning implementation is shipped with a client software. LND also has a command line utility for the same. The LND also allows to directly interact with it using Remote Procedure Calls(RPC). A structured suite of gRPC [7] API's are defined [14] for developers to build their own clients. Once we implement our routing algorithm in the LND, we need to change payment related gRPC services to adapt accordingly. Thus, we will develop a minimal client of our own which can interact via gRPC with the LND. This will allow us to have a fine-grain control over the emulation environment.

As a conclusion the Table 6.2 presents the implementations and their respective modes we will setup in our study:

| | Implementation | Mode |
|---|---|---|
| Lightning Node | LND | simnet |
| Bitcion Full Node | BTCD | regtest |
| Lightning Client | Write our own Client | NA |

Table 6.2: Chosen software components and their modes

## 6.2 Emulation Network Architecture

Now that the individual components are fixed we can design our emulation network. The simnet feature of the LND allows to spawn local nodes on a machine and also connect just one Bitcoin node at the backend. Thus, we can launch only one BTCD for all the LND nodes to connect with. The Lightning Client we develop is also made generic so that it can interact with all nodes at once. This gives better control to simulate different test scenarios. Our emulation network architecture is as shown in the Figure 6.2.

The proposed setup has both advantages and disadvantages:

- Advantages: Saves us time, energy and bandwidth to run our experiments. Also, better control over the setup as it is running on only one host computer.
- Disadvantages: The maximum number of LND nodes that can be spawned will be limited by the memory and the computing resources of the host computer. This problem can be solved if the Lightning nodes can run on different hosts and still be a part of the same simnet. We could not find any implementations that provided this feature.

### 6.2.1 Bootstrapping the emulation network

We want to evaluate our design by sending single/multi-hop payments over the Lightning network. Before sending any payments between the Lightning nodes we need to ensure that:

- The emulation network is connected as shown in Figure 6.2.
- The Lightning nodes have opened channels between them according to some desired test network graph.

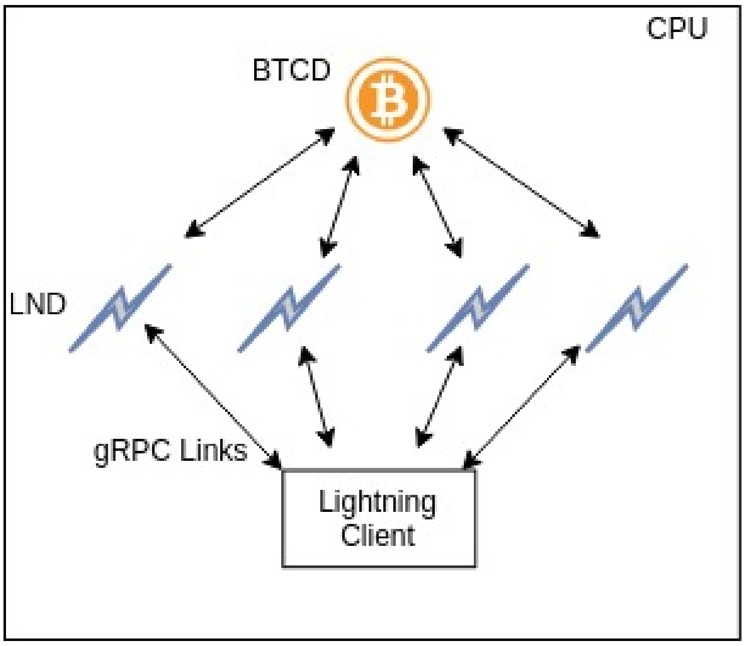

Figure 6.2: Emulation Network Setup

We describe in detail the activities performed so that our network is bootstrapped to perform payments between the nodes.

- Starting the nodes: The BTCD node is started in the regression testing mode. The required number of LND nodes are spawned in the simnet mode. The LND's are connected backend to the BTCD.
- Wallet creation: Lightning node has a wallet which manages its tokens, keys, tracks balance and signs transactions. When a node is being spawned for the first time, we need to create and initialize such a wallet for every node. gRPC services are provided to do the same.
- Payout to Bitcoin addresses: We generate new bitcoin addresses for every node. Next, we iteratively select all the bitcoin addresses to be the mining reward address and mine a few blocks. The Bitcoin balance in these addresses is necessary for funding the channels. While mining blocks, care is taken that all the LND nodes are synced with the latest mined block height, else it may lead to inconsistencies.
- Opening channels: For the required number of nodes a network graph is generated. The edges in the graph depict the channels to be opened between the nodes. Depending on edges in the graph, the channels are opened between pairs of nodes by initiating the funding transactions. Later, bitcoin blocks

are mined so that the funding transaction goes through and channel opening is complete.

The implemented Lightning client orchestrates and ensures all of these activities.

## 6.3 Integration in the LND

The SpeedyMurmurs algorithm as explained in the Chapter 5 is implemented and integrated with the various components in the LND. The routing algorithm is configured with one underlying spanning tree as LND does not support Atomic Multi-Path [21] payments yet. LND is made to use SpeedyMurmurs to route payments instead of the onion encrypted source routing.

To conclude this chapter, we have defined the emulation setup and also the involved components. We have also discussed the integrations done to the LND and also the functionalities of the Lightning client that we have implemented.

# Chapter 7

# Evaluations and Results

In the previous chapter, we described our system setup on which we will be doing our evaluations. Following up on it, in this chapter, we execute evaluation tests and present and discuss the results. We evaluate the performance of SpeedyMurmurs and also compare it with the existing source routing in the Lightning network. By evaluating the performance of a routing algorithm, its characteristics can be determined, such as the quality of service it offers, the amount of network congestion it creates and the load(communication and computation) it puts on the hosts. Various metrics are available to help us determine such characteristics. We first discuss the metrics for the evaluations, then describe the dataset and the execution environment, finally the results obtained and their discussion.

## 7.1 Performance Metrics

The performance of a routing algorithm in a payment channel network can be characterized by the following metrics:

- Success ratio: It is the percentage of payments that were successful. The payment is considered successful if the destination node receives the amount as it had requested in its invoice.
- Success volume ratio: Success volume is the total volume of the successful payments. Here, we find the ratio between the success volume of SpeedyMurmurs with that of Source routing. This metric helps to compare the success volume between both the algorithms.
- Path Lengths: The number of hops required to reach the destination in a successful payment transaction. The path lengths cannot be determined in SpeedyMurmurs, hence we add an extra field in the probe message(Table 5.1)(only for evaluation purposes)so that every hope can increment accordingly.
- Transaction delay: It is the total time taken between the initiation and termination of a payment transaction. The timer is started after the source node

receives the invoice for the transaction, and is stopped when it receives either a success or failure message from one of its peers. We only consider payments that were successful.

- Network Stabilization: It is the total number of bytes that were exchanged between all the nodes to have an updated routing state information to achieve a particular network snapshot. In SpeedyMurmurs, a node has an up to date routing state information when:

  - The distributed spanning tree is built and the algorithm has terminated(no more spanning tree messages in the network).
  - The coordinate gossip has also terminated(when no more gossip messages are transmitted related to coordinate). The nodes know individual coordinates as well as that of their neighbours.

  For SpeedyMurmurs we count the number of bytes required for the spanning tree construction(Table 5.5) and the gossip to establish node coordinates(Table 5.6). In the source routing, to have an updated routing state the nodes must have constructed the local channel network topology of the input network snapshot. We count the bytes due to the channel announcement and channel update messages of the Lightning network, which were explained in Section 2.4.2.

- Transaction overhead: The number of bytes that were exchanged between the nodes to complete a payment transaction. In SpeedyMurmurs we count the query message(Table 5.1) and the payment forwarding message(Table 5.2) exchanged. For source routing the onion messages(Table 2.2) are considered to evaluate this metric.

- Stabilization overhead: The number of bytes transmitted in the network to have an updated routing state information when a node opens/closes a channel. To measure the overhead, we first setup a network snapshot of required size. Then, 5 new nodes are made to open channels with 5 random existing nodes. We count the number of bytes that are exchanged after opening the new channels.

## 7.2 Parameters

The different parameters configured for running the tests are described below:

- Network size: We evaluate networks with size 50 and 100 nodes. We would have liked to test on larger networks, however, 100 nodes were the maximum we could execute on one server. The server environment is discussed in Section 7.3.

- Channel network topology: The Lightning channel topology in its current state resembles a hub and spoke model [57]. Simulating a hub and spoke with a maximum of 100 nodes will result in very small transaction path lengths. This will not provide enough scope to evaluate routing algorithms. Thus, we generate connected random graphs and evaluate using them. The ErdsRnyi model [34] is used to generate the graphs. The Figure 7.1 shows one such topology graph for 50 nodes. The vertices are the nodes and the edges are the channels between them. The Table 7.1 shows the number of bi-directional channels opened between the nodes in respective networks.

| **Network Size** | **Number of channels** |
|---|---|
| 50 | 73 |
| 100 | 153 |

Table 7.1: The number of nodes in the network and the number of channels opened

- Channel balance: Every channel is uniformly funded with 1000000 satoshis. This could have been any other amount as well, the value does not matter for our tests.

- Transaction amounts: The transaction amount in the Lightning network is hidden. So, there is no information about the distribution of the transaction amounts in Lightning. Thus, we generate our own dataset for testing. Four datasets are generated, each containing random amounts within varying percentages(5, 10, 25 and 50 per cent) of the funding transaction(1000000 satoshis in our case). This approach of capping the transaction amount at different percentages will provide better insights about the effectiveness of the routing algorithm, than sending random amounts without particular cappings.

- Number of payments: In total 5000 payments are executed in the network. For each payment, a random source and a destination node is chosen. By executing 5000 payments in a network of 50-100 nodes, various degrees of network channel imbalance occurs during the execution. The performance of both the algorithms during such imbalanced environments would be interesting.

The same parameters will be used for Source routing and SpeedyMurmurs while execution of tests.

## 7.3 Execution environment

The experiments were executed on the Distributed ASCI Supercomputer 4 [5] (DAS-4) systems. DAS-4 supercomputers are widely used in experimental com-

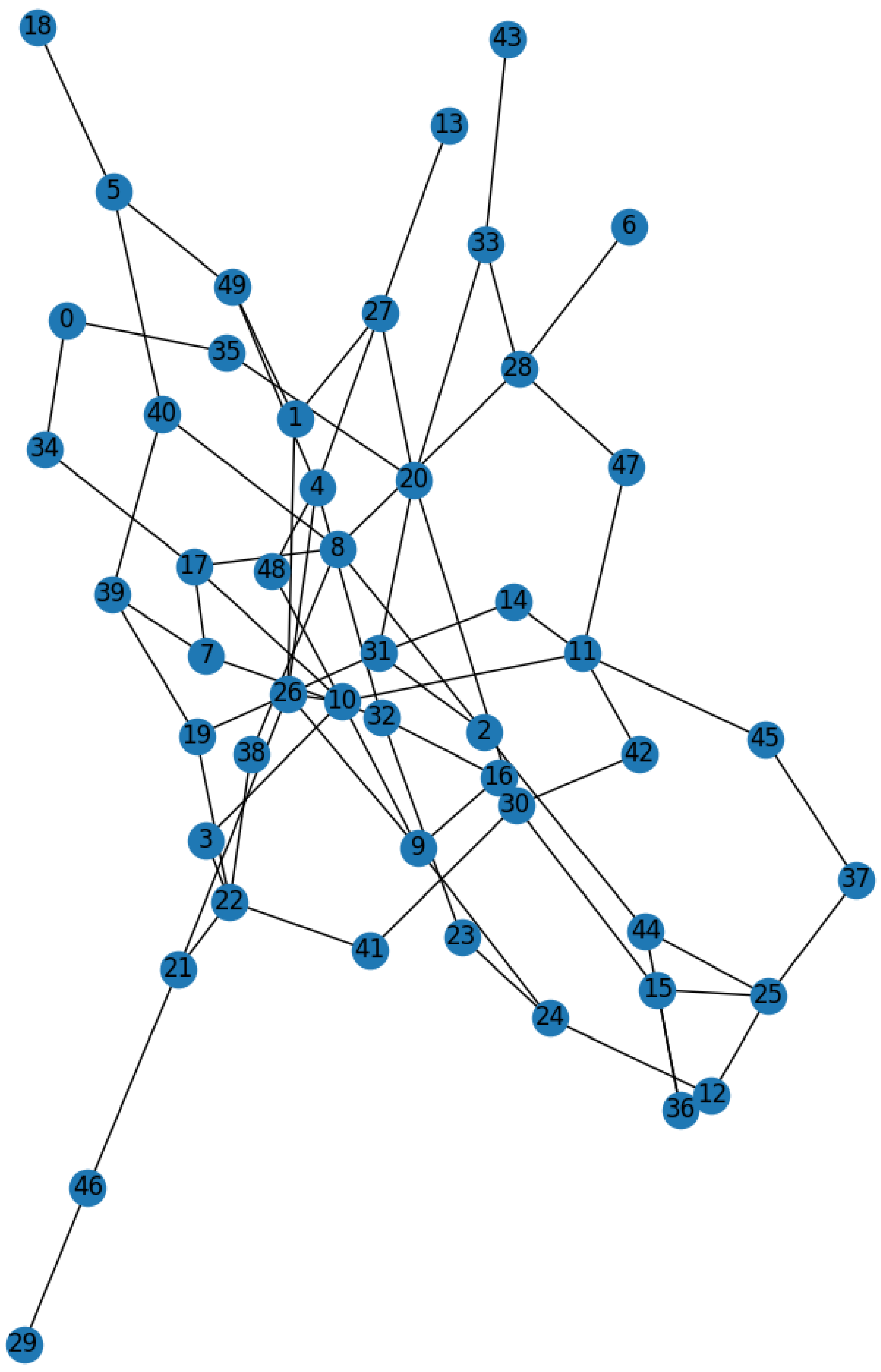

Figure 7.1: Channel topology for 50 nodes

puter science research in the Netherlands. Every node in the cluster is a dual-quad-core node with 2.4GHz speed. Different cluster sites provide access to varying memory sizes from 24GB to 48GB of RAM. The DAS-4 runs on the CentOS Linux operating system. The clustering capability within the DAS could not be harnessed as our setup can run only on one host node.

## 7.4 Results and Discussion

In this Section we present and discuss the results obtained according to the performance metrics.

### 7.4.1 Success Ratio

The Figure 7.2 shows the success ratio of SpeedyMurmurs and the Source routing algorithms. For smaller transaction amounts we observe that Source routing achieves nearly 100% success and SpeedyMurmurs perform above 85% success ratio. For larger amounts, SpeedyMurmurs dips by a maximum of about 25% compared to the Source routing. This is expected as SpeedyMurmurs incorporates a greedy approach in finding paths with enough bandwidth, which does not always provide the optimal solution. In our implementation, we have only one spanning tree configured for SpeedyMurmurs. If we construct multiple spanning trees in the network and use Atomic Multi-Path [21] to divide the transaction amount into smaller chunks and settle the payments, the success ratio then will be much higher for SpeedyMurmurs.

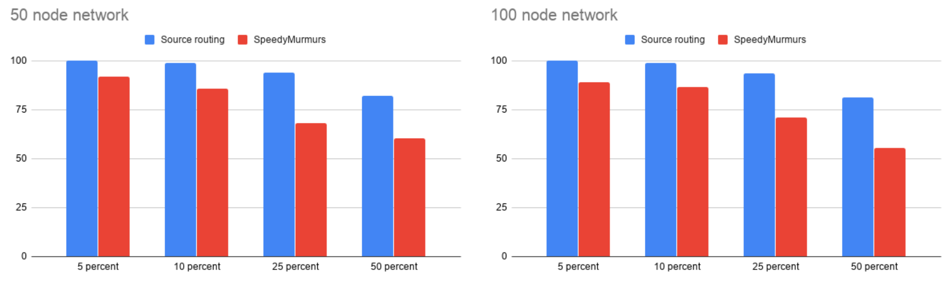

Figure 7.2: Success ratio of SpeedyMurmurs and Source routing

### 7.4.2 Success Volume

The Figure 7.3 shows the ratio of the total volume of payments that were successful between SpeedyMurmurs and Source routing. If we compare this result with the success ratio in Figure 7.2, we can derive that SpeedyMurmurs did not perform well with high-volume payments. The performance of SpeedyMurmurs will increase by configuring multiple spanning trees as explained in Section 7.4.1.

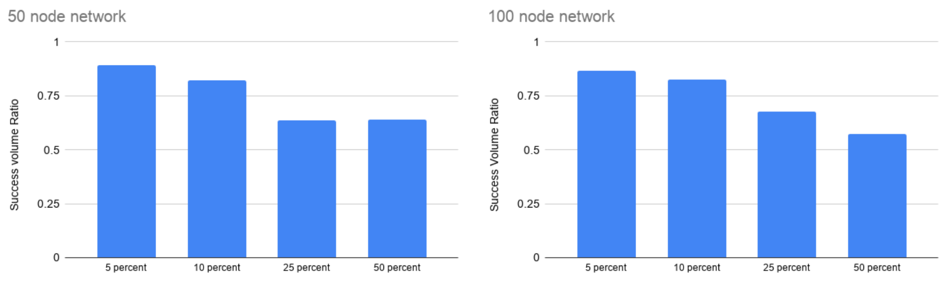

Figure 7.3: Success volume ratio between SpeedyMurmurs and Source routing

### 7.4.3 Path Length and Delays

The Figure 7.4 and 7.5 show the average path lengths and the transaction delays in both 50 and 100 node network. It can be observed that routes in SpeedyMurmurs, have on an average only about an extra hop when compared to Source routing. The exception being when the transaction amounts are up to 50%. The average transaction delays are proportional to the path lengths for smaller payments(5 and 10 %). The sudden rise in the delay and the increased path lengths at higher transaction amount for Source routing can be explained by its repeated effort of finding a valid path. The gossip protocol in the current Lightning network allows nodes to know the link capacity between all the nodes but not the bandwidth. With source routing, Lightning first tries to settle payments with a path having the least routing fees(i.e. shortest paths as fees are uniform in our simulation) based on the knowledge of link capacities. When shorter paths fail due to the non-availability of the required bandwidth Lightning tries with longer paths until it finds one or exhausts all the paths. The behaviour with SpeedyMurmurs is different. It uses greedy approach to find shortest paths with enough bandwidth. If no route is found, there are no retries to change course and find a new path. Different route explorations can be possible with SpeedyMurmurs when multiple spanning trees are configured.

### 7.4.4 Network stabilization

The Figure 7.6 shows the amount of data transferred in MegaBytes for all the nodes to have an updated state information required for the routing. SpeedyMurmurs clearly is better here, with the Source routing's data transmission at least 100 times larger than the SpeedyMurmurs. Increasing the network size from 50 to 100 nodes resulted in a two-fold increase in transmission for SpeedyMurmurs. However, for Source routing it increased nearly 5 times. These results prove that SpeedyMurmurs consumes less bandwidth and also can scale well. The results with regard to

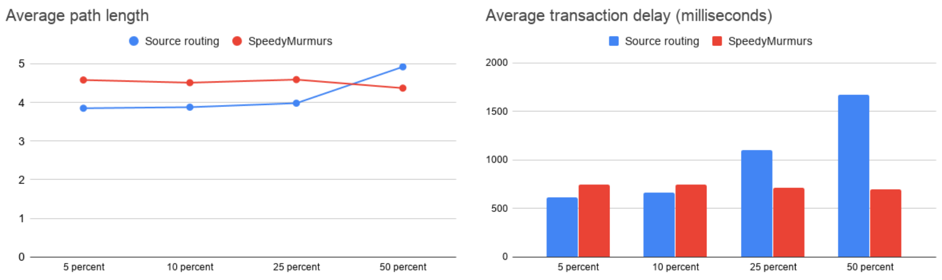

Figure 7.4: Average path length and transaction delays for 50 node network

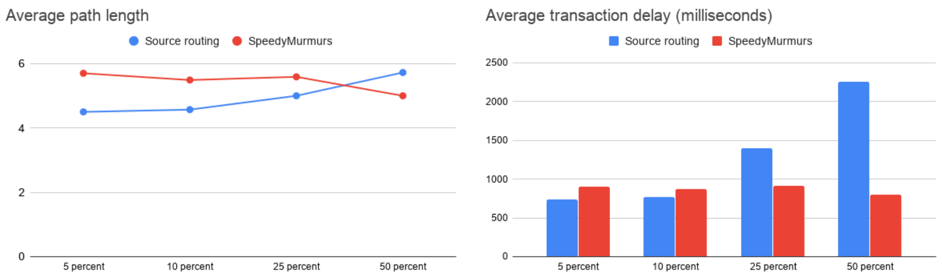

Figure 7.5: Average path length and transaction delays for 100 node network

SpeedyMurmurs will change with the introduction of a secure spanning tree protocol. The new spanning tree algorithm would have its own message formats as against in Table 5.5 that would be transmitted in the network.

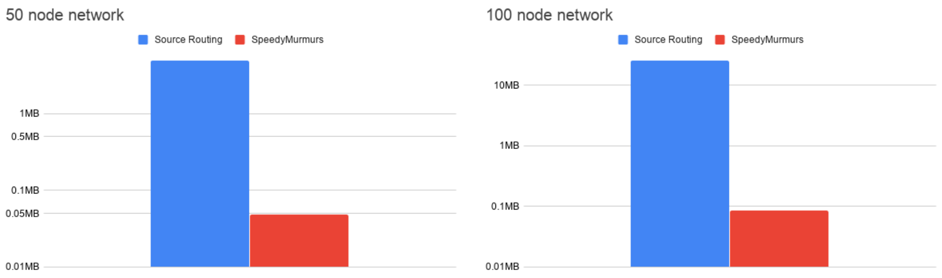

Figure 7.6: Number of MegaBytes transferred for the network stabilization

### 7.4.5 Transaction overhead

The Figure 7.7 shows the overhead in MegaBytes transferred to accomplish all the transactions in the network. We include the overhead of failed payments along with the ones being successful. The reason being, failed payments also create network congestion by either querying for routes or attempting to forward payments.

We should not disregard that data. We can see that the transaction overhead in SpeedyMurmurs is fairly constant over different transaction amounts. On the other hand, we observe in Source routing that the overhead increases with the increase in the transaction amounts. This is due to the behaviour of Source routing that tries all the viable paths if it doesn't find one as explained in the Section 7.4.3. SpeedyMurmurs, though being an on-demand route querying algorithm shows promising results when compared with Source routing. It has an overhead of at least 50% or much lesser than that of Source routing. This is due to the effect of not using the onion routing by the nodes to forward payments.

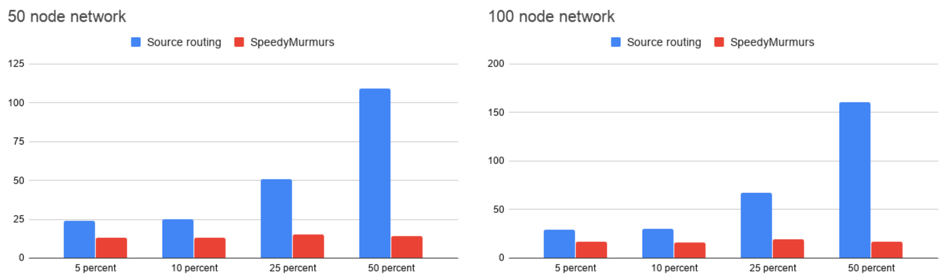

Figure 7.7: Number of MegaBytes transferred for all the transactions

### 7.4.6 Stabilization overhead

The Figure 7.8 shows the number of KiloBytes transferred in the network when 5 new links(channels) were established. It can be observed that SpeedyMurmurs outclasses the performance of Source routing. In SpeedyMurmurs, when a new node/link is added, only a sub-network is affected to update their routing states(spanning tree links, coordinates). The affected sub-network depends on the position of node/link in the network. The worst-case occurs when the entered new node becomes the root in the spanning tree. This would force the entire network to rebuild the spanning tree. Thus, while developing the root election algorithm for SpeedyMurmurs, it should strive for less new emergent leaders in the network. While in Source routing addition of a node/link affects the entire network. The node/link is advertised to every participating node, which results in a high stabilization overhead.

By analysing all the results in this section, we can conclude that SpeedyMurmurs easily outperformed source routing in the data consumption for any of the routing operations. One of the main reasons is due to its on-demand stabilization towards the network changes. This property helps to have low network congestion and allows scaling to a much larger network.

The success ratio in SpeedyMurmurs was at worst 25% lesser than source rout-

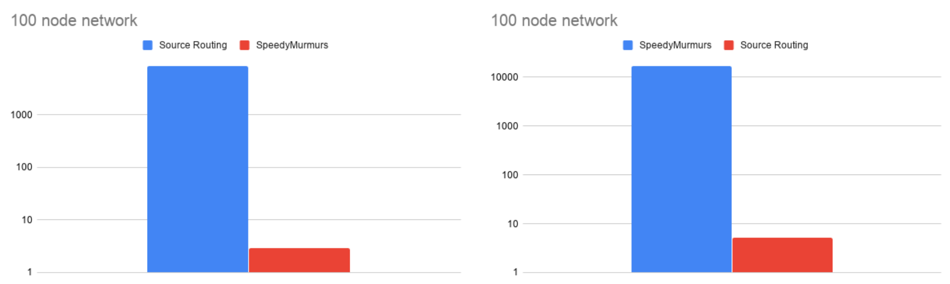

Figure 7.8: Number of KiloBytes transferred during the network stabilization

ing. This can be attributed to its greedy approach in discovering paths, which is not an optimal solution. The same tests need to be executed with multiple spanning trees configured under SpeedyMurmurs. This will enable SpeedyMurmurs to try multiple routes and also split the payment into smaller chunks. Doing so will take the success ratio higher. Also, in the real Lightning Network, a failed payment can be tried after a while. As the link balances change dynamically in the network, a previously failed route may become valid after some time. Still, if no routes are found, a flow-based algorithm such as the Ford-Fulkerson [36] can be used to discover if a route exists in the network.

# Chapter 8

# Conclusion and Future Work

## 8.1 Conclusion

Payment channel networks are an emerging field to solve the transaction throughput and latency issues in the existing blockchain systems. The Lightning Network has been one of the successful payment channel networks to operate on the main network. Routing payments is a key component in the Lightning network that needs to be handled in a proper manner to scale the network efficiently. With this premise the thesis has sought to fulfill the following research objective:

Design a payment routing algorithm for the Lightning network that builds upon key ideas in previous work, but takes the specifics of Lightning into consideration.

To answer this question, we began by leveraging SpeedyMurmurs, a generic payment routing algorithm, to make it specific according to the Lightning network. Subsequently, two important problems in SpeedyMurmurs were identified and solved. An on-demand privacy-preserving method was designed to compute the transaction fees and the CLTV expiry. Similarly, another design to send routing errors in a secure privacy-preserving manner was proposed.

To evaluate our routing algorithm, we designed an emulation network and identified all its components. The evaluations were performed according to various performance metrics. The results show that SpeedyMurmurs is highly efficient with respect to the amount of data used in the network for all its operations, compared with the source routing. Considering that only one spanning tree was configured during evaluations, SpeedyMurmurs performed decently to find viable routes to make payments. SpeedyMurmurs though incorporated an on-demand fee collection and route discovery, its transaction overhead was much lesser than that of the source routing in the Lightning network.

Thus, we were able to design, implement, test and evaluate a routing algorithm specific to the Lightning Network, which satisfies our main research objective.

## 8.2 Future Work

The following describes the related future research directions:

- Spanning tree: An efficient root election and spanning tree protocol needs to be designed for SpeedyMurmurs. The protocol also needs to adhere to the security and privacy guarantees provided by SpeedyMurmurs. It should also be tolerant towards different Byzantine behaviours exhibited by nodes while constructing the spanning tree. The authors in [27] attempt for such an algorithm, which could be a good starting point to investigate further on this topic.

- Concurrent payments: The thesis did not explore the effects of concurrency in the routing algorithm. The performance of the routing algorithm with varying degree of concurrent payments need to be studied. Currently, Lightning blocks funds until the payment is successful. There are ongoing research to have non-blocking algorithms [46] during a transaction. This will affect the performance of the routing algorithm which needs to be studied further.

- Optimising fees: Most of the Lightning nodes on the mainnet have the same routing fees at this moment. However, it cannot be guaranteed it will remain the same in the future. SpeedyMurmurs tries to finds shortest routes irrespective of the routing fees of intermediaries. Thus, with non-uniform fees it is possible that a shorter path has higher fees than the longer ones. SpeedyMurmurs need to incorporate fees as well when making routing decisions.

- Atomic Multi-path(AMP): There are proposals [21] to make Lightning network support Atomic Multi-path payments. When AMP will be introduced into the Lightning, the performance of SpeedyMurmurs also needs to be measured with varying number of spanning trees configured.

To conclude the thesis, though the Lightning Network is relatively new, the network size is growing at a fast pace . The existing source routing algorithm will turn inefficient soon at such a growth rate. This thesis provides a design of an efficient routing algorithm that could be a suitable contender to replace the source routing algorithm.

# Bibliography

[1] *Bitcoin Core*. https://github.com/bitcoin/bitcoin.

[2] *Blockchain Exploreer*. https://www.blockchain.com.

[3] *Btcd - an alternative bitcoin full node*. https://github.com/bitcoin/bitcoin.

[4] *c-lightning: A specification compliant Lightning Network implementation in C*. https://github.com/ElementsProject/lightning.

[5] *DAS-4: Distributed ASCI Suptercomputer 4*. http://www.cs.vu.nl/das4.

[6] *Eclair: A scala implementation of the Lightning Network*. https://github.com/ACINQ/eclair.

[7] *GRPC- Google Remote Proceedure Calls*. https://grpc.io/.

[8] *Lightning Labs - taking blockchains to the next layer*. https://lightning.engineering/.

[9] *Lightning Network Daemon*. https://github.com/lightningnetwork/lnd.

[10] Lightning network specifications. https://github.com/lightningnetwork/lightning-rfc. [Online; accessed 29-July-2019].

[11] *Lightning Network Statistics*. https://1ml.com/statistics.

[12] *lit - a lightning node you can run on your own*. https://github.com/mit-dci/lit.

[13] *Litecoin*. https://github.com/litecoin-project/litecoin.

[14] *LND gRPC - API Reference*. https://api.lightning.community/.

[15] *Neutrino - experimental Bitcoin light client*. https://github.com/lightninglabs/neutrino.

[16] *Ptarmigan: A C++ implementation of the Lightning Network*. https://github.com/nayutaco/ptarmigan.

[17] *The Raiden Network*. https://raiden.network/.

[18] *Statistics on Bitcoin transaction types*. https://p2sh.info.

[19] *The Tor Project*. https://www.torproject.org.

[20] *Manny Trillo. Stress Test Prepares VisaNet for the Most Wonderful Time of the Year*, 2013. https://www.visa.com/blogarchives/us/2013/10/10/stress-test-prepares-visanet-for-the-most-wonderful-time-of-the-year/index.html.

[21] Amp: Atomic multi-path payments over lightning. https://lists.linuxfoundation.org/pipermail/lightning-dev/2018-February/000993.html, 2018.

[22] Ravindra K Ahuja, Thomas L Magnanti, and James B Orlin. Network flows. 1988.

[23] Frederik Armknecht, Ghassan O Karame, Avikarsha Mandal, Franck Youssef, and Erik Zenner. Ripple: Overview and outlook. In *International Conference on Trust and Trustworthy Computing*, pages 163–180. Springer, 2015.

[24] Iddo Bentov, Charles Lee, Alex Mizrahi, and Meni Rosenfeld. Proof of activity: Extending bitcoin's proof of work via proof of stake. *IACR Cryptology ePrint Archive*, 2014:452, 2014.

[25] Daniel J Bernstein. Chacha, a variant of salsa20. In *Workshop Record of SASC*, volume 8, pages 3–5, 2008.

[26] Vitalik Buterin and Virgil Griffith. Casper the friendly finality gadget. *arXiv preprint arXiv:1710.09437*, 2017.

[27] Martin Byrenheid, Stefanie Roos, and Thorsten Strufe. On the limits of byzantine-tolerant spanning tree construction in route-restricted overlay networks. *arXiv preprint arXiv:1901.02729*, 2019.

[28] Kyle Croman, Christian Decker, Ittay Eyal, Adem Efe Gencer, Ari Juels, Ahmed Kosba, Andrew Miller, Prateek Saxena, Elaine Shi, Emin Gün Sirer, et al. On scaling decentralized blockchains. In *International Conference on Financial Cryptography and Data Security*, pages 106–125. Springer, 2016.

[29] George Danezis and Ian Goldberg. Sphinx: A compact and provably secure mix format. In *2009 30th IEEE Symposium on Security and Privacy*, pages 269–282. IEEE, 2009.

[30] Christian Decker, Rusty Russell, and Olaoluwa Osuntokun. eltoo: A simple layer2 protocol for bitcoin. *White paper: https://blockstream. com/eltoo. pdf*, 2018.

[31] Christian Decker and Roger Wattenhofer. A fast and scalable payment network with bitcoin duplex micropayment channels. In *Symposium on Self-Stabilizing Systems*, pages 3–18. Springer, 2015.

[32] Whitfield Diffie and Martin Hellman. New directions in cryptography. *IEEE transactions on Information Theory*, 22(6):644–654, 1976.

[33] Marco Dorigo and Thomas Stützle. Ant colony optimization: overview and recent advances. In *Handbook of metaheuristics*, pages 311–351. Springer, 2019.

[34] Paul Erdős and Alfréd Rényi. On the evolution of random graphs. *Publ. Math. Inst. Hung. Acad. Sci*, 5(1):17–60, 1960.

[35] NIST FIPS. 198: The keyed-hash message authentication code (hmac). *National Institute of Standards and Technology, Federal Information Processing Standards*, page 29, 2002.

[36] LR Ford and DR Fulkerson. ≪maximal flow through a network≫, canadian journal of mathematics. 1956.

[37] Adem Efe Gencer, Robbert van Renesse, and Emin Gün Sirer. Service-oriented sharding with aspen. *arXiv preprint arXiv:1611.06816*, 2016.

[38] Arthur Gervais, Ghassan O Karame, Karl Wüst, Vasileios Glykantzis, Hubert Ritzdorf, and Srdjan Capkun. On the security and performance of proof of work blockchains. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, pages 3–16. ACM, 2016.

[39] CYRIL Grunspan and RICARDO Pérez-Marco. Ant routing algorithm for the lightning network. *arXiv preprint arXiv:1807.00151*, 2018.

[40] Lewis Gudgeon, Pedro Moreno-Sanchez, Stefanie Roos, Patrick McCorry, and Arthur Gervais. Sok: Off the chain transactions. *IACR Cryptology ePrint Archive*, 2019:360, 2019.

[41] Jordi Herrera-Joancomarti, Guillermo Navarro-Arribas, Alejandro Ranchal Pedrosa, Perez-Sola Cristina, and Joaquin Garcia-Alfaro. *On the difficulty of hiding the balance of lightning network channels*. PhD thesis, Dépt. Réseaux et Service de Télécom (Institut Mines-Télécom-Télécom SudParis , 2019.

[42] Burt Kaliski. Pkcs# 5: Password-based cryptography specification version 2.0. 2000.

[43] Aggelos Kiayias, Alexander Russell, Bernardo David, and Roman Oliynykov. Ouroboros: A provably secure proof-of-stake blockchain protocol. In *Annual International Cryptology Conference*, pages 357–388. Springer, 2017.

[44] Loi Luu, Viswesh Narayanan, Chaodong Zheng, Kunal Baweja, Seth Gilbert, and Prateek Saxena. A secure sharding protocol for open blockchains. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, pages 17–30. ACM, 2016.

[45] Giulio Malavolta, Pedro Moreno-Sanchez, Aniket Kate, and Matteo Maffei. Silentwhispers: Enforcing security and privacy in decentralized credit networks. In *NDSS*, 2017.

[46] Giulio Malavolta, Pedro Moreno-Sanchez, Aniket Kate, Matteo Maffei, and Srivatsan Ravi. Concurrency and privacy with payment-channel networks. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, pages 455–471. ACM, 2017.

[47] Petar Maymounkov and David Mazieres. Kademlia: A peer-to-peer information system based on the xor metric. In *International Workshop on Peer-to-Peer Systems*, pages 53–65. Springer, 2002.

[48] Andrew Miller, Iddo Bentov, Ranjit Kumaresan, Christopher Cordi, and Patrick McCorry. Sprites and state channels: Payment networks that go faster than lightning. *arXiv preprint arXiv:1702.05812*, 2017.

[49] Satoshi Nakamoto. Bitcoin: A peer-to-peer electronic cash system. 2008.

[50] Christos H Papadimitriou and David Ratajczak. On a conjecture related to geometric routing. *Theoretical Computer Science*, 344(1):3–14, 2005.

[51] Colin Percival. Stronger key derivation via sequential memory-hard functions, 2009.

[52] Radia Perlman. An algorithm for distributed computation of a spanningtree in an extended lan. In *ACM SIGCOMM Computer Communication Review*, volume 15, pages 44–53. ACM, 1985.

[53] Joseph Poon and Thaddeus Dryja. The bitcoin lightning network: Scalable off-chain instant payments. *See https://lightning. network/lightning-network-paper. pdf*, 2016.

[54] Pavel Prihodko, Slava Zhigulin, Mykola Sahno, Aleksei Ostrovskiy, and Olaoluwa Osuntokun. Flare: An approach to routing in lightning network. *White Paper (bitfury. com/content/5-white-papers-research/whitepaper_flare_an_approach_to_routing_in_lightning_n etwork_7_7_2016. pdf)*, 2016.

[55] Vincent Rijmen and Joan Daemen. Advanced encryption standard. *Proceedings of Federal Information Processing Standards Publications, National Institute of Standards and Technology*, pages 19–22, 2001.

[56] Stefanie Roos, Pedro Moreno-Sanchez, Aniket Kate, and Ian Goldberg. Settling payments fast and private: Efficient decentralized routing for path-based transactions. *arXiv preprint arXiv:1709.05748*, 2017.

[57] István András Seres, László Gulyás, Dániel A Nagy, and Péter Burcsi. Topological analysis of bitcoin's lightning network. *arXiv preprint arXiv:1901.04972*, 2019.

[58] Vibhaalakshmi Sivaraman, Shaileshh Bojja Venkatakrishnan, Mohammad Alizadeh, Giulia Fanti, and Pramod Viswanath. Routing cryptocurrency with the spider network. *arXiv preprint arXiv:1809.05088*, 2018.

[59] Paul F Tsuchiya. The landmark hierarchy: a new hierarchy for routing in very large networks. In *ACM SIGCOMM Computer Communication Review*, volume 18, pages 35–42. ACM, 1988.

[60] Peng Wang, Hong Xu, Xin Jin, and Tao Wang. Flash: Efficient dynamic routing for offchain networks. *arXiv preprint arXiv:1902.05260*, 2019.

[61] Gavin Wood et al. Ethereum: A secure decentralised generalised transaction ledger. *Ethereum project yellow paper*, 151(2014):1–32, 2014.