2022-00

Master Thesis

Revoke and Update: A More Flexible Payment Protocol for Payment Channel Networks

Yu Shen B.Sc.

Faculty of Electrical Engineering, Mathematics and Computer Science

Delft University of Technology

Revoke and Update: A More Flexible Payment Protocol for Payment Channel Networks

Thesis

submitted in partial fulfillment of the
requirements for the degree of

Master of Science

in

Computer Engineering

by

Yu Shen B.Sc.
born in Jiangxi, China

This work was performed in:

Distributed Systems Group
Department of Software Technology
Faculty of Electrical Engineering, Mathematics and Computer Science
Delft University of Technology

Delft University of Technology

Delft University of Technology
Department of
Software Technology

The undersigned hereby certify that they have read and recommend to the Faculty of Electrical Engineering, Mathematics and Computer Science for acceptance a thesis entitled **"Revoke and Update: A More Flexible Payment Protocol for Payment Channel Networks"** by **Yu Shen B.Sc.** in partial fulfillment of the requirements for the degree of **Master of Science**.

Dated: 18.08.2022

Chairman:

Zekeriya Erkin

Supervisor:

Stefanie Roos

Abstract

Payment channel networks (PCNs) are a promising solution to the blockchain scalability problem. They move payments off-chain, i.e., not all payments have to be included in the blockchain. Thus, they do not require that every payment is broadcast to all participants and verified by them. Not requiring global consensus reduces latency, computation, and hence energy consumption.

In PCNs, a payer can route a multi-hop payment to a payee via intermediaries. Yet, Lightning, the only prominent payment channel network, has two major issues when it comes to multi-hop payments. First, the payer decides on the path without being able to take local capacity restrictions into account. Second, due to the atomicity of payments, any failure in the path causes a failure of the complete payment. While there are suggested solutions for each of these problems — namely splitting payments, local routing, and adding redundancy —, there is no solution that solves both of them.

In this work, we propose JustForward: The payer adds redundancy to the payment by initially committing to sending a higher amount than the actual payment value v. Intermediaries decide on how to forward a received payment, potentially splitting it between multiple paths. If they cannot (or do not want to) forward the complete payment value, they may reduce the amount they forward. If paths for at least v coins are found, the receiver and source jointly select the amounts of sub-payments that make up the full payment. Payment commitments are updated accordingly and fulfilled. In order to guarantee atomicity and correctness of the payment value, we use a modified Hashed Time Lock Contract (HTLC) for paying that requires both the payer and the payee to provide a secret pre-image. JustForward furthermore is the first local routing protocol to include fees: The source adds a maximal amount of fees they are willing to pay to the payment. Intermediaries take fees as they see fit, taking into consideration that subsequent nodes on the path will not forward the payment if there are insufficient fees left.

We prove that the proposed protocol achieves balance security, atomicity, bounded loss for the sender, and optionally unlinkability of split sub-payments of the same overall payments. Furthermore, our evaluation on both synthetic and real-world Lightning topologies shows JustForward outperforms existing algorithms in terms of the fraction of successful payments as long as the degree of redundancy is not too high. Concretely, for a moderate amount of redundancy, JustForward increases the success ratio of payments by about 10%.

Acknowledgments

I would like to thank my supervisors Stefanie Roos and Oguzhan Ersoy for their assistance during the writing of this thesis. I am also grateful to my parents for their support. Without these people, this would not have been possible.

Yu Shen B.Sc.
Delft, The Netherlands
18.08.2022

Contents

List of Figures

1 Introduction

In November 2021, the capacity of the cryptocurrency market hit a record high again that is almost the same as the United Kingdom's GDP in 2021. One of the most important function of cryptocurrencies is that parties can transfer money to each other. Currently, blockchain is widely used as the ledger of cryptocurrencies to record transactions [26].

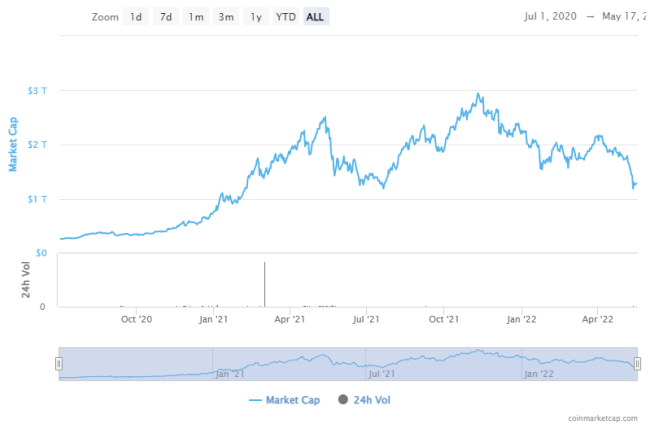

Figure 1.1: Total Cryptocurrency Market Cap [13]

In the blockchain, transactions are integrated into several blocks. These blocks are chained by their hashes: each block includes the previous block's hash inside it. In this way, the modification to a block makes all later blocks invalid. To append a new block to the blockchain, a party needs to be chosen by a consensus algorithm at first. The consensus algorithm is identical for all involved parties. So, the chosen party is the same for every party. Then, the chosen party can publish a new block on the blockchain.

The construction of blockchain largely limits the transaction speed of cryptocurrencies [17]. For example, the block size of Bitcoin is 1MB which means one block can only include 3500 transactions at most. Considering that a block is generated every 10 minutes, the maximum transaction speed of Bitcoin is about 6 transactions per second. A cryptocurrency can increase the block size to alleviate this problem. However, a larger block size means a stricter requirement on network bandwidth and storage. Different blockchain protocols like Bitcoin-ng and sharding [12], [22], [35] can also be used to increase the transaction speed. However, such solutions may be infeasible for blockchains already in use because of participants' disagreement to modify consensus mechanism. The payment channel is proposed as a solution to increase the transaction

speed without modifying the consensus of blockchain [31]. With a payment channel, two parties can make transactions without the involvement of the blockchain. They only need to commit an open channel transaction and a close channel transaction which saves time and computing power to upload every transaction on-chain.

Payment Channel Networks (PCNs) extend the payment channel to make transactions between parties in different channels possible [31]. In PCNs, payments can be routed through multiple intermediate nodes to reach the receiver. Bitcoin's PCN, the Lightning Network, uses Hash Time Locked Contracts (HTLCs) to realize the function above [31]. At first, the receiver C chooses a preimage and sends the hash of this preimage to the sender A. A establishes an HTLC with an intermediate node B using this hash. This HTLC says that A gives B some coins if B can reveal the preimage for the chosen hash in a given time. B establishes another HTLC with C using the same hash. Then, C can reveal its preimage to receive A's money and B can obtain its money from A using the revealed preimage.

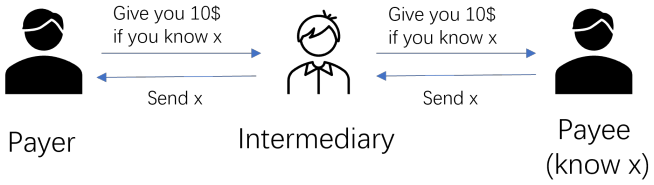

Figure 1.2: Example of HTLCs in PCN

In a PCN, a transaction can be forwarded through different paths. The choice of paths influences the performance of PCN. Hence different algorithms are proposed to achieve different performance metrics. To decrease the routing fees and delay, one implementation of the Lightning Network, Lightning Network Daemon, uses the shortest path based routing algorithm [20]. If nodes in the chosen path do not have enough funds, the success rate of payments can be limited. Routing algorithms like Flash [42] try to increase payments' success rate by implementing a distributed max-flow routing algorithm for PCNs. To have a higher success rate, Spider proposed the packet-switched routing to route payments in a similar way as data transmission [37].

Although those proposed routing algorithms are able to improve PCNs' performance to some extent, they always have some limitations because of PCNs' different properties. Another way to improve PCNs' performance is to break those limitations by refining the payment protocol. For example, Atomic Multi-Path payment (AMP) [28] protocol allows the sender to divide payment into several sub-payments and forward them through different paths. Combining with multipath routing algorithms like SilentWhispers [23], AMP is able to route larger payments with a higher success rate. Splitting locally [10] improves the idea of AMP by giving intermediate nodes the right to split payments further. It makes the routing of payments more flexible and increases the success rate of payments. Despite these protocols' high success rate, they are not resilient to depleted channels which means payments can fail because of a single depleted or unresponsive node in between. Boomerang [5] solves this problem by forwarding

payments with redundancy. With Boomerang, a payment can succeed, despite some sub-payments are failed.

1.1 Research question

Protocols like Boomerang can be used to route payments with depleted channels in between. However, such protocols largely increase the load of intermediate nodes. They are also not scalable when the amount of payment increases. We wonder how to realize the payment protocol with redundancy in a more scalable way.

Another issue of Boomerang is the fees for intermediate nodes. In Boomerang, some intermediate nodes need to lock their funds without any reward since payments they forward are redundant. Intermediate nodes may reject to use Boomerang because their benefits are damaged. We believe that it is necessary to incentive all participants who have locked funds. If all participants of a successful payment who have locked funds are rewarded, we say this payment protocol is fair to intermediate nodes.

To summarize, we come up with the following research question:
What is the method to use redundancy to increase the success rate of payments while achieving balance security, scalability, and fairness for intermediate nodes?

1.2 Contributions

In this thesis, we design JustForward, a payment protocol with redundancy. It lets intermediate nodes split payments and decrease payment size freely which increases the success rate of payment routing.

We also formalize the fairness of local routing in this thesis. At first, we point out the necessity to incentive all participants who have locked their funds. In JustForward, some sub-payments are revoked because of the redundancy in payment. To incentive intermediaries who forward revoked sub-payments, we introduce revoke fees for them. Secondly, we formalize the properties of a fair fee policy in local routing and implement Ersoy's Sybil-Proof fee policy [11] in JustForward.

To analyze the security of JustForward, We formalize the security requirements and analyze JustForward based on these requirements. We also simulate our protocol on both synthetic and real-world Lightning topologies to evaluate its performance. Simulations show that JustForward can increase the success rate of payments by about 10% when the amount of redundancy is moderate.

1.3 Outline

In Chapter 2, background about the blockchain and PCNs is introduced. Chapter 3 provides research related to our research question. In Chapter 4, we model the activities of PCNs and formalize the security requirements. In Chapter 5, we describe our payment protocol formally. The security of our protocol is discussed in Chapter

6. The evaluation setup and results are described in Chapter 7. In Chapter 8, we summarize this thesis and point out possible future works.

2 Background

In this chapter, we introduce the basics of our protocol. Blockchain, payment channels, and PCNs are presented at first. Afterwards, we introduce the routing of payments in PCNs.

2.1 Blockchain

Blockchain is the basic building block for most cryptocurrencies [45]. The data unit of the blockchain is block. A block can store transactions, scripts, or other data inside it. Most importantly, a valid block always includes the hash of the previous block. It ensures the integrity of data in blocks since any modification to a block makes its hash in the next block incorrect. In this way, blocks are chained by their hashes.

Suppose an individual participant wants to publish a transaction on-chain. It can send this transaction to IP addresses hardcoded in source code or addresses provided by some central servers. Then, this transaction is spread further until it is broadcast to the whole network. Ideally, this transaction is included in the next block and published on-chain in a few minutes.

In a permissionless blockchain, an individual participant can choose to become a miner to get rewards. A miner is a participant who tries to publish a new block by executing a consensus algorithm. Currently, multiple consensus algorithms are proposed to realize different properties like Proof-of-Work and Proof-of-Stake [19], [30]. With consensus algorithms, all miners have the possibility to publish a new block. In the Proof-of-Work algorithm, miners who solve a puzzle can publish a new block. Multiple miners may publish different blocks simultaneously. For example, the puzzle of the Proof-of-Work algorithm is solved by multiple miners simultaneously, and these miners publish their blocks separately. In this case, other miners need to choose one chain to follow and append their blocks to the chosen chain. Later, the chain with the most effort to build is recognized as the main chain.

2.2 Payment channel

Payment channel is a so-called Layer-2 protocol that allows transactions to happen outside the blockchain [16]. In the payment channel proposed by Poon and Dryja [31], two parties need to lock their funds on the chain initially. The locked funds can't be spent until the closing of this channel. During the existence of a payment channel, off-chain transactions can happen between two parties continuously. After each transaction, two parties update their balance and sign an agreement together. Such an agreement includes each party's balance, signature, and timestamp. One party can always choose to close its payment channel by publishing the latest balance

information of each party on-chain. If the published balance information is fake or outdated, the other party can obtain the locked funds of each party by publishing the correct balance information on-chain. Other types of channels are also proposed to increase the efficiency and transaction speed of the payment channel [3], [8], [25]. In this thesis, we focus on the Lightning Network's payment channel because of its simple design and popularity.

Payment channels are helpful to reduce transaction fees between two parties with frequent transactions. It also reduces the load of miners and increases the throughput of PCNs since transactions in payment channels do not involve other parties.

2.3 Payment channel network

A payment channel only allows transactions to happen between two parties. However, a party always wants to trade with arbitrary parties in the network. To realize it only with payment channels, this party needs to create payment channels with every party in the network that requires a large amount of collateral. Payment channel networks [1], [21], [31] solve this problem by connecting different channels. In a payment channel network, payment channels are considered as edges in a graph, and participants are regarded as vertices in this graph. We say two parties are connected if there is a path between them in the graph. In this network, transactions can happen between any connected parties.

To realize the described function, several multi-hop payment (MHP) protocols are proposed [4], [15], [24], [40], [41]. The most commonly used one is the Hash-Time-Locked-Contract (HTLC) [31]. If A makes an HTLC with B, it locks some coins for a fixed period and selects a hash value h. After this period, the locked funds go to B if B can provide the preimage for h. Otherwise, the locked funds return to A. With this construction, transactions can happen between parties in different channels. At first, the receiver samples a preimage and send the hash of this preimage to the sender. The sender sets an HTLC with an intermediate using the receiver's hash. This intermediate node can create another HTLC with other intermediate nodes using the same hash. Such a procedure continues until the receiver is reached. Then, the receiver can obtain coins from an intermediate node by revealing its preimage. The intermediate node that pays to the receiver can obtain coins from the previous intermediate node by revealing the same preimage. Finally, the sender is reached by an intermediate node. This node can also get coins from the sender using the revealed preimage since there is an HTLC between it and the sender.

With HTLCs, a party can only transfer funds through a single path. If there is not a path with enough capacity to carry out this payment, this payment can't succeed. To improve the MHP protocol, Lightning network extends it to an Atomic Multi-Path payment (AMP) protocol that allows a sender to forward a payment through multiple paths [28]. In AMP, the sender calculates shares of a preimage by secret sharing algorithms. These shares are used as hashes for HTLCs in different paths. If the receiver is reached in every path, the receiver learns the preimage by assembling hashes of HTLCs together. Then, it can obtain coins by revealing this preimage like the AMP.

2.4 Routing in the payment channel network

In a PCN, a path from the sender to the receiver is necessary to carry out a payment. Although we can always find a path between two connected nodes, there may be one or several parties without enough collateral in this path. Then, forwarded payments can't succeed. Thus, routing algorithms that can find out proper paths for a given payment are indispensable for PCNs.

For a routing algorithm, there are five crucial properties: effectiveness, efficiency, scalability, cost-effectiveness, and privacy [16]. Effectiveness stands for the success rate of routed payments. Efficiency is the time complexity, message complexity, and space complexity of an algorithm. Scalability means a algorithm can still be effective with a larger PCN and more concurrent transactions. Cost-effectiveness means fees of founded paths are minimized. Privacy means information of involved parties is not leaked.

Based on different forwarding strategies, routing algorithms in PCNs can be classified into two classes: source routing [39] and local routing. Source routing lets the sender decide the whole path to the receiver while each party can only decide who the next party is in local routing. In PCNs, the topology is known to every party since the open channel transactions are published on-chain. So, a sender is able to choose a path by itself in source routing. Local routing needs the help of every participant to decide the path. Thus, local routing protocols are more complicated than source routing protocols in general. However, local routing is more flexible because intermediate parties can forward payments among other channels if one of its channels is depleted.

3 Related work

Unlike the transmission of data in networks, the forwarding of payment in PCNs changes the capacities of every node it passes through. It is possible that a channel is depleted because too many payments go through it. In PCNs, atomicity needs to be satisfied in most payment protocols because a payment can only be failed or successful. These differences make it hard to apply existing routing algorithms in PCNs. Thus, a number of routing algorithms and protocols are proposed to increase the performance of PCNs.

3.1 Source routing algorithms

In source routing, the sender determines the route of payments based on PCNs' topology. Such a task can be simplified as a single source graph theory problem which means there are existing exact algorithms [6], [7], [14] that can be utilized.

Based on the BellmanFord shortest path algorithm, Cheapay [44] realized a routing protocol with the minimum fees. To route payments through multiple paths, Di Stasi et al. proposed a heuristic k-shortest path algorithm[38]. It splits a payment into k sub-payments and finds the k approximately shortest paths using a heuristic method.

Coinexpress [43] tries to maximize the success rate of payments by implementing a distributed max-flow routing algorithm. It finds augment flows by a distributed BFS algorithm. After flows with enough capacities are found, the sender terminates the finding process and forwards its payment.

Flash improves the distributed max-flow routing algorithm further by handling payments with different sizes differently [42]. Flash routes large payments using the distributed max-flow algorithm. For other payments, Flash routes them using the capacity information obtained before. In this way, Flash decreases the number of messages required for distributed max-flow routing algorithm.

The algorithms above can have a good performance in the ideal situation where channels' capacities are accurate and static. However, the capacities of channels change continuously in a real PCN. Also, capacity information included in messages can be malicious. Thus, a high-performance source routing algorithm is hard to realize in a real PCN.

3.2 Rebalancing of PCNs

To increase the throughput of PCNs, REVIVE is proposed as a protocol to refund depleted channels without on-chain transactions [18]. The main idea of REVIVE is to let nodes refund their channels through cycles of PCNs. Initially, participants of REVIVE select a leader and send their rebalancing requests to this leader. This leader

can calculate a solution that maximizes the rebalanced funds by linear programming [27].

REVIVE is a useful protocol to decrease the number of depleted channels in PCNs. Nevertheless, it highly depends on the topology of PCNs and the capacities of other channels. Therefore, apart from rebalancing protocols, PCNs still need a high-performance routing algorithm.

3.3 SpeedyMurmurs

SpeedyMurmurs is an efficient embedding-based local routing algorithm [34]. It constructs multiple spanning trees from the landmarks where landmarks are elected by the whole PCN based on the node connectivity or centrality. Based on this spanning tree, each node is assigned an id that is used to represent its coordinate in the tree. With this id, nodes can measure the distance between different nodes and forward payments to the neighbour with the closest distance.

When a sender wants to forward a payment, it splits the amount into several parts randomly at first. Then, different sub-payments are sent to the receiver based on the receiver's coordinate. Before the real forwarding, the sender probes nodes in paths to obtain their capacity. To avoid concurrency issues, nodes block the required funds before it forwards the probe to the next node. After probes are received, the sender learns about paths' capacity and can decide whether to forward this payment.

By introducing the embedding-based routing algorithm, SpeedyMurmurs achieves a higher success rate and lower delay than existing source routing algorithms. However, it still has several issues from the perspective of implementation. First, it assumes that landmarks are already known. However, the method to elect such landmarks remains a problem. The connectivity and centrality of a node may be hard to compute in a PCN. Second, probes are required to obtain the capacity of paths. The malicious nodes that forward fake probes are not considered in this algorithm.

3.4 Boomerang: a payment protocol with redundancy

Boomerang extends the traditional AMP protocol to make it support redundant payments[5]. It uses the characteristics of polynomials and homomorphic hash functions to realize the redundancy of payments. At first, a polynomial with order u is chosen by the receiver where its coefficients are $a_1, a_2...a_u$. The receiver sends hashes of coefficients to the sender. Then, the sender can use these hashes to calculate the hash of the chosen polynomial for some variables because of the homomorphic property. The sender calculates $u + v$ such hashes and sends them with $u + v$ sub-payments. The receiver can admit this payment when it receives u hashes. In this way, payment can still be successful if there are v failed sub-payments.

Boomerang decreases the latency and increases the success rate by adding redundancy to the routing process. However, such a measure occupies more funds in the PCN without paying more routing fees which may be unfair to participants. Also, the number of sub-payments increases as the payment size increases that is not scalable.

At the same time, Boomerang only supports sub-payments with the same payment size that is not flexible enough.

3.5 Spear: a low latency payment protocol with redundancy

Similar to Boomerang, Spear is another payment protocol with redundancy [32]. It uses a modified HTLC to realize the redundancy of payments. Assume Alice is the sender of a payment and Bob is the receiver of this payment. Alice splits the payment into k sub-payments and forwards them through k paths. For each sub-payment, Alice chooses a different hash and uses the chosen hash and Bob's hash to construct an HTLC. After those sub-payments are received by Bob, Bob can choose some of them and ask Alice for preimages of these sub-payments. If the sum of these sub-payments equals the desired payment size, Alice sends its preimages to Bob and Bob can unlock funds using these preimages.

Spear achieved a low latency because the sender does not need to worry about overdrawing the receiver. It also supports sub-payments with different payment sizes. It inspires us how to design a payment protocol with a changeable payment size. Spear, however, needs to decide the payment size of each sub-payment beforehand. It is difficult in a PCN because the balances of channels vary all the time.

3.6 Spider: a high throughput routing algorithm

Spider is a packet-switched routing method for PCNs. It splits a payment into small sub-payments and forwards them separately. Rather than forwarding these sub-payments at once, a sender can forward them at a constant speed. Such a mechanism is helpful to balance the capacities of channels and increase the throughput of PCNs because intermediaries can digest a large payment slowly.

Spider also includes a so-called WaterFilling algorithm to balance the capacities of intermediaries. When an intermediary receives several payments, it consumes the channel with the highest capacity until its capacity equals the second-highest capacity. After that, two channels in the last step are consumed until their capacity is equal to the third-highest capacity. The same process repeats until all payments are handled. WaterFilling algorithm decreases the number of depleted channels and can increase the throughput of PCNs further.

Spider's packet-switched routing has the same problem as Boomerang. It drastically increases the number of payments and load of intermediaries.

3.7 Ethna: Non-Atomic Payment Splitting

Ethna is a payment protocol that supports payment splitting and non-atomic payments [9]. With Ethna, intermediate nodes can split a payment into sub-payments and forward them to different neighbours. Intermediate nodes can also forward payments with a smaller payment size. In this case, the payment can still be partly completed with a smaller payment size.

In Ethna, the sender forwards its payment to intermediaries at first. Intermediaries can split received payments and decrease the payment size by themselves. When the receiver receives a sub-payment, it can acknowledge it by signing a sub-receipt. This sub-receipt includes the time-lock and deserved funds of each party in the path. Later, if some sub-payments' paths share the same node, the receiver should calculate the sum of deserved funds for this node and include it in the sub-receipt. Finally, each party can unlock their funds using received sub-receipts.

Ethna realized a flexible payment protocol but it is inconvenient to use at the same time. The implementation of Ethna depends on smart contracts. However, PCNs like Lightning Network do not support smart contracts. Also, non-atomic payments may cause trouble for the sender. In a real situation, the sender will not receive a good unless all is paid, so partial payment is often not sensible.

3.8 Splitting Payments Locally

Eckey et al. designed a protocol to enable intermediate nodes to split a payment based on their local view of channels' capacities which is able to increase the success rate of local routing [10]. This protocol also ensures that no honest party can lose money.

This protocol is similar to the AMP protocol mentioned before. In an AMP protocol, only the sender is able to split payment. In splitting locally, every node has the right to split and route a payment.

For the routing of payments, two modes are provided:

- **Hop Distance routing:** payments are forwarded to nodes with shorter hop distance. The hop distance is the length of the shortest path between two nodes. This routing algorithm is similar to the routing algorithm in Lightning Network except intermediate nodes can split payments arbitrarily.

- **Interdimensional SpeedyMurmurs (INTSM) routing:** this algorithm is based on SpeedyMurmurs. At first, multiple spanning trees are constructed and maintained as in SpeedyMurmurs. After that, each node is assigned multiple ids which are used to represent its position in spanning trees. In SpeedyMurmurs, payments are split by the sender and forwarded separately based on different spanning trees. In this algorithm, spanning trees are considered as a whole. An intermediate node can forward payment to a neighbour if one of this neighbour's ids is closer to the receiver than itself.

This protocol is also extended to support unlinkable payment splitting which means an intermediate node is not able to distinguish between a normal payment and a split payment. This functionality is realized by an additive homomorphic hash function and an additive homomorphic encryption algorithm. In the extended protocol, every intermediate node creates a new preimage and a hash. The new hash is added to the hash of the HTLC with former nodes. The corresponding preimage is encrypted with the receiver's public key and added to a specific message. This message includes the sum of every intermediate node's encrypted preimage. Because of the additive homomorphic

property, the receiver can calculate the required preimage with this message and unlock coins in HTLCs.

Although this protocol can increase the success rate of payments, it still has several shortcomings. At first, one failed sub-payment can make the whole payment fail in splitting locally. Secondly, routing algorithms in this protocol consume all capacity of a channel if the forwarded payment is large. It may result in more depleted channels and influence the throughput of the PCN.

4 System model

We model a PCN as a directed graph $G = (V, E)$ with a capacity function C. Vertex set V represents nodes in the PCN, edge set E includes channels between different nodes, and function C denotes the amount of coins that can be transferred through a channel. In PCNs, two parties lock their coins in a payment channel, and the number of coins each party locks determines how many coins can this party spend. For example, if Alice locks 10 coins in the payment channel with Bob, she can send 10 coins to Bob at most. In our model, the weights of edges represent the usable coins of each party. Given a node v_i and a channel (v_i, v_j), C returns usable coins of the given node in a channel which is denoted as $C(v_i, (v_i, v_j))$. Similarly, the weight of edge from v_i from v_j equals $C(v_i, (v_i, v_j))$. In PCNs, the balance of nodes can change because of off-chain transactions. A real PCN may also support conditional payments like HTLCs. Thus, C should maintain different parties' balances for different channels continuously.

The activities of payment channels are abstracted as a set of APIs. Different parties can call those APIs to communicate and construct HTLCs with each other. Those APIs can also send messages to corresponding parties to trigger their callback functions.

We use four APIs for different events. "cPay" is called when a party wants to construct an HTLC with a neighbour. Unlike the HTLC of Lightning Network, two hashes are passed to APIs in our model because of our protocol's different design. The reason behind it is discussed in Chapter 5. After "cPay" is called, it sends "cPaid" messages to corresponding neighbours. "updateHTLC" function is called when a party wants to modify the size of its payments. At first, this function checks the validity of update requests. A valid update request should be sent from the payee rather than the payer. If Alice sends a payment to Bob, only Bob is able to update this payment. If the update request is valid, "updateHTLC" changes the payment size of the corresponding HTLC and sends a "updatedHTLC" message to the payer. "cPay-unlock" function is called when a party knows the preimages of an HTLC and wants to unlock the money in this HTLC. This API checks the validity of preimages and sends "cPay-unlocked" messages to corresponding parties if provided preimages are correct. At the same time, this API also should maintain the capacity function correctly. In case of expired payments, parties can use "refund" API to get their money back. For simplicity, we denote the calling to an API as $API \longrightarrow F$ where F is a Turning machine that implements those APIs. Tables 4.1 and 4.2 show the detailed parameters of APIs.

Table 4.1: Meaning of parameters

pid	payment id
e	corresponding payment channel's id
$size$	payment size

fee: routing fees
h_S, h_R: hashes for HTLC
x_S, x_R: preimages for HTLC
T: time lock
R: the receiver's id

Table 4.2: APIs

$(cPay, pid, e, size, fee, h_S, h_R, T, R) \longrightarrow F$

If the caller has enough funds in channel e, construct an HTLC among channel e. The amount of locked funds equals $size + fee$, the time-lock is T, and hashes of preimages are h_S and h_R. Afterwards, sends $(cPaid, pid, e, size, fee, h_S, h_R, T, R)$ to the other party in channel e.

$(updateHTLC, pid, size, fee) \longrightarrow F$

Check whether the sender has an unfinished payment whose ID is pid. If so, compare this payment's $size$ and fee with the new $size$ and fee. If the new $size$ and fee are smaller, invalidate the old HTLC and make a new one with the new $size$ and fee.

$(cPay\text{–}unlock, pid, x_S, x_R) \longleftarrow F$

If x_S and x_R are valid preimages for payment pid's hashes, unlock the locked funds and assign it to the payee.

$(refund, pid) \longleftarrow F$

If current timestamp is larger than payment pid's timelock, unlock the locked funds and return it to the payer.

For security analysis, we assume the existence of a probabilistic polynomial time adversary A who tries to steal money from other honest parties. It can choose an arbitrary number of parties to corrupt and control the input, output, and communication of those corrupted parties. To execute a protocol $\pi(G(V, E))$ with A, we assume a Turing machine F implements the APIs above and is able to communicate with each party. At first, we initiate F with the PCN topology $G(V, E)$ and a capacity function C. Secondly, F sends G, the capacity information, and other necessary information to corresponding parties. Then, adversary A can choose some parties to corrupt. After that, honest parties interact with each other based on π. Protocol π terminates when all uncorrupted parties return their output and the result of this execution is denoted as: $EXEC^F_{\pi,A}(G(V, E), C, \Delta, S, R, size, fee) = \{C', honest\}$, where C' is the capacity

function after this execution, *honest* is the set of honest parties, and Δ is the time to synchronize a transaction on the blockchain.

During the whole process, we assume the communication between different parties and calls of APIs are secure. Thus, adversary A is not able to modify or drop messages of honest parties. In practice, cryptography technology likes digital signature can be used to prevent messages from being tampered. To ensure the delivery of messages, participants can use acknowledgments to check whether the sent message is received. Similar to split locally, we also assume the communication is synchronous with exactly 1 round delay. It ensures the message can be delivered at the next round for sure.

4.1 Security requirements

Eckey et al. defined balance security, atomicity, and finality in splitting locally [10]. We define our security requirements similarly since our protocol is built upon it. The only difference is that fees are included in our security requirements because we also want to realize a fairer incentive mechanism.

To describe those security requirements in a formal way, we define what is a valid receipt at first. Definition 1 indicates that a receipt should include the sender's address S, the receiver's address R, payment size, the sender's hash h_s, and the receiver's hash h_r. S and R are used to identify the participants of a receipt. h_s and h_r are hashes chosen by the sender and receiver to construct HTLCs. Only if the preimages of both hashes are revealed, other parties can get money from the sender. In this case, the sender knows the receiver's preimage if it loses any coins. The sender and other parties can check the validity of the receipt by:

$$validate(receipt(S, R, paymentsize, h_s, h_r), x_s, x_r) = \begin{cases} True & H(x_s) = h_s \wedge H(x_r) = h_r \\ False & otherwise \end{cases} \tag{4.1}$$

where x_s is the preimage for the sender's hash, x_r is the preimage for the receiver's hash, and $H()$ is a hash function that satisfies both the preimage resistance and second preimage resistance. Thus, if the sender loses money, it knows the preimages of hashes and holds a valid receipt.

Definition 1 $receipt(S, R, size, h_s, h_r) = Sign_{sk_R}(S, R, payment\ size, h_s, h_r)$ *where S and R are the addresses of the sender and receiver, sk_R is the secret key of the receiver R, and $Sign$ is a digital signature algorithm (public keys of each party are assumed to be known by everyone because each party has at least one open channel transaction on-chain and the public key of this party is included in this transaction).*

Then, we can define the security requirements formally. Balance security for intermediate nodes means intermediate nodes never lose their money. In definition 2, we sum an honest party's money over all channels after execution of a payment and subtract the money of this party before execution from it. If the result is equal or larger than 0, we can say this party never loses money.

Definition 2 (Balance security for intermediaries)
$\forall\ v_i \in honest\ where\ v_i \neq S$
$\sum_{(v_i,v_j)\in E} C^{'}(v_i,(v_i,v_j)) - \sum_{(v_i,v_j)\in E} C(v_i,(v_i,v_j)) \geq 0$

Bounded loss for the sender means an honest sender never loses more money than the payment size plus the fees. We define it similar to definition 2.

Definition 3 (Bounded loss for the sender) *if* $S \in honest$
then $\sum_{(S,v_j)\in E} C(S,(S,v_j)) - \sum_{(S,v_j)\in E} C^{'}(S,(S,v_j)) \leq payment\ size + fee$

Atomicity means if an honest sender loses any coins, it holds a receipt that admits the whole payment and fees have been paid.

Definition 4 (Atomicity)
if $\sum_{(S,v_j)\in E} C(S,(S,v_j)) - \sum_{(S,v_j)\in E} C^{'}(S,(S,v_j)) > 0 \wedge S \in honest$
then $validate(receipt(S,R,payment\ size,h_s,h_r),x_s,x_r) = true$

Finality means all honest parties should terminate in finitely many rounds.

Definition 5 (Finality) $\forall\ v_i \in honest$, v_i *terminates in nitely many rounds.*

Formal security requirements : given any graph G=(V,E) where $(V_i,V_j) \in E \Leftrightarrow (V_j,V_i) \in E$, capacity function C, ppt adversary A, sender S $\in$ V, receiver R $\in$ V, payment size $size$, and fees fee, protocol π is secure if its result $EXEC^{F}_{\pi,A}(G(V,E),C,\Delta,S,R,size,fee) = \{C^{'},honest\}$ satisfies Definition 2, 3, 4, and 5.

5 Payment Protocol

Our payment protocol JustForward includes three phases: forward, modify, and complete. In the forward phase, the sender splits a payment into some sub-payments and forwards them to neighbours. In this phase, we use two hashes to construct HTLCs. One hash is chosen by the receiver. It is used to check whether the receiver has received its money. The other hash is chosen by the sender because we need to make sure the payment size is correct. In our protocol, intermediaries and the receiver have the right to modify the payment size. Only if the modified payment size is correct, can the sender reveal its preimage. In this way, the sender can avoid losing funds. Unlike payments in the Lightning network, intermediate nodes in JustForward can split sub-payments further or decrease the payment size. In the modification phase, the receiver needs to receive enough sub-payments from neighbours at first. Then, it can choose multiple sub-payments as candidates and revoke other sub-payments. If the sum of sub-payments is larger than the desired payment size, the receiver can request candidates to decrease their payment sizes. Intermediate nodes continue to revoke or modify their payments if they are requested to do so. In the complete phase, the sender checks whether the updated payment size equals the desired payment size. If so, the sender sends the preimage it holds to the receiver. The receiver uses this preimage and its own preimage to complete all HTLCs in the complete phase.

5.1 High-level explanation of JustForward

The key idea of JustForward is to forward payments with redundancy and revoke those redundant payments later. The forward phase is similar to "Splitting locally" except intermediate nodes are also able to decrease the payment size in JustForward. As shown in figure 5.1, the sender sends 5$ to intermediary1 at first. Then, intermediary1 decreases the payment size to 4$ and splits it into two sub-payments. It forwards 2$ to intermediary2 and 2$ to intermediary3. Intermediary2 decreases the payment size to 1$ and forwards it to the receiver while intermediary3 forwards 2$ to the receiver.

In JustForward, the size of sub-payments arriving at the receiver may be larger or smaller than the desired payment size. If there are not enough sub-payments, the receiver can conclude that this payment can not be completed. Otherwise, the receiver needs to revoke or modify some sub-payments. To ensure the receiver can not obtain additional money from the sender, the sender needs to have some control over HTLCs. Thus, HTLCs in JustForward have two conditions: one hash from the sender and one hash from the receiver. The revoking of payments is done by revoking corresponding transactions that is a normal operation in PCNs [31]. In PCNs, a party is able to create a temporary address to receive and spend money. The payee creates a temporary address with a temporary private key and a temporary public key for each revocable

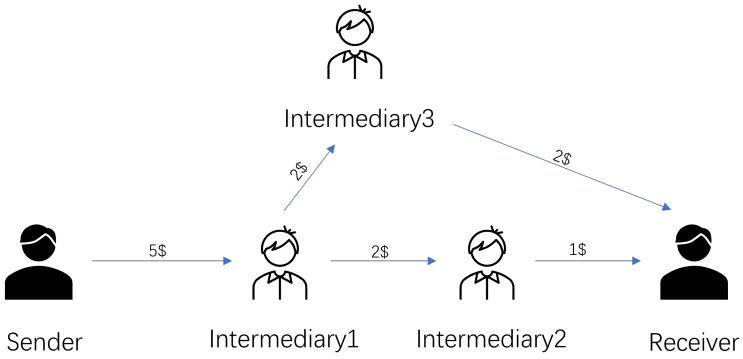

Figure 5.1: Forward phase of JustForward

transaction. It sends the temporary public key and address to the payer. Then, the payer transfers money to the payee's temporary address by spending its output. In our case, the payer creates an HTLC at first. One output of this HTLC is set to the payee's temporary address. Hence the payee can receive the payer's money if it can reveal the required preimage within the timelock. Afterwards, the payee can spend the output of the HTLC by signing new transactions with its temporary private key. To revoke the old HTLC, the payee gives its temporary private key to the payer. By doing so, the payer is able to sign a new HTLC with the payee's temporary private key. In this HTLC, the payer chooses a smaller timelock. So, the payer can always spend the output of the payee's temporary address before the payee. If the payee still tries to spend the output of its temporary address, the payer can punish the payee by publishing the HTLC generated by itself on-chain. Then, the payee will lose its funds because of its dishonest behavior. So, we can safely say that the old HTLC is "revoked". In our case, a payee will not revoke an HTLC unless there is a new HTLC that can ensure its balance security. As shown in figure 5.2, the payee first sends an update request to an intermediary. The intermediary signs a new HTLC and sends it to the payee. The payee gives the old HTLC's temporary private key to the intermediary. One may ask what happens if the payee does not give its temporary private key to the intermediary. In this case, the intermediary can notice that the old HTLC is not revoked successfully and refuse to modify former HTLCs to prevent the loss of funds. Then, the whole payment fails because of unrevoked sub-payments. Otherwise, the intermediary modifies the HTLC with the payer obeying the same process.

A natural question is why should intermediate nodes revoke their sub-payments as requested. To overcome this problem, we introduce revoke fees for intermediate nodes. Nodes with revoked payments can still get some incentives that motivate them to cooperate with the sender and the receiver. We use a separate transaction with a new HTLC to forward revoke fees. So, two transactions with different temporary secret keys are required for one sub-payment: one for the normal payment and the other one for revoking fees. For the amount of fees, the sender decides the maximal amount of fees it is willing to pay initially. Intermediaries take fees as they see fit and forward the remaining fees to the next node. Currently, we assume a function decides the fees of

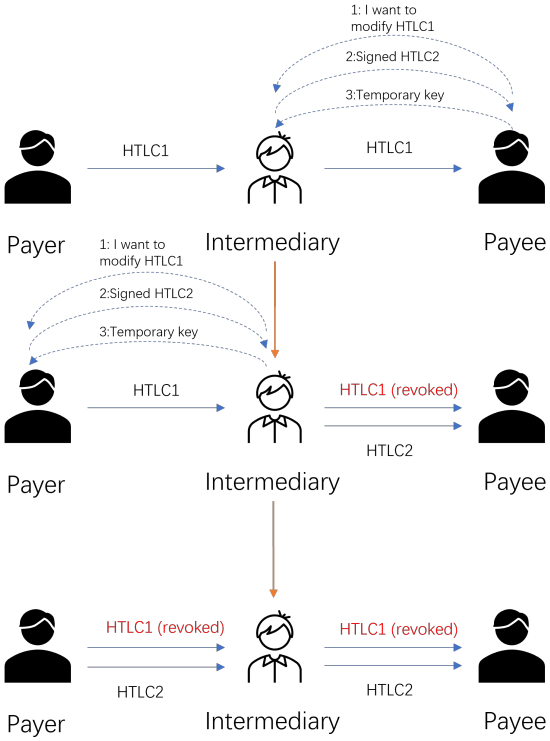

Figure 5.2: Revoke and update HTLCs

intermediaries. This function takes the payment size and remaining fees as input and outputs the deserved fees of the current node. For the detailed fee policy, we discuss it in section 5.4. With this construction, the transaction for revoke fees still exists after the revoking of a normal payment. One may argue that an intermediate node can drop every payment to benefit the most. This argument does not consider the influence of success rate, fee rate, and behaviors of other parties. At first, the revoking of sub-payments decreases the success rate of payments that also damages intermediaries' expected profits. Secondly, an intermediate node can not always benefit the most by dropping every payment. For an intermediary:

$$profit = \sum_{i=1}^{n} \left[fwd_i/size_i \cdot fee_i^{normal} + (1 - fwd_i/size_i) \cdot fee_i^{revoke} \right]$$

$$= \sum_{i=1}^{n} \left[fwd_i/size_i \cdot (fee_i^{normal} - fee_i^{revoke}) + fee_i^{revoke} \right]$$

where n is the number of received payments, fwd_i is the forwarded amount of i-th payment, $size_i$ is the payment size of i-th payment, fee_i^{normal} is the maximum normal fee for i-th payment, fee_i^{revoke} is the maximum revoke fee for i-th payment, and *capacity* is the maximum amount of funds that can be forwarded. Since the revoke fee is less than the normal fee, an intermediary needs to forward every received payment to obtain the

maximum profit when the sum of payment size is not larger than this node's capacity:

$$profit_{max} = \sum_{i=1}^{n} fee_i^{normal},\ if\ \sum_{i=1}^{n} size_i <= capacity$$

When the sum of revoke fees is larger than an intermediary's capacity, this intermediary can drop every payment to get the maximum benefit:

$$profit_{max} = capacity,\ if\ \sum_{i=1}^{n} fee_i^{revoke} >= capacity$$

Otherwise, an intermediary needs to drop some payments and forward other payments to make the best use of its capacity. So, an intermediary can benefit most by dropping every received payment only when it receives a large number of payments. However, if a node drops every payment, it is less likely for its neighbours to forward new payments to it because it causes neighbours' payments to be revoked and reduces neighbours' expected profit. Then, this node needs to forward payments again.

In the modify phase, the receiver revokes both the normal payment and the transaction for revoke fees. It makes new transactions with recalculated payment size and revoke fees. Intermediaries can check the correctness of the new transaction and continue to modify the transaction with previous nodes.

It is also possible that a node can not forward payments further because of the insufficient capacities of connected channels. In this case, this node still needs to forward revoke fees to the receiver through other intermediaries because it is necessary to incentive the revoking of payments. If none of its channels' capacity is larger than the revoke fees, this payment will definitively fail because of non-revoked sub-payments.

Ideally, the sender finds that the sum of modified sub-payments equals the desired payment size. Then, the sender sends its preimage to the receiver through a secure communication channel. Later, the receiver can obtain money from neighbours by revealing the sender's preimage and its own preimage. If the sum of sub-payments is incorrect, the sender can know this payment is failed and refuse to hand over its preimage.

5.2 Formal description

In this section, we describe JustForward formally for different parties. *Route* is the function to split and route payments. It decides how to split and forward a payment based on the topology of the PCN and the capacities of connected channels. In JustForward, *Route* may also decrease the size of payment when the capacities of channels are limited. *Reschedule* is used to determine the updated payment size of each sub-payment.

For HTLCs, we need to ensure there is enough time for honest parties to update payments and publish their HTLCs on-chain. In the worst case, it takes $(\Delta + 1)$ rounds for an intermediate node to know that its payment is published on-chain by neighbours. To get money back, this node needs another $(\Delta + 1)$ rounds to publish the HTLCs with

former nodes on-chain. So, if we want to make sure honest nodes have enough time to publish their HTLCs (which also means honest parties will not lose coins), the difference of time-locks for adjacent nodes should be $2 \cdot (\Delta + 1)$ at least. So, if there are n intermediate nodes, the sender's time-lock should be $t_0 + 2n \cdot (\Delta + 1)$ where t_0 is the start point of the payment. Since local routing is applied in JustForward, it is impossible to know the number of intermediate nodes beforehand. In a real situation, a sender can set this time by itself to limit the number of intermediate nodes. We set the time-lock for the worst situation where every node in the PCN is visited (one node can't be visited multiple times). Apart from the time to publish HTLCs, F needs another round to send messages to corresponding parties. So, the sender sets its time-lock to $t_0 + |V|(1 + 2 \cdot (\Delta + 1))$, where $|V|$ is the number of nodes in the PCN. JustForward has a revoke phase which is done by the calls to $updateHTLC$. $updateHTLC$ also needs another round to send messages to corresponding parties. The sender adds a round for each intermediate node to the time-lock and the time-lock becomes $t_0 + |V|(2 + 2 \cdot (\Delta + 1))$.

Figure 5.3 depicts the process of JustForward. At first, the sender and receiver exchange their hashes. Then, the sender forwards its payment to intermediaries. In this step, intermediaries are able to decrease the payment size and split a payment into several sub-payments. After sub-payments are received by the receiver, the receiver can use "updateHTLC" to modify the size of those sub-payments. If the sum of modified sub-payments equals the desired payment size, the sender sends its preimage to the receiver and the receiver can use this preimage to unlock its funds.

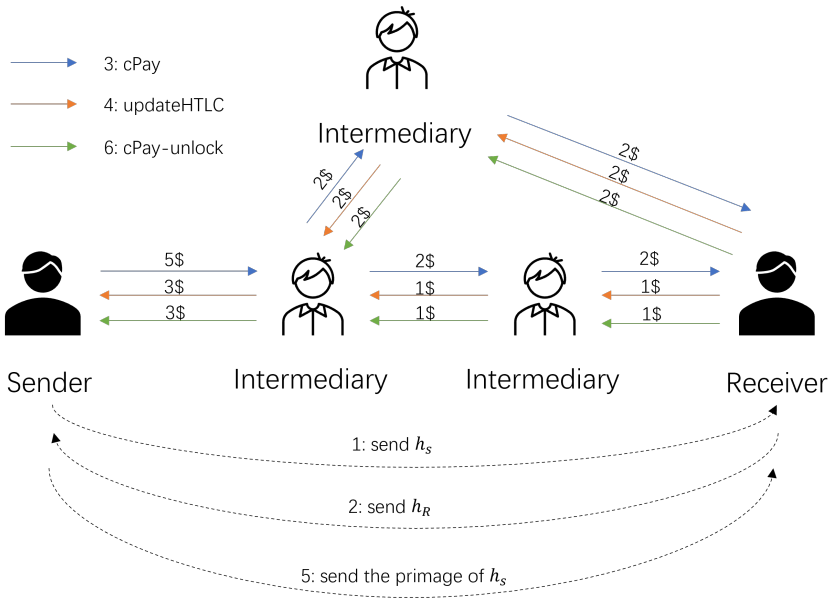

Figure 5.3: Forward a payment using JustForward

Meaning of variables and functions

$a \longrightarrow b$: send message a to b
$x \longleftarrow \mathsf{Z}_P$: sample a number in the number field Z_P and assign it to variable x
$id \longleftarrow \{0,1\}^*$: chose a random id
F: calling to APIs
S: sender's ID
R: receiver's ID
G: the topology of PCN
C_i: node i's capacity function, can output this node's capacity in each channel
$x_S \; and \; x_R$: sampled preimages for HTLCs
$H(x)$: a hash function
$Sign_k(m)$: sign the message m using key k
$Vrfy_k(m, Sig)$: verify the signature Sig of message m using key k
T: the latest time to complete a payment
$C_{redundant}(\geq 1)$: redundancy of payments
C_{revoke}: the ratio of routing fees to revoke fees
$fee^{routing}$: fees if the whole payment succeeds
fee^{revoke}: fees if the whole payment is revoked
Δ: time needed to publish a payment on chain
$[k]$: Natural numbers from 1 to k
$Route(amount, I, R)$: For the payment from I to R with size $size$, decide how to split it and what is the next node
$Fee(totalfees)$: decide fees for current node based on a fee policy
$Reschedule_G(candidates, amount)$: choose some sub-payments from candidates to let the sum of them equal amount, the size of those sub-payments also can be adjusted

In the initialization phase, different parties obtain necessary information and initialize their variables.

Initialization

Sender:

1: $Receive\ (G, S, R, size, fee, C_S)$
2: $out := \emptyset$ {The set of all sub-payments}
3: $sum_{pay} := 0$ {Sum of updated payment size}
4: $sum_{fee} := 0$ {Sum of updated fees}

Receiver:

1: $Receive(G, S, R, size, fee, C_R)$
2: $in := \emptyset$ {The set of received sub-payments}
3: $candidate := \emptyset$
4: $sum_{pay} := 0$ {Sum of received sub-payments}
5: $finish := 0$ {Are all candidates decided}

Intermediaries:

1: $Receive(G, C_I)$
2: $fw := \emptyset$ {The set of forwarded sub-payments}
3: $cnt := \emptyset$ {Count sub-payments for each received payment}
4: $sum_{pay} := \emptyset$ {Sum of updated sub-payments' sizes}
5: $sum_{fee} := \emptyset$ {Sum of updated sub-payments' fees}
6: $old_{pay} := \emptyset$ {The initial payment size of each payment}
7: $oldFee^{routing} := \emptyset$ {The initial routing fee of each payment}
8: $oldFee^{revoke} := \emptyset$ {The initial revoke fee of each payment}

In step 1, the sender sends its hash to the receiver. Upon receiving the sender's hash, the receiver checks the signature of this hash and sends back its hash if the signature is valid. Upon receiving the receiver's hash, the sender checks the receiver's signature. If the signature of the receiver is valid, the sender decides how to forward its payment using the "Route" function in line 3 of step 3. In lines 4 to 9, the sender forwards and stores each sub-payment. In step 4, F checks whether the party who calls "cPay" API has enough capacity. If so, F maintains this party's capacity accordingly and sends a "cPaid" message to the next node. After a "cPaid" message is received, the intermediary calculates the fees and timelock in lines 2 to 4 of step 5. From line 5 to line 12, it routes and forwards the received payment like the sender. In step 6, the receiver receives a "cPaid" message at first. In line 2, the receiver calculates the sum of received sub-payments. In line 3, the receiver stores the received sub-payment into the candidate set. If the amount of received sub-payments is larger than the payment size, the receiver chooses sub-payments using the "Reschedule" function in line 7. From line 6 to line 10, the receiver updates rescheduled sub-payments by "updateHTLC" API. In lines 13 and 14, the receiver sets all later payments' size to 0.

Forward phase

(1) Sender initiates a payment:
In round t_0, the sender samples a preimage and calculate the hash value of it. Then, this hash is signed and sent to the receiver.

1: $x_S \longleftarrow \mathsf{Z}_P, h_S := H(x_S)$
2: Send $(h_S, Sign_{sk_S}(h_S))$ to the receiver

(2) Receiver upon receiving (h_S, Sig_{h_S})**:**
If the received signature is valid, the receiver sends a signed message back.

1: **if** Sig_{h_S} is a valid signature of the sender **then**
2: $x_R \longleftarrow \mathsf{Z}_P, h_R := H(x_R)$
3: $Sig = Sign_{sk_R}(S, R, size, h_S, h_R)$
4: Send $(init, h_S, h_R, Sig)$ to the sender
5: **else**
6: $Terminate$
7: **end if**

(3) Sender upon receiving $(init, h_S, h_R, Sig)$**:**
In round $t_0 + 2$, the sender checks the validity of the receiver's message. If it is valid,

the sender tries to route and forward its payment.

1: **if** Sig is a valid signature for $(S, R, size, h_S, h_R)$ **then**
2: $T := t_0 + 2 + |V|(2 + 2 \cdot (\Delta + 1))$
3: $(e_j, size_j)_{j \in [k]} = Route_G(size \cdot C_{redundant}, S, R)$
4: **for** $j \in [k]$ **do**
5: $fee_j = size_j / size \cdot fee$
6: $pid_j \longleftarrow \{0,1\}^*$
7: $out := out \cup \{(e_j, size_j, fee_j, pid_j)\}$
8: $(cPay, pid_j, e_j, size_j, fee_j, fee_j \cdot C_{revoke}, h_S, h_R, T, R) \longrightarrow F$
9: **end for**
10: **else**
11: $Terminate$
12: **end if**

(4) F receives $(cPay, pid, e, size, fee_{routing}, fee_{revoke}, h_S, h_R, T, R)$ **from** I **or** S**:**

1: **if** I's capacity in channel $e \geq size + fee$ **then**
2: decrease I's capacity by $size + fee$
3: send $(cPaid, pid, e, size, fee^{routing}, fee^{revoke}, h_S, h_R, T, R)$ to the other party of channel e
4: **else**
5: $Terminate$
6: **end if**

(5) Intermediary upon receiving $(cPaid, pid, e, size, fee^{routing}, fee^{revoke}, h_S, h_R, T, R)$**:**

1: **if** $current\ round \leq t_0 + 1 + |V|$ **then**
2: $T := T - 2(\Delta + 1)$
3: $cnt[T] := 0, sum_{pay}[T] := 0, sum_{fee}[T] := 0, old_{pay}[T] := size$
4: $oldFee^{routing}[T] := fee^{routing}, oldFee^{revoke}[T] := fee^{revoke}$
5: $(e_j, size_j)_{j \in [k]} = Route_G(size, I, R)$
6: **for** $j \in [k]$ **do**
7: $fee_j^{routing} = Fee(size_j / size \cdot fee^{routing})$
8: $fee_j^{revoke} = Fee(size_j / size \cdot fee^{revoke})$
9: $pid_j \longleftarrow \{0,1\}^*$
10: $fw[T] := fw[T] \cup \{(e_j, size_j, fee_j^{routing}, fee_j^{revoke}, pid_j, pid)\}$
11: $cnt[T] := cnt[T] + 1$
12: $(cPay, pid_j, e_j, size_j, fee_j^{routing}, fee_j^{revoke}, h_S, h_R, T, R) \longrightarrow F$
13: **end for**
14: **else**
15: $Terminate$
16: **end if**

(6) Receiver upon receiving $(cPaid, pid, e, size, fee^{routing}, fee^{revoke}, h_S, h_R, T, R)$**:**

The receiver calculates the sum of received sub-payments. If the sum reaches the

payment size, the receiver tries to reschedule received sub-payments.

1: **if** $finish = 0$ **then**
2: $sum_{pay} := sum_{pay} + size$
3: $candidate = candidate \cup (pid, size, fee^{routing}, fee^{revoke})$
4: **if** $sum_{pay} >= size$ **then**
5: $(pid_j, size_j, fee_j)_{j \in [k]} = Reschedule_G(candidate)$
6: **for** $j \in [k]$ **do**
7: $(updateHTLC, pid_j, size_j, fee_j) \longrightarrow F$
8: $in = in \cup \{pid_j\}$
9: $finish = 1$
10: **end for**
11: **end if**
12: **else**
13: $(updateHTLC, pid, 0, fee^{revoke}) \longrightarrow F$
14: $in = in \cup \{pid\}$
15: **end if**

In the modify phase, F receives participants' update requests first. It checks whether the update request is sent from a payment's payee. If so, F changes the corresponding party's capacity and sends an "updatedHTLC" message to the found payment's payer. Upon receiving an "updatedHTLC" message, the intermediary I finds the corresponding split sub-payment from the fw set. In line 2 of step 8, I determines whether the updated payment size and fees are correct. If so, it updates the stored sub-payment with the new payment size and fees. In lines 8 and 9, I sums split sub-payments' fees and payment size for each corresponding unsplit sub-payment. If all split sub-payments are updated, I updates the unsplit sub-payment with recalculated payment size and fees in lines 11 and 12. In line 13, I requests the former node to update its payment using "updateHTLC" API. In step 9, the sender receives a "updatedHTLC" message from F. From line 2 to line 4, the sender update stored sub-payments and sums the updated payment size and fees. If the updated payment size and fees are correct, it sends its preimage to the receiver. Otherwise, the sender terminates immediately and the forwarded payment fails.

Modify phase

(7) F upon receiving $(updateHTLC, pid, size_{new}, fee_{new})$ **from** I **or** R**:**

1: **if** I has a payment with pid **then**
2: increase the capacity of the former node based on $size_{new}$ and fee_{new}
3: send $(updatedHTLC, pid_0, size_{new}, fee_{new})$ to the former node
4: **end if**

(8) Intermediary upon receiving $(updatedHTLC, pid_0, size_{new}, fee_{new})$**:**
Intermediate node needs to make sure the updated payment size and fees are correct. If so, it waits until all sub-payments are updated. Afterwards, it tries to update the former payment

1: $Let\ x_j := (e_j, size_j, fee_j^{routing}, fee_j^{revoke}, pid_j, pid) \in fw[T] \quad s.t.\ pid_j = pid_0$
2: **if** $size_{new} > size_j \vee fee_{new} > size_{new}/size_j \cdot fee_j^{routing} + (1 - size_{new}/size_j) \cdot fee_j^{revoke}$ **then**
3: $Terminate$
4: **else**
5: $fw[T] := fw[T] \backslash \{x_j\}$
6: $fw[T] := fw[T] \cup \{(e_j, size_{new}, fee_{new}, 0, pid_j, pid)\}$
7: $cnt[T] := cnt[T] - 1$
8: $sum_{pay}[T] := sum_{pay}[T] + size_{new}$
9: $sum_{fee}[T] := sum_{fee}[T] + fee_{new}$
10: **if** $cnt[T] = 0$ **then**
11: $sum_{fee}[T] += sum_{pay}[T]/old_{pay}[T] \cdot oldFee^{routing}[T]$
12: $sum_{fee}[T] += (1 - sum_{pay}[T]/old_{pay}[T]) \cdot oldFee^{revoke}[T]$
13: $(updateHTLC, pid, sum_{pay}[T], sum_{fee}[T]) \longrightarrow F$
14: **end if**
15: **end if**

(9) Sender upon receiving $(updatedHTLC, pid_0, size_{new}, fee_{new})$**:**
When the sender receives update requests, it counts the sum of those updated sub-payments. If the sum equals the payment size plus fees, it sends its preimage to the receiver.

1: $Let\ x_j := (e_j, size_j, fee_j, pid_j) \in out[T] \quad s.t.\ pid_j = pid_0$
2: $out := out \backslash \{x_j\}$
3: $out := out \cup \{(e_j, size_{new}, fee_{new}, pid_j)\}$
4: $sum_{pay} += size_{new}, sum_{fee} += fee_{new}, cnt -= 1$
5: **if** $sum_{pay} = size \wedge cnt = 0$ **then**
6: $fee = fee/C_{redundant} + fee \cdot C_{revoke} \cdot (redundant - 1)/redundant$
7: **if** $sum_{pay} > size \vee sum_{fee} > fee$ **then**
8: $Terminate$
9: **else**
10: Send x_S to the receiver
11: **end if**
12: **end if**

In step 10, the receiver checks whether the received preimage is correct. If so, it unlocks its funds by calling "cPay-unlock" with the preimages it holds. In step 11, F receives preimages from the receiver or an intermediary. If the preimages are correct, F changes the corresponding party's capacity and sends a "cPay-unlocked" message to the sender or an intermediary. Afterwards, intermediaries use the preimages in "cPay-unlocked" messages to unlock their funds in line 2 of step 12. Next, the sender receives "cPay-unlocked" messages from F. It checks whether all sub-payments are completed. If so, it returns the received preimages.

Complete phase

(10) Receiver upon receiving x_S:
The receiver uses preimages to unlock funds.

if $H(x_S) = h_S$ **then**
  for $pid \in in$ **do**
    $(cPay\text{–}unlock, pid, x_S, x_R) \longrightarrow F$
  end for
  $wait\ for\ \Delta + 1\ rounds\ to\ return\ (x_S, x_R)$
end if

(11) F upon receiving $(cPay\text{–}unlock, pid, x_S, x_R)$ **from** I **or** R**:**

if x_S and x_R are valid preimages for the payment with pid **then**
  increase I's capacity by $size + fee$
  send $(cPay\text{–}unlocked, pid, x_S, x_R)$ to the former node of I
end if

(12) Intermediary upon receiving $(cPay\text{–}unlocked, pid_0, x_S, x_R)$**:**

$Let\ x_j := (e_j, size_j, fee_j, fee_j^{revoke}, pid_j, pid) \in fw[T] \quad s.t.\ pid_j = pid_0$
$(cPay\text{–}unlock, pid, x_S, x_R) \longrightarrow F$
$fw[T] := fw[T] \backslash \{x_j\}$
if $\forall T, fw[T] = \emptyset$ **then**
  $wait\ for\ \Delta + 1\ rounds\ to\ return\ (x_S, x_R)$
end if

(13) Sender upon receiving $(cPay\text{–}unlocked, pid_0, x_S, x_R)$**:**

$Let\ x_j := (e_j, size_j, fee_j, fee_j^{revoke}, pid_j, pid) \in fw[T] \quad s.t.\ pid_j = pid_0$
$out := out \backslash \{x_j\}$
if $out = \emptyset$ **then**
  $return\ (x_S, x_R)$
end if

If some parties do not behave as expected, the error handling functions below make sure that honest parties always terminate and never lose money. For the sender, if there are still some unfinished payments after the timelock, it unlocks its funds by "refund" and terminates after $\Delta + 1$ rounds. For each "refund" request, F checks the timelock of the corresponding payment. If the corresponding payment expires, F increases the capacity of the refund initiator. If the receiver does not terminate in $t_0+2+2|V|$ rounds, it simply terminates in this round. In every round, the intermediary checks whether there are expired payments and sends a "refund" request for each expired payment. When the forward set fw is empty, the intermediary terminates.

Error handling

Sender in round T**:**

for $pid \in out$ **do**
  $(refund, pid) \longrightarrow F$

3: **end for**
4: $wait\ for\ \Delta + 1\ rounds\ to\ Terminate$

F upon receiving $(refund, pid)$ **from** I **or** S**:**

1: **if** the payment with pid expires **then**
2: increase I's capacity by $size + fee$
3: **end if**

Receiver in round $t_0 + 2 + 2|V|$**:**

1: $Terminate$

Intermediary in every round:

1: **if** $fw[now] \neq \emptyset$ **then**
2: **for** $(e_j, v_j, fee_j, pid_0, pid) \in fw[now]$ **do**
3: $(refund, pid_0) \longrightarrow F$
4: **end for**
5: **if** $fw = \emptyset$ **then**
6: $wait\ for\ 2(\Delta + 1)\ rounds\ to\ Terminate$
7: **end if**
8: **end if**

5.3 Protocol with unlinkability

The protocol described in the last section uses the same hashes for different sub-payments. A party can know which payment has been split by observing its hashes and choose to drop it since split payments have a higher failure rate [10]. To overcome this issue, the former protocol can be modified to let different parties use different hashes. We use the same method as splitting locally [10] to modify JustForward. The detailed protocol is shown in Appendix A.

5.4 Fee policy

Fee policy in PCNs means how intermediate nodes charge fees for themselves. In JustForward, fee policies can be integrated into the Route function. PCNs usually apply some fixed fee policies in real life. For example, intermediate nodes charge a base fee plus a transaction fee in the Lightning network. However, such a measure does not work in local routing, since the sender does not know the whole path to the receiver and can not decide the fees in advance. A feasible solution for local routing is to let each intermediate node decide the fee by itself and the receiver gets all remaining fees. A greedy node may consume all fees. Then, no other node is willing to forward this payment because there are insufficient fees left which also means the greedy node locks its funds without any benefits. Therefore, intermediaries need to balance their profit and other nodes' interests to maximize the benefits.

Although it is hard to obtain a fixed fee policy because of nodes' different game strategies, we can still summarize properties for the desired fee policy. Assume a

payment goes through a path $v_1, v_2, \ldots, v_n$, and $f_1, f_2, \ldots, f_n$, are fees for corresponding parties, properties below should hold for all rational parties:

- A rational receiver v_n always chooses the payment with the largest f_n: f_n is the fees remaining for the receiver. The receiver profits more with a larger f_n.
- For any $i < j < n$, $f_j < f_i$ holds: a closer distance to the receiver means a faster speed and a higher chance of getting coins back. Assume the success rate of payments is suc, the profitability of intermediaries equals $f_i \cdot suc / [(n-i) \cdot \Delta]$. If the fee for each intermediary is the same, intermediaries will only forward payments to close receivers to maximize their profit. Such consequences damage the liquidity of the PCN and the profits of all parties. Hence rational parties should agree that the farther an intermediary from the receiver, the more rewards it can get.
- The obtained fee policy is Sybil-proof: Sybil-proof means an intermediate node can't get more fees by forwarding payments to Sybil nodes controlled by itself. This property ensures that honest parties' profits will not be stolen by Sybil nodes. To ensure this property, a larger n should result in a smaller f_n. Then, a longer path with Sybil nodes will have a smaller f_n. If the first property holds at the same time, this path will have a lower probability to be chosen because of the smaller f_n. In this way, malicious parties are not able to benefit from Sybil nodes.

Fortunately, there is a Sybil-proof incentive mechanism proposed by Ersoy et al. [11] that satisfies the requirements above. This mechanism is used to determine how to assign fees to parties in a path. Local routing of payments in PCNs lacks such a mechanism to assign fees to intermediaries. So, we choose it as our fee policy. In this mechanism, the fees are decided by:

$$Fee_i^n = \begin{cases} Fee \cdot C(1-C)^{i-1} & i < n \\ Fee \cdot (1-C)^{n-1} & i = n \end{cases}$$

where Fee is the fee forwarded by the sender, n is the number of involved nodes, Fee_i^n is the fee for the i^{th} node, and C is a constant. To simplify the procedure of implementation, we assume all honest parties reach a consensus on this fee policy and use the same constant C. This assumption is an equilibrium for all intermediaries. It may be reached after a long game. It is also possible that all participants reach a consensus on this fee policy beforehand.

5.5 Routing

Since the adjustments to payment size are allowed in our protocol, we define the property of routing functions differently. Given the *payment size*, current node I, and receiver R:

$$Route(payment\ size, I, R) = ((I, I_j), size_j)_{j \in [k]}$$

$$s.t. (I, I_j) \in E,\ size_j \leq C(I, (I, I_j)),\ \textstyle\sum_{j \in [k]} size_j \leq payment\ size$$

Based on the definition above, we implement a new routing algorithm. Similar to the routing algorithm of split locally, our algorithm selects neighbours whose distance is closer to the receiver at first. The distance function can also be SpeedyMurmurs based or Hop Distance based. Then, our algorithm calculates the sum of selected neighbours' capacities. The routing algorithm of split locally terminates immediately if the sum is smaller than the payment size. Our routing algorithm sets the payment size to the sum and uses the revoke mechanism to revoke the extra payment size if the sum is smaller than the payment size. Afterward, we use a so-called Waterfilling algorithm [37] to decide the payment size for each neighbour.

6 Security analysis

Generally speaking, our security analysis is similar to split locally because of the similar security requirements. The main differences between our protocol and split locally are redundancy and fees. So our main contribution is the proof of security requirements with the existence of redundancy and fees.

6.1 Termination

Lemma 1 *Finality is satis ed in JustForward.*

Proof of lemma *In the modify phase, the sender sums the updated payment size. It terminates immediately if the size of updated payments is incorrect. In the complete phase, the sender terminates when all sub-payments are nished. If some parties behave incorrectly, the sender's error handling function also ensures that it terminates in round* $T + \Delta + 1$ *where* T *is the time-lock set by the sender.*

For an honest receiver, if the preimage of the sender is received, it terminates after $\Delta + 1$ *rounds from the current round. Otherwise, its error handling function ensures that it terminates in round* $t_0 + 2 + 2|V|$ *where* t_0 *is the start round of payment and* $|V|$ *is the number of nodes in a PCN.*

An intermediary stores all sub-payments in the set fw*. Upon the receiving of a "cPay-unlocked" message, it deletes the corresponding element in* fw*. When* $fw = \emptyset$ *which means every sub-payment is completed, it terminates after* $\Delta + 1$ *rounds from the current round. If some sub-payments expire which means it is impossible to let* $fw = \emptyset$*, it terminates after* $2(\Delta + 1)$ *rounds from the current round.*

Since all honest parties terminate in nite rounds, we can conclude that JustForward satis es the de nition of nality.

6.2 Balance security of the sender

Lemma 2 *Bounded loss for the sender is satis ed in JustForward.*

Proof of lemma *Based on the de nition of APIs, a party loses money only when a valid cPay-unlock message is sent. A cPay-unlock message includes the ID of a payment. APIs can look up the payment size and fees of payment by this ID. If this message is valid, the capacity of the corresponding party decreases by* $payment\ size + fees$*. A valid cPay-unlock message requires correct preimages for* h_S *and* h_R*. In JustForward, only the sender knows the preimage of* h_S *and it only sends this preimage when the sum of payment size and fees of all sub-payments equals or is smaller than the desired value. Thus, an honest sender never loses money more than* $payment\ size + fees$ *in JustForward.*

6.3 Balance security of intermediate nodes

Lemma 3 *JustForward satisfies the balance security for intermediate nodes.*

Proof of lemma *For an honest intermediate node, the definition of the routing function ensures that the sum of $size$ and fee in cPay messages it sent is always smaller than the sum of $size$ and fee in cPaid messages it received. In the protocol without unlinkability, the same hashes are used for all sub-payments. An intermediate node can use later nodes' preimages directly. In the protocol with unlinkability, we use different hashes for different sub-payments. An honest intermediary samples a random number x in Z_P for each sub-payment. The new hash value h equals the sum of the original hash value and the hash of x. Because of the additive homomorphism, $Hash(x_0 + x) = Hash(x_0) + Hash(x)$ holds where x_0 is the preimage for the original hash value. So, an intermediary can obtain x_0 by simple arithmetic when later nodes reveal the preimage of $Hash(x_0) + Hash(x)$. To ensure a node always has enough time to claim coins if it pays, a node always subtracts $2 \cdot (\Delta + 1))$ from the next HTLC's time-lock. $\Delta + 1$ is the maximum time to know it has lost some money. The other $\Delta + 1$ is the maximum time to publish an HTLC on-chain. Thus, if an intermediate node pays, it learns preimages for payment and can unlock more funds using received preimages.*

In the modify phase, intermediate nodes calculate the sum of $size$ and $fees$ of all updated sub-payments. Afterward, it adds fees for itself. Then, it calls updateHTLC API to update the $size$ and $fees$ of the former payment. Such a procedure ensures that the sum of updated HTLCs with later nodes is still smaller than the sum of HTLCs with former nodes. Because the time-lock T and conditions are not changed in the modify phase, we can conclude that an honest intermediate node never loses coins in the modify and complete phase too.

Because intermediate nodes never lose money during the whole process, we can conclude that the balance security of intermediate nodes is ensured in JustForward.

6.4 Atomicity

Lemma 4 *JustForward satisfies atomicity of definition 4.*

Proof of lemma *Atomicity means if an honest sender loses any coins, it can get a valid receipt that shows it has paid the promised money. In JustForward, the capacity of the sender decreases only when other parties call cPay-unlock API to obtain money from the sender. If this call is valid, API sends $(cPay\text{–}unlocked, pid_0, x_S, x_R)$ to the sender. So, a sender learns preimages of payment if it loses money because of this payment. The initialization process of payments ensures that an honest sender locks its funds only after it has received a signed receipt. If a sender loses money, it knows the preimages of the signed receipt. According to equation 4.1, the receipt kept by the sender is valid if it loses any money. So, atomicity is satisfied in JustForward.*

6.5 Unlinkability

Unlinkability means it is impossible to link a sub-payment to a specific payment. In this thesis, side channel is not considered because of its complexity. In the protocol with unlinkability, we use the same method as splitting locally [10] to decide the hashes and preimages. The forward phase, complete phase, and error handling of our protocol are almost the same as splitting locally except the existence of fees and redundancy. Redundancy makes the payment size variable which makes it even harder to link a sub-payment with another payment. Routing fees of payments are included in the payment size and divided into several parts as the split of payments. So, routing fees do not leak more information than splitting locally. Revoke fees of payments are carried in a separate transaction. One may argue that a party can link a sub-payment to another payment by calculating the revoke fee rate. However, the revoke fee rate changes after each forwarding because of the Sybil-proof fee policy. Only if a party knows the path from the sender to itself and each party uses the same parameters to decide fees, can this party calculate the revoke fee rate of the sender. In local routing, intermediary is not able to obtain such information. So, revoke fees also do not leak more information than splitting locally. In the modify phase, intermediaries only modify the payment size and keep other attributes unchanged. So, we can conclude that no information is leaked in the modify phase. To sum up, the same unlinkability as splitting locally is satisfied in JustForward.

7 Evaluation

In this chapter, we evaluate the performance of JustForward. We design different experiments to study the effectiveness and scalability of JustForward. We also compare JustForward with other related works to investigate JustForward's advantages and shortcomings.

In section 7.1, we explain the implementation of our simulator. Then, we describe the dataset, metric, and purposes of different setups. In the last section, results are presented and analyzed.

7.1 Implementation of simulator

Our simulator [36] is built upon Stefanie Roos's Payment Routing simulator[33]. Stefanie Roos's simulator implemented some mainstream routing algorithms like SpeedyMurmurs and splitting locally. An executor is also implemented in this simulator that enables us to run different algorithms with the same setup. Communication of nodes and cryptographic operations are not considered, since they have little influence on routing algorithms' performance.

Initially, we extended the original routing algorithm executor to a concurrent version using a way similar to State-Dependent Processing [29]. The old executor routes and forwards payments one by one which can't reflect the real performance. To improve it, we created a time queue and assigned an ID for each payment. IDs of payments are stored in the time queue based on payments' execution time. Rather than handle payments one by one, multiple concurrent payments are routed step by step in our executor. At first, a start time is assigned to each payment. Payments' IDs are stored in the time queue based on their start time. Then, the time queue is iterated from 0 to the maximum time. At a specific time point, the executor takes IDs from the time queue and obtains the corresponding routing status. Payments' routing status is packaged by a data structure that includes multiple sub-payments. Sub-payments are also packaged by a class that includes the current node, previous nodes, payment size, and receiver. After obtaining a payment's routing status, we can execute corresponding routing algorithms and update the routing status. When a payment fails or succeeds, we update the statistics. Otherwise, we store this payment's ID in the time queue again. We set the delay of payment forwarding to 1 round by default. The delay can also be increased by adjusting the corresponding parameter.

In our protocol, the sender can forward payment with a larger amount. To simulate it, the executor multiplies the payment size by $1 + redundancy$ initially. Intermediate nodes can decrease the payment size if they don't have enough funds. Receiver can also revoke some sub-payments to adjust the payment size. We simulate the balance changes of revoked payments by forwarding payments in the opposite direction. If

an intermediate node wants to revoke a sub-payment or decrease a payment's size, it simply sends a payment of the same size from itself to the sender among the same route. To determine whether a payment is failed, a size check mechanism is introduced. After each execution, the sizes of sub-payments are summed. If the sum is smaller than the real payment size, the executor concludes that this payment fails and revokes it.

We also implemented a new routing algorithm for payments with redundancy. Other algorithms stop routing when there are some failed sub-payments. Our algorithm adjusts the payment size and continues to forward this payment in this case. At the same time, the Waterfilling algorithm is used to balance the capacities of different channels in our routing algorithm.

The Sybil-proof fee policy is implemented inside our executor. We added a variable to the data structure of payments. This variable represents the remaining fees for a node. Intermediate nodes charge fees from the remaining fees and forward the updated remaining fees to other nodes. In a real PCN, different channels may have different fee rates. For simplicity, we assume the same fee rate is used by every channel. In a real PCN, the receiver needs to choose those sub-payments with more remaining fees. In our case, an earlier arrival time means a shorter distance. Also, payments that go through fewer nodes hold more remaining fees because all channels use the same fee rate. So, the receiver can choose those earlier sub-payments safely in our simulator. To make the fee policy compatible with the revoke mechanism, we also implemented a fee revoke mechanism that revokes fees during the revoking of payments.

7.2 Dataset

We evaluated different protocols and algorithms on a real-world Lightning dataset and a randomly generated dataset.

The Lightning Network dataset was a snapshot of the Lightning Network from March 1, 2020. We deleted disconnected nodes and channels without capacity. Then, we obtained a graph with 6329 nodes and 32629 channels. Figure 7.1 shows the distribution of nodes' degree. The capacity of each channel is also obtained from the snapshot. As shown in figure 7.2, most channels' capacity is lower than 1000000 satoshis. There are also several channels with an extremely high capacity. Payments' sender and receiver are chosen from all nodes randomly. We don't use the real transaction data because it is unavailable in the Lightning Network. For the payment size, we set a payment size rate and multiply the sender's total capacity by this rate to get the actual payment size.

To simulate the size of the Lightning Network in the future, we use the Barabasi-Albert (BA) model [2] to generate the topology of network. BA graph is a scale-free model that means only a few nodes have a high degree and other nodes' follows a power law. Several systems like World Wide Web and social networks are thought to be approximately scale-free. We assume the topology of the Lightning Network will also approximate a scale-free graph as its size grows. Thus, we use the BA model to generate a random graph with 10000 nodes where each new node is connected to 6

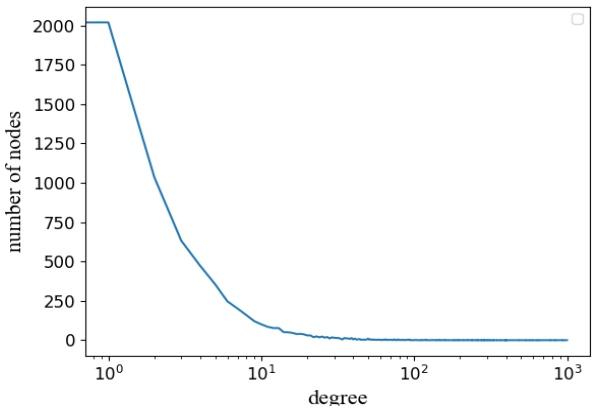

Figure 7.1: Degree distribution of the Lightning Network

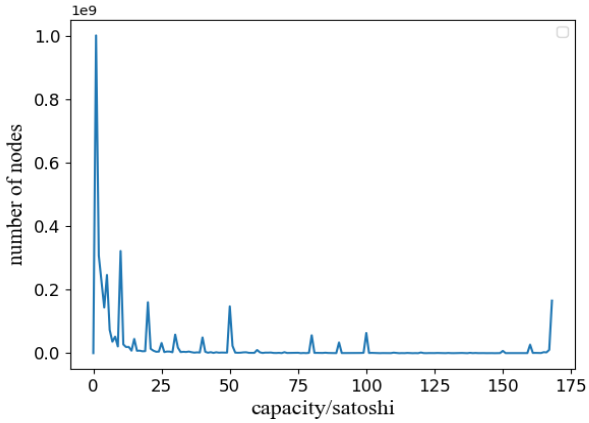

Figure 7.2: Capacity distribution of the Lightning Network

existing nodes. Capacities of channels are exponentially distributed with an average value of 200 because the distribution of channels' capacities in the Lightning Network is roughly exponential if we ignore those spikes. Payments' sender and receiver are also randomly chosen from all nodes.

7.3 Performance metric

Success rate: It is the most important metric for PCNs. It equals the percentage of successful payments.

Depleted channels: It is the number of depleted channels. In the Lightning Network work, channels with capacities lower than 100 satoshis are considered depleted. In the random graph, channels with capacities lower than one satoshi are considered depleted because of channels' relatively low initial capacities. Depleted channels can reflect the status of a PCN. In a PCN with a large number of depleted channels,

payments are more likely to be failed because of insufficient capacity.

Occupied funds: HTLCs lock channels' funds for a period of time which may influence PCNs' liquidity. We use occupied funds to denote locked funds of the whole PCN.

7.4 Setup

In our simulations, results are averaged over 10 runs to overcome the impact of randomness. The delay of payment forwarding is 1 round by default. We used Interdimensional SpeedyMurmurs with 5 spanning trees as our distance function, since splitting locally[10] shows that more than 5 trees have little influence. For the splitting locally protocol, we used the Split By Distance algorithm that has the best performance[10]. The number of sub-payments for Boomerang is set to 100 since Boomerang achieves the highest success rate with 100 sub-payments [5].

At first, we designed an experiment to evaluate the influence of payment size. For the Lightning Network, the payment size rate was $\{0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0, 9\}$. For the random graph, the expected value of the payment size was $\{50, 100, 150, 200, 250, 300, 350, 400, 450\}$. In this experiment, 100000 payments are simulated in 1000 seconds.

Then, we studied how the success rate changes over time and the reasons behind it. We simulated 300000 payments in 3000 seconds and logged the success rate, depleted channels, and occupied funds. In this experiment, the payment size rate of the Lightning Network was set to 0.4 and the payment size of the random graph was 240.

At last, we tried to figure out the impact of redundancy. Ideally, a higher redundancy results in a higher success rate. However, a higher redundancy consumes more capacities too and may have a negative effect on the success rate. We designed a situation with a large number of ongoing payments to amplify the negative effect of redundancy. We set the delay of payment forwarding to 10 rounds and simulated 400000 payments in 10000 seconds to create a high-contention situation. The payment size in the random graph was 240 while the payment size rate of the Lightning Network was 0.2. We changed the redundancy from 1 to 5 and logged their results.

7.5 Results

We denote JustForward as R and Boomerang as B. The number behind R or B is used to identify the redundancy of payments.

Payment size experiment: The results of the random graph in Figure 7.3 are quite straightforward. Hop-distance based shortest path routing algorithm has the worst performance. All other algorithms perform similarly when the payment size is low (25 percent of the expected channel capacity). JustForward's advantage becomes larger when the payment size gets larger since the probability of failed sub-payments increases with the increase of payment size. When the payment size is 450, the success rate of our protocol is about twice of SplitClosest. Compared with Boomerang, we also achieve a similar success rate with a lower redundancy.

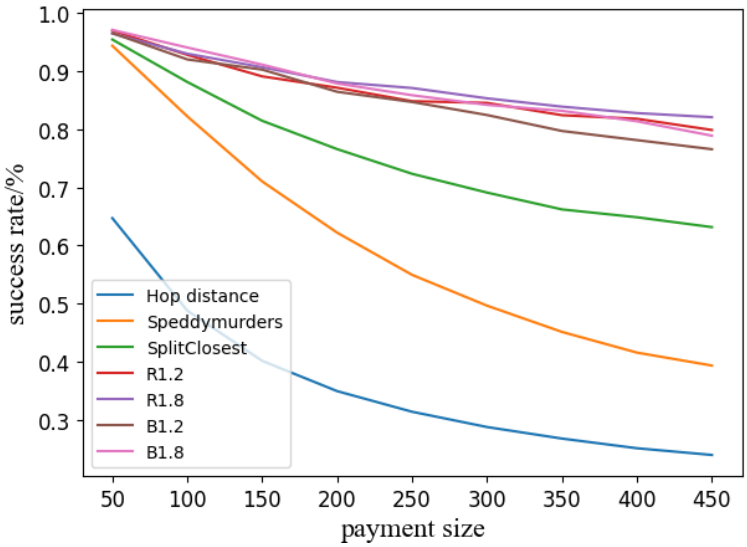

Figure 7.3: Success rate of the random graph with different payment size

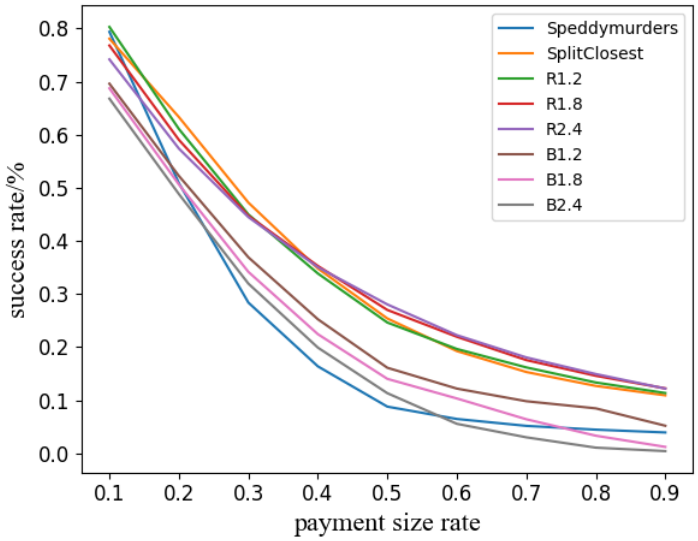

Figure 7.4: Success rate of the Lightning Network with different payment size

Results are more complicated in the Lightning Network. Initially, SplitClosest has the best performance. When the payment size is larger than 0.4, JustForward outperforms other algorithms. Algorithms' differences in the Lightning Network are smaller than in the random graph. A possible explanation is the topology of different

datasets. In the random graph, each node has at least six channels. In the Lightning Network, most nodes only have 1 channel that makes the Lightning Network has worse connectivity. Thus, most of the payments in the Lightning Network failed because of the topology rather than the routing algorithms. Boomerang performs badly in the Lightning Network because it does not allow sub-payments to be split further. In the random graph, the performance of Boomerang is acceptable because the random graph has more undepleted channels and 100 sub-payments is enough to make payments successful.

In this experiment, JustForward has a better performance when the payment size is large. This is because our redundancy mechanism makes the routing of payments tolerant to depleted channels. Our capacity-balanced routing algorithm also can decrease the number of depleted channels and result in a higher success rate.

Changes of success rate: For the Lightning Network, SplitClosest and our protocol have a similar success rate initially. Then, JustForward starts to outperform SplitClosest as time goes by. Figure 7.6 shows that our protocol has a higher performance because of less depleted channels. In the first 2000 seconds, the number of depleted channels in SplitClosest increases continuously. On contrary, the depleted channels of our protocol only increase slightly. In the end, the number of SplitClosest's depleted channels is about 6 times higher than our protocol that results in more failed payments. Figure 7.7 shows our protocol occupies much more funds than other protocols. Firstly, JustForward needs to handle payments with a larger size because of the redundancy. Secondly, split locally stop the forwarding of payment when there is one failed sub-payment. JustForward still forwards remaining sub-payments to other nodes because of the redundancy. So, JustForward potentially needs to handle more ongoing payments.

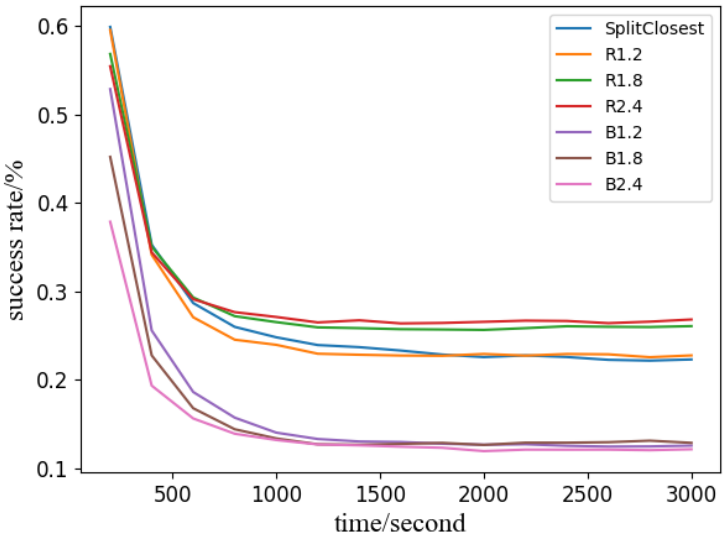

Figure 7.5: Success rate of the Lightning Network at different times

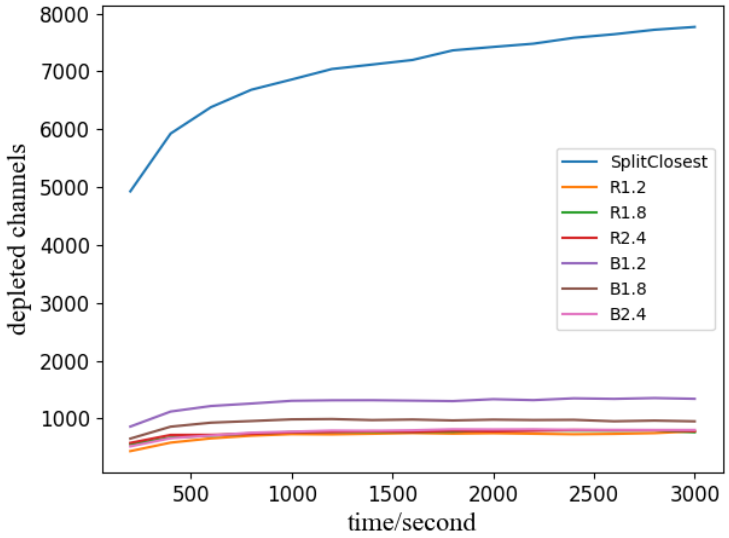

Figure 7.6: Depleted channels in the Lightning Network

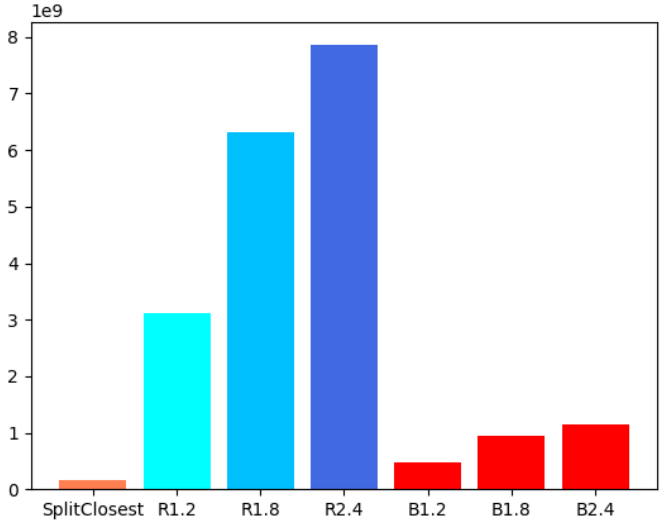

Figure 7.7: Occupied funds of the Lightning Network

In the random graph, the number of depleted channels equals 0 all the time. This explains why the success rate does not have a significant change in figure 7.8. Similarly, JustForward occupies more funds than SplitClosest and Boomerang in figure 7.9.

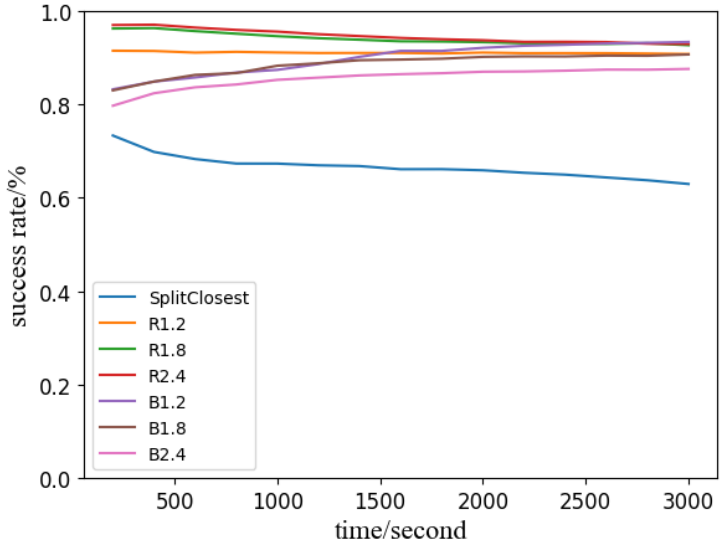

Figure 7.8: Success rate of the random graph at different times

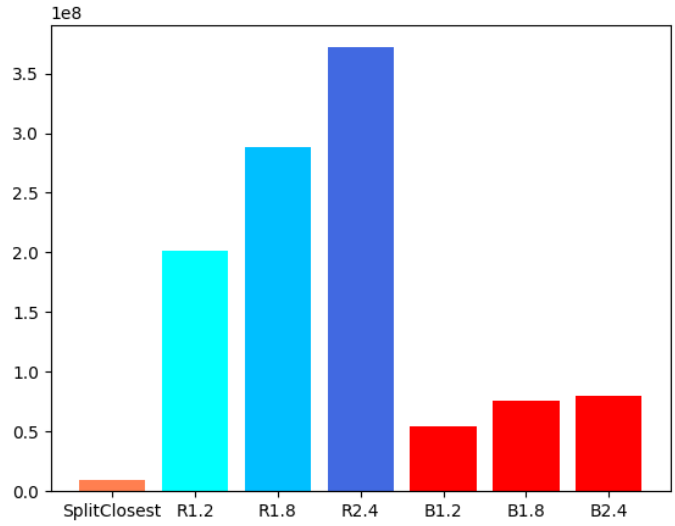

Figure 7.9: Occupied funds of the random graph

In this experiment, JustForward occupies more funds to obtain a higher success rate. It exchanges the liquidity of PCNs and routing fees for a higher success rate. The comparison between JustForward and Boomerang also shows that the flexibility of JustForward makes it achieve a higher success rate too. More importantly, JustForward gives participants more options to forward payments. It would be interesting to change the redundancy value depending on the concurrency in PCNs. For example, partici-

pants can forward payments without redundancy when there are only a few ongoing payments in a PCN.

Redundancy experiment: Figure 7.10 and figure 7.11 depicts the impact of redundancy. In both figures, the success rate increases initially and drops later. The increased redundancy can make payments resistant to more depleted channels. Thus, the success rate increases as the redundancy increase initially. However, a higher redundancy also occupies more funds which results in more depleted channels and a smaller success rate. This explains why the success rate starts to drop and even becomes lower than the initial success rate later. So, a higher redundancy does not always result in a better performance.

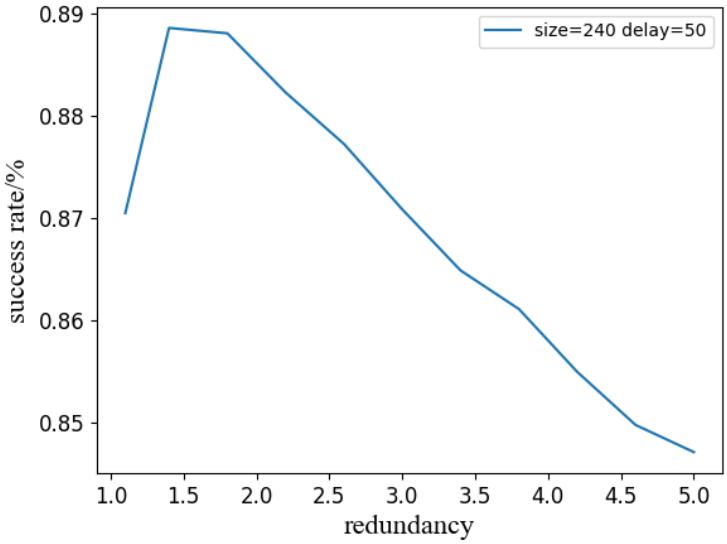

Figure 7.10: Success rate of the random graph with different redundancy

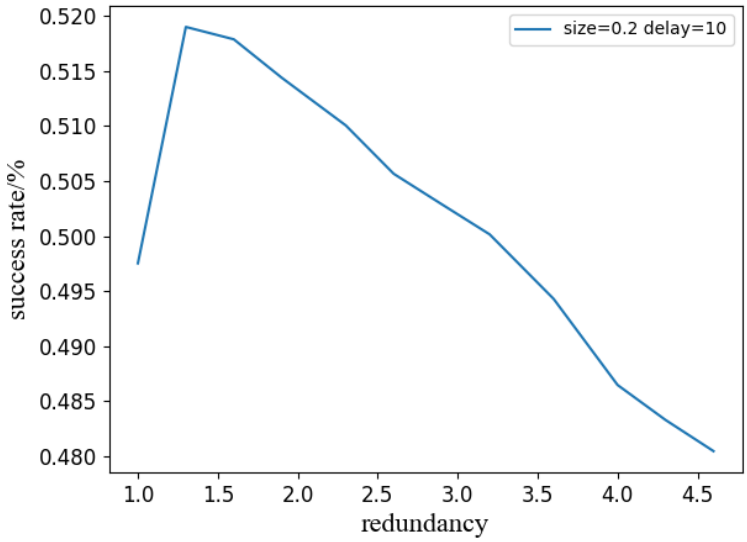

Figure 7.11: Success rate of the Lightning Network with different redundancy

8 Conclusion and future works

Payment channel networks have a huge potential to increase the transaction speed and throughput of blockchains. However, payments can easily fail with existing payment protocols because of depleted channels. Boomerang improves it by adding redundancy to payments. But it increases the number of sub-payments and damages intermediaries' benefits at the same time. Thus, we propose our research question: What is the method to use redundancy to increase the success rate of payments while achieving balance security, scalability, and fairness for intermediate nodes?

8.1 Conclusion

To realize a payment protocol with redundancy, we proposed a payment update mechanism at first. We built it upon the transaction revoke mechanism of the Lightning Network. The payer sends an updated transaction to the payee. Then, the payee revokes the old transaction and signs the updated transaction. With this mechanism, intermediaries and the receiver are able to decrease the payment size if the sender forwards a payment with redundancy.

Combining the payment update mechanism and splitting locally, we got JustForward, a payment protocol with high flexibility. With JustForward, intermediaries can split payments and adjust payment size according to their needs. Scalability is also satisfied in JustForward since payments are split by intermediaries during routing rather than by the sender in advance.

To incentivize all participants who have locked ftheir funds, we introduced revoke fees for revoked payments. With JustForward, intermediaries who forward a revoked payment can still obtain some rewards. Afterward, we defined the properties of a fair fee policy and integrated such a fee policy into our protocol.

We also designed a routing algorithm to route payments with redundancy. It can adjust the payment size during routing. Furthermore, we implemented a Waterfilling algorithm to balance different channels' capacities and decrease the number of depleted channels.

In the security analysis of JustForward, we defined the finality, balance security, and atomicity of a payment protocol. Then, we proved that JustForward satisfies the defined properties.

In the performance analysis, we benchmarked JustForward and splitting locally in different situations. We found that JustForward has a higher success rate when there are numerous concurrent payments or the payment size is huge. We also found that an extremely high redundancy can reduce the success rate because it occupies more funds and damages the liquidity of PCNs.

8.2 Future work

8.2.1 Routing

Currently, we use an intuitive method to adjust the payment size. Our routing algorithm decreases the payment size only when a node does not have enough capacity to forward payment. A better routing algorithm may adjust the payment size more flexibly. For example, it can decrease the payment size of large payments and forward small payments without split or size adjustment. Also, the routing information of past payments is not utilized in our routing algorithm. A better routing algorithm may forward fewer funds to channels with a low success rate in the past.

8.2.2 Protocol

In JustForward, we use two transactions to forward payment. The usage of one transaction is only to decide the amount of revoke fees. A possible improvement is to integrate the revoke fees into the normal payment and decide the amount of revoke fees by message exchange. Such an improvement is helpful to make the payment process easier and decrease nodes' overhead.

Another possible improvement is to refine the HTLC to make it support partial payments. In JustForward, we need extra rounds to modify HTLCs from the receiver to the sender that results in a higher delay. If there is a protocol that allows HTLCs to be completed partially, updates of HTLCs are not necessary anymore.

8.2.3 Privacy

In this thesis, the privacy of participants is not analyzed. However, our protocol has the potential to realize a privacy-preserving routing since it is built upon split locally and SpeedyMurmurs that can keep payments private. In the future, we may define our privacy goals formally and analyze the privacy of our protocol based on these goals.

Bibliography

[1] B. AG, *Raiden network*, 2019. [Online]. Available: https://raiden.network/.

[2] R. Albert and A.-L. Barabási, "Statistical mechanics of complex networks," *Rev. Mod. Phys.*, vol. 74, pp. 47–97, 1 Jan. 2002. DOI: 10.1103/RevModPhys.74.47. [Online]. Available: https://link.aps.org/doi/10.1103/RevModPhys.74.47.

[3] L. Aumayr, O. Ersoy, A. Erwig, *et al.*, "Generalized bitcoin-compatible channels," *IACR Cryptol. ePrint Arch.*, p. 476, 2020. [Online]. Available: https://eprint.iacr.org/2020/476.

[4] L. Aumayr, P. Moreno-Sanchez, A. Kate, and M. Maffei, "Blitz: Secure multi-hop payments without two-phase commits," in *30th USENIX Security Symposium, USENIX Security 2021, August 11-13, 2021*, M. Bailey and R. Greenstadt, Eds., USENIX Association, 2021, pp. 4043–4060. [Online]. Available: https://www.usenix.org/conference/usenixsecurity21/presentation/aumayr.

[5] V. K. Bagaria, J. Neu, and D. Tse, "Boomerang: Redundancy improves latency and throughput in payment-channel networks," in *Financial Cryptography and Data Security - 24th International Conference, FC 2020, Kota Kinabalu, Malaysia, February 10-14, 2020 Revised Selected Papers*, J. Bonneau and N. Heninger, Eds., ser. Lecture Notes in Computer Science, vol. 12059, Springer, 2020, pp. 304–324. DOI: 10.1007/978-3-030-51280-4_17. [Online]. Available: https://doi.org/10.1007/978-3-030-51280-4%5C_17.

[6] K. M. Chandy and J. Misra, "Distributed computation on graphs: Shortest path algorithms," *Commun. ACM*, vol. 25, no. 11, pp. 833–837, 1982. DOI: 10.1145/358690.358717. [Online]. Available: https://doi.org/10.1145/358690.358717.

[7] T. Cheung, "Graph traversal techniques and the maximum flow problem in distributed computation," *IEEE Trans. Software Eng.*, vol. 9, no. 4, pp. 504–512, 1983. DOI: 10.1109/TSE.1983.234958. [Online]. Available: https://doi.org/10.1109/TSE.1983.234958.

[8] S. Dziembowski, L. Eckey, S. Faust, and D. Malinowski, "PERUN: virtual payment channels over cryptographic currencies," *IACR Cryptol. ePrint Arch.*, p. 635, 2017. [Online]. Available: http://eprint.iacr.org/2017/635.

[9] S. Dziembowski and P. Kedzior, "Ethna: Channel network with dynamic internal payment splitting," *IACR Cryptol. ePrint Arch.*, p. 166, 2020. [Online]. Available: https://eprint.iacr.org/2020/166.

[10] L. Eckey, S. Faust, K. Hostáková, and S. Roos, "Splitting payments locally while routing interdimensionally," *IACR Cryptol. ePrint Arch.*, p. 555, 2020. [Online]. Available: https://eprint.iacr.org/2020/555.

[11] O. Ersoy, Z. Ren, Z. Erkin, and R. L. Lagendijk, "Transaction propagation on permissionless blockchains: Incentive and routing mechanisms," in *Crypto Valley Conference on Blockchain Technology, CVCBT 2018, Zug, Switzerland, June 20-22, 2018*, IEEE, 2018, pp. 20–30. DOI: 10.1109/CVCBT.2018.00008. [Online]. Available: https://doi.org/10.1109/CVCBT.2018.00008.

[12] I. Eyal, A. E. Gencer, E. G. Sirer, and R. van Renesse, "Bitcoin-ng: A scalable blockchain protocol," in *13th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2016, Santa Clara, CA, USA, March 16-18, 2016*, K. J. Argyraki and R. Isaacs, Eds., USENIX Association, 2016, pp. 45–59. [Online]. Available: https://www.usenix.org/conference/nsdi16/technical-sessions/presentation/eyal.

[13] *Global cryptocurrency charts*, https://coinmarketcap.com/charts/, 2022.

[14] A. V. Goldberg and R. E. Tarjan, "A new approach to the maximum-flow problem," *J. ACM*, vol. 35, no. 4, pp. 921–940, 1988. DOI: 10.1145/48014.61051. [Online]. Available: https://doi.org/10.1145/48014.61051.

[15] M. Green and I. Miers, "Bolt: Anonymous payment channels for decentralized currencies," in *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security, CCS 2017, Dallas, TX, USA, October 30 - November 03, 2017*, B. Thuraisingham, D. Evans, T. Malkin, and D. Xu, Eds., ACM, 2017, pp. 473–489. DOI: 10.1145/3133956.3134093. [Online]. Available: https://doi.org/10.1145/3133956.3134093.

[16] L. Gudgeon, P. Moreno-Sanchez, S. Roos, P. McCorry, and A. Gervais, "Sok: Layer-two blockchain protocols," in *Financial Cryptography and Data Security - 24th International Conference, FC 2020, Kota Kinabalu, Malaysia, February 10-14, 2020 Revised Selected Papers*, J. Bonneau and N. Heninger, Eds., ser. Lecture Notes in Computer Science, vol. 12059, Springer, 2020, pp. 201–226. DOI: 10.1007/978-3-030-51280-4_12. [Online]. Available: https://doi.org/10.1007/978-3-030-51280-4%5C_12.

[17] G. Karame, "On the security and scalability of bitcoin's blockchain," in *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security, Vienna, Austria, October 24-28, 2016*, E. R. Weippl, S. Katzenbeisser, C. Kruegel, A. C. Myers, and S. Halevi, Eds., ACM, 2016, pp. 1861–1862. DOI: 10.1145/2976749.2976756. [Online]. Available: https://doi.org/10.1145/2976749.2976756.

[18] R. Khalil and A. Gervais, "Revive: Rebalancing off-blockchain payment networks," in *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security, CCS 2017, Dallas, TX, USA, October 30 - November 03, 2017*, B. Thuraisingham, D. Evans, T. Malkin, and D. Xu, Eds., ACM, 2017, pp. 439–453. DOI: 10.1145/3133956.3134033. [Online]. Available: https://doi.org/10.1145/3133956.3134033.

[19] A. Kiayias, A. Russell, B. David, and R. Oliynykov, "Ouroboros: A provably secure proof-of-stake blockchain protocol," in *Advances in Cryptology - CRYPTO 2017 - 37th Annual International Cryptology Conference, Santa Barbara, CA, USA, August 20-24, 2017, Proceedings, Part I*, J. Katz and H. Shacham, Eds., ser. Lecture Notes in Computer Science, vol. 10401, Springer, 2017, pp. 357–388. DOI: 10.1007/978-3-319-63688-7_12. [Online]. Available: https://doi.org/10.1007/978-3-319-63688-7%5C_12.

[20] *Lightning network daemon*, https://github.com/lightningnetwork/lnd, 2022.

[21] M. Lokhava, G. Losa, D. Mazières, *et al.*, "Fast and secure global payments with stellar," in *Proceedings of the 27th ACM Symposium on Operating Systems Principles, SOSP 2019, Huntsville, ON, Canada, October 27-30, 2019*, T. Brecht and C. Williamson, Eds., ACM, 2019, pp. 80–96. DOI: 10.1145/3341301.3359636. [Online]. Available: https://doi.org/10.1145/3341301.3359636.

[22] L. Luu, V. Narayanan, C. Zheng, K. Baweja, S. Gilbert, and P. Saxena, "A secure sharding protocol for open blockchains," in *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security, Vienna, Austria, October 24-28, 2016*, E. R. Weippl, S. Katzenbeisser, C. Kruegel, A. C. Myers, and S. Halevi, Eds., ACM, 2016, pp. 17–30. DOI: 10.1145/2976749.2978389. [Online]. Available: https://doi.org/10.1145/2976749.2978389.

[23] G. Malavolta, P. Moreno-Sanchez, A. Kate, and M. Maffei, "Silentwhispers: Enforcing security and privacy in decentralized credit networks," in *24th Annual Network and Distributed System Security Symposium, NDSS 2017, San Diego, California, USA, February 26 - March 1, 2017*, The Internet Society, 2017. [Online]. Available: https://www.ndss-symposium.org/ndss2017/ndss-2017-programme/silentwhispers-enforcing-security-and-privacy-decentralized-credit-networks/.

[24] G. Malavolta, P. Moreno-Sanchez, C. Schneidewind, A. Kate, and M. Maffei, "Anonymous multi-hop locks for blockchain scalability and interoperability," in *26th Annual Network and Distributed System Security Symposium, NDSS 2019, San Diego, California, USA, February 24-27, 2019*, The Internet Society, 2019. [Online]. Available: https://www.ndss-symposium.org/ndss-paper/anonymous-multi-hop-locks-for-blockchain-scalability-and-interoperability/.

[25] A. Miller, I. Bentov, S. Bakshi, R. Kumaresan, and P. McCorry, "Sprites and state channels: Payment networks that go faster than lightning," in *Financial Cryptography and Data Security - 23rd International Conference, FC 2019, Frigate Bay, St. Kitts and Nevis, February 18-22, 2019, Revised Selected Papers*, I. Goldberg and T. Moore, Eds., ser. Lecture Notes in Computer Science, vol. 11598, Springer, 2019, pp. 508–526. DOI: 10.1007/978-3-030-32101-7_30. [Online]. Available: https://doi.org/10.1007/978-3-030-32101-7%5C_30.

[26] U. Mukhopadhyay, A. Skjellum, O. Hambolu, J. Oakley, L. Yu, and R. R. Brooks, "A brief survey of cryptocurrency systems," in *14th Annual Conference on Privacy, Security and Trust, PST 2016, Auckland, New Zealand, December 12-14, 2016*, IEEE, 2016, pp. 745–752. DOI: 10.1109/PST.2016.7906988. [Online]. Available: https://doi.org/10.1109/PST.2016.7906988.

[27] J. A. Nelder and R. Mead, "A simplex method for function minimization," *Comput. J.*, vol. 7, no. 4, pp. 308–313, 1965. DOI: 10.1093/comjnl/7.4.308. [Online]. Available: https://doi.org/10.1093/comjnl/7.4.308.

[28] O. Osuntokun, *Amp: Atomic multi-path payments over lightning*, 2018-02-06.

[29] N. Papadis and L. Tassiulas, "State-dependent processing in payment channel networks for throughput optimization," *CoRR*, vol. abs/2103.17207, 2021. arXiv: 2103.17207. [Online]. Available: https://arxiv.org/abs/2103.17207.

[30] S. Park, A. Kwon, G. Fuchsbauer, P. Gazi, J. Alwen, and K. Pietrzak, "Spacemint: A cryptocurrency based on proofs of space," in *Financial Cryptography and Data Security - 22nd International Conference, FC 2018, Nieuwpoort, Curacao, February 26 - March 2, 2018, Revised Selected Papers*, S. Meiklejohn and K. Sako, Eds., ser. Lecture Notes in Computer Science, vol. 10957, Springer, 2018, pp. 480–499. DOI: 10.1007/978-3-662-58387-6_26. [Online]. Available: https://doi.org/10.1007/978-3-662-58387-6%5C_26.

[31] J. Poon and T. Dryja, *The bitcoin lightning network: Scalable o -chain instant payments*, 2016. [Online]. Available: https://www.bitcoinlightning.com/wp-content/uploads/2018/03/lightning-network-paper.pdf.

[32] S. Rahimpour and M. Khabbazian, "Spear: Fast multi-path payment with redundancy," in *AFT '21: 3rd ACM Conference on Advances in Financial Technologies, Arlington, Virginia, USA, September 26 - 28, 2021*, F. Baldimtsi and T. Roughgarden, Eds., ACM, 2021, pp. 183–191. DOI: 10.1145/3479722.3480997. [Online]. Available: https://doi.org/10.1145/3479722.3480997.

[33] S. Roos, *Paymentrouting*, https://github.com/stef-roos/PaymentRouting, 2021.

[34] S. Roos, P. Moreno-Sanchez, A. Kate, and I. Goldberg, "Settling payments fast and private: Efficient decentralized routing for path-based transactions," in *25th Annual Network and Distributed System Security Symposium, NDSS 2018, San Diego, California, USA, February 18-21, 2018*, The Internet Society, 2018. [Online]. Available: http://wp.internetsociety.org/ndss/wp-content/uploads/sites/25/2018/02/ndss2018%5C_09-3%5C_Roos%5C_paper.pdf.

[35] "SCP: A computationally-scalable byzantine consensus protocol for blockchains," *IACR Cryptol. ePrint Arch.*, p. 1168, 2015, Withdrawn. [Online]. Available: http://eprint.iacr.org/2015/1168.

[36] Y. Shen, *Payment routing simulator for justforward*, https://github.com/tokisamu/PaymentRouting, 2022.

[37] V. Sivaraman, S. B. Venkatakrishnan, M. Alizadeh, G. Fanti, and P. Viswanath, "Routing cryptocurrency with the spider network," in *Proceedings of the 17th ACM Workshop on Hot Topics in Networks, HotNets 2018, Redmond, WA, USA, November 15-16, 2018*, ACM, 2018, pp. 29–35. DOI: 10.1145/3286062.3286067. [Online]. Available: https://doi.org/10.1145/3286062.3286067.

[38] G. D. Stasi, S. Avallone, R. Canonico, and G. Ventre, "Routing payments on the lightning network," in *IEEE International Conference on Internet of Things (iThings) and IEEE Green Computing and Communications (GreenCom) and IEEE Cyber, Physical and Social Computing (CPSCom) and IEEE Smart Data (SmartData), iThings/GreenCom/CPSCom/SmartData 2018, Halifax, NS, Canada, July 30 - August 3, 2018*, IEEE, 2018, pp. 1161–1170. DOI: 10.1109/Cybermatics_2018.2018.00209. [Online]. Available: https://doi.org/10.1109/Cybermatics%5C_2018.2018.00209.

[39] C. A. Sunshine, "Source routing in computer networks," *Comput. Commun. Rev.*, vol. 7, no. 1, pp. 29–33, 1977. DOI: 10.1145/1024853.1024855. [Online]. Available: https://doi.org/10.1145/1024853.1024855.

[40] S. Tripathy and S. K. Mohanty, "MAPPCN: multi-hop anonymous and privacy-preserving payment channel network," in *Financial Cryptography and Data Security - FC 2020 International Workshops, AsiaUSEC, CoDeFi, VOTING, and WTSC, Kota Kinabalu, Malaysia, February 14, 2020, Revised Selected Papers*, M. Bernhard, A. Bracciali, L. J. Camp, *et al.*, Eds., ser. Lecture Notes in Computer Science, vol. 12063, Springer, 2020, pp. 481–495. DOI: 10.1007/978-3-030-54455-3_34. [Online]. Available: https://doi.org/10.1007/978-3-030-54455-3%5C_34.

[41] I. Tsabary, M. Yechieli, A. Manuskin, and I. Eyal, "MAD-HTLC: because HTLC is crazy-cheap to attack," in *42nd IEEE Symposium on Security and Privacy, SP 2021, San Francisco, CA, USA, 24-27 May 2021*, IEEE, 2021, pp. 1230–1248. DOI: 10.1109/SP40001.2021.00080. [Online]. Available: https://doi.org/10.1109/SP40001.2021.00080.

[42] P. Wang, H. Xu, X. Jin, and T. Wang, "Flash: Efficient dynamic routing for offchain networks," in *Proceedings of the 15th International Conference on Emerging Networking Experiments And Technologies, CoNEXT 2019, Orlando, FL, USA, December 09-12, 2019*, A. Mohaisen and Z. Zhang, Eds., ACM, 2019, pp. 370–381. DOI: 10.1145/3359989.3365411. [Online]. Available: https://doi.org/10.1145/3359989.3365411.

[43] R. Yu, G. Xue, V. T. Kilari, D. Yang, and J. Tang, "Coinexpress: A fast payment routing mechanism in blockchain-based payment channel networks," in *27th International Conference on Computer Communication and Networks, ICCCN 2018, Hangzhou, China, July 30 - August 2, 2018*, IEEE, 2018, pp. 1–9. DOI: 10.1109/ICCCN.2018.8487351. [Online]. Available: https://doi.org/10.1109/ICCCN.2018.8487351.

[44] Y. Zhang, D. Yang, and G. Xue, "Cheapay: An optimal algorithm for fee minimization in blockchain-based payment channel networks," in *2019 IEEE International Conference on Communications, ICC 2019, Shanghai, China, May 20-24, 2019*, IEEE, 2019, pp. 1–6. DOI: 10.1109/ICC.2019.8761804. [Online]. Available: https://doi.org/10.1109/ICC.2019.8761804.

[45] Z. Zheng, S. Xie, H. Dai, X. Chen, and H. Wang, "An overview of blockchain technology: Architecture, consensus, and future trends," in *2017 IEEE International Congress on Big Data, BigData Congress 2017, Honolulu, HI, USA, June 25-30, 2017*, G. Karypis and J. Zhang, Eds., IEEE Computer Society, 2017, pp. 557–564. DOI: 10.1109/BigDataCongress.2017.85. [Online]. Available: https://doi.org/10.1109/BigDataCongress.2017.85.

Appendix A

9

Details of the protocol with unlinkability

Meaning of variables and functions

$a \longrightarrow b$: send message a to b
F: ideal fuctionality
S: sender's ID
R: receiver's ID
G: the topology of PCN
C_i: node i's balance in each channel
$x_S \ and \ x_R$: sampled preimages for HTLCs
$H(x)$: an additive homomorphic hash function
$Sign_k(m)$: sign the message m using key k
$Vrfy_k(m, Sig)$: verify the signature Sig of message m using key k
T: the latest time to complete a payment
C_{revoke}: the ratio of routing fees to revoke fees
$C_{redundant}(\geq 1)$: redundancy of payments
$fee^{routing}$: fees if the whole payment succeeds
fee^{revoke}: fees if the whole payment is revoked
Δ: time needed to publish a payment on chain
$[k]$: Natural numbers from 1 to k
$Route(size, I, R)$: For the payment from I to R with size $size$, decide how to split it and what is the next node
$Fee(totalfees)$: decide fees for current node based on a fee policy
$Reschedule_G(candidates, size)$: choose some sub-payments from candidates to let the sum of them equal $size$, the size of those sub-payments also can be adjusted
$Enc_k(M)$: using Paillier encryption scheme to encrypt M with key k
$Dec_k(M)$: using Paillier encryption scheme to decrypt M with key k
skR: the secret key of receiver
pkR: the public key of receiver

Initialization

At first, different parties obtain necessary information and initialize internal variables.
Sender:

1: $Receive\ (G, S, R, size, fee, C_S)$
2: $out := \emptyset$ {The set of all sub-payments}

3: $sum_{pay} := 0$ {Sum of updated payment size}
4: $sum_{fee} := 0$ {Sum of updated fees}

Receiver:

1: $Receive(G, S, R, size, fee, C_R)$
2: $in := \emptyset$ {The set of received sub-payments}
3: $candidate := \emptyset$
4: sum_{pay} {Sum of received sub-payments}
5: $finish =: 0$ {Are all candidates decided}

Intermediaries:

1: $Receive(G, C_I)$
2: $fw := \emptyset$ {The set of forwarded sub-payments}
3: $cnt := \emptyset$ {Count sub-payments for each received payment}
4: $sum_{pay} := \emptyset$ {Sum of updated sub-payments' sizes}
5: $sum_{fee} := \emptyset$ {Sum of updated sub-payments' fees}
6: $old_{pay} := \emptyset$ {The initial payment size of each payment}
7: $oldFee^{routing} := \emptyset$ {The initial routing fee of each payment}
8: $oldFee^{revoke} := \emptyset$ {The initial revoke fee of each payment}

Forward phase

Sender initiates a payment:
In round t_0, the sender samples a preimage and calculate the hash value of it. Then, this hash is signed and sent to the receiver.

1: $x_S \longleftarrow \mathsf{Z}_P, h_S := H(x_S)$
2: Send $(h_S, Sign_{sk_S}(h_S))$ to the receiver

Receiver upon receiving (h_S, Sig_{h_S})**:**
If the received signature is valid, the receiver sends a signed message back.

1: **if** Sig_{h_S} is a valid signature of the sender **then**
2: $x_R \longleftarrow \mathsf{Z}_P, h_R := H(x_R)$
3: $Sig = Sign_{sk_R}(S, R, size, h_S, h_R)$
4: Send $(init, h_S, h_R, Sig)$ to the sender
5: **else**
6: $Terminate$
7: **end if**

Sender upon receiving $(init, h_S, h_R, Sig)$**:**
In round $t_0 + 2$, the sender checks the validity of the receiver's message. If it is valid, the sender tries to route and forward its payment.

1: **if** Sig is a valid signature for $(S, R, size, h_S, h_R)$ **then**
2: $T := t_0 + 2 + |V|(2 + 2 \cdot (\Delta + 1))$
3: $(e_j, size_j)_{j \in [k]} = Route_G(size \cdot C_{redundant}, S, R)$
4: **for** $j \in [k]$ **do**
5: $fee_j = size_j / size \cdot fee$

6: $pid_j \longleftarrow \{0,1\}^*$
7: $x_j \longleftarrow \mathsf{Z}_P$
8: $out := out \cup \{(e_j, size_j, fee_j, x_j, pid_j)\}$
9: $(cPay, pid_j, e_j, size_j, fee_j, fee_j \cdot C_{revoke}, h_S + H(x_j), h_R + H(x_j), T, R, Enc_{pkR}(x_j)) \longrightarrow F$
10: **end for**
11: **else**
12: $Terminate$
13: **end if**

F receives $(cPay, pid, e, size, fee_{routing}, fee_{revoke}, h_S, h_R, T, R, M)$ **from** I **or** S**:**

1: **if** I's capacity in channel $e \geq size + fee$ **then**
2: decrease I's capacity by $size + fee$
3: send $(cPaid, pid, e, size, fee^{routing}, fee^{revoke}, h_S, h_R, T, R, M)$ to the other party of channel e
4: **else**
5: $Terminate$
6: **end if**

Intermediary upon receiving
$(cPaid, pid, e, size, fee^{routing}, fee^{revoke}, h_S, h_R, T, R, M)$**:**

1: **if** $current\ round \leq t_0 + 1 + |V|$ **then**
2: $T := T - 2(\Delta + 1)$
3: $cnt[T] := 0, sum_{pay}[T] := 0, sum_{fee}[T] := 0, old_{pay}[T] := size$
4: $oldFee^{routing}[T] := fee^{routing}, oldFee^{revoke}[T] := fee^{revoke}$
5: $(e_j, size_j)_{j \in [k]} = Route_G(size, I, R)$
6: **for** $j \in [k]$ **do**
7: $fee_j^{routing} = Fee(size_j/size \cdot fee^{routing})$
8: $fee_j^{revoke} = Fee(size_j/size \cdot fee^{revoke})$
9: $pid_j \longleftarrow \{0,1\}^*$
10: $x_j \longleftarrow \mathsf{Z}_P$
11: $fw[T] := fw[T] \cup \{(e_j, size_j, fee_j^{routing}, fee_j^{revoke}, x_j, pid_j, pid)\}$
12: $cnt[T] := cnt[T] + 1$
13: $(cPay, pid_j, e_j, size_j, fee_j^{routing}, fee_j^{revoke}, h_S + H(x_j), h_R + H(x_j), T, R, M + Enc_{pkR}(x_j)) \longrightarrow F$
14: **end for**
15: **else**
16: $Terminate$
17: **end if**

Receiver upon receiving
$(cPaid, pid, e, size, fee^{routing}, fee^{revoke}, h_S, h_R, T, R, M)$**:**
The receiver calculates the sum of received sub-payments. If the sum reaches the payment size, the receiver tries to reschedule received sub-payments.

1: **if** $finish = 0$ **then**

2: $sum_{pay} := sum_{pay} + size$
3: $candidate = candidate \cup \{(pid, size, fee^{routing}, fee^{revoke}, h_S, h_R, M)\}$
4: **if** $sum_{pay} >= size$ **then**
5: $(pid_j, size_j, fee_j, h_{Sj}, h_{Rj}, M_j)_{j\in[k]} = Reschedule_G(candidate)$
6: **for** $j \in [k]$ **do**
7: $(updateHTLC, pid_j, size_j, fee_j) \longrightarrow F$
8: $in = in \cup \{(pid_j, h_{Sj}, h_{Rj}, M_j)\}$
9: **end for**
10: $finish = 1$
11: **end if**
12: **else**
13: $(updateHTLC, pid, 0, fee^{revoke}) \longrightarrow F$
14: $in = in \cup \{(pid, h_S, h_R, M)\}$
15: **end if**

Modify phase

F upon receiving $(updateHTLC, pid, size_{new}, fee_{new})$ **from** I **or** R**:**

1: **if** I has a payment with pid **then**
2: increase the capacity of the former node based on $size_{new}$ and fee_{new}
3: send $(updatedHTLC, pid_0, size_{new}, fee_{new})$ to the former node
4: **end if**

Intermediary upon receiving $(updatedHTLC, pid_0, size_{new}, fee_{new})$**:**
Intermediate node needs to make sure the updated payment size and fees are correct. If so, it waits until all sub-payments are updated. Afterwards, it tries to update the former payment

1: *Let* $u_j := (e_j, size_j, fee_j^{routing}, fee_j^{revoke}, x_j, pid_j, pid) \in fw[T] \quad s.t.\ pid_j = pid_0$
2: **if** $size_{new} > size_j \vee fee_{new} > size_{new}/size_j \cdot fee_j^{routing} + (1 - size_{new}/size_j) \cdot fee_j^{revoke}$ **then**
3: $Terminate$
4: **else**
5: $fw[T] := fw[T] \backslash \{u_j\}$
6: $fw[T] := fw[T] \cup \{(e_j, size_{new}, fee_{new}, 0, x_j, pid_j, pid)\}$
7: $cnt[T] := cnt[T] - 1$
8: $sum_{pay}[T] := sum_{pay}[T] + size_{new}$
9: $sum_{fee}[T] := sum_{fee}[T] + fee_{new}$
10: **if** $cnt[T] = 0$ **then**
11: $sum_{fee}[T] += sum_{pay}[T]/old_{pay}[T] \cdot oldFee^{routing}[T]$
12: $sum_{fee}[T] += (1 - sum_{pay}[T]/old_{pay}[T]) \cdot oldFee^{revoke}[T]$
13: $(updateHTLC, pid, sum_{pay}[T], sum_{fee}[T]) \longrightarrow F$
14: **end if**
15: **end if**

Sender upon receiving $(updatedHTLC, pid_0, size_{new}, fee_{new})$**:**
When the sender receives update requests, it counts the sum of those updated sub-payments. If the sum equals the payment size plus fees, it sends its preimage to the receiver.

1: $Let\ u_j := (e_j, size_j, fee_j, x_j, pid_j) \in out[T] \quad s.t.\ pid_j = pid_0$
2: $out := out \backslash \{u_j\}$
3: $out := out \cup \{(e_j, size_{new}, fee_{new}, pid_j)\}$
4: $sum_{pay} += size_{new}, sum_{fee} += fee_{new}, cnt -= 1$
5: **if** $sum_{pay} = size \wedge cnt = 0$ **then**
6: $\quad fee = fee/C_{redundant} + fee \cdot C_{revoke} \cdot (redundant - 1)/redundant$
7: $\quad$ **if** $sum_{pay} > size \vee sum_{fee} > fee$ **then**
8: $\quad\quad Terminate$
9: $\quad$ **else**
10: $\quad\quad$ Send x_S to the receiver
11: $\quad$ **end if**
12: **end if**

Complete phase

Receiver upon receiving x_S**:**
The receiver uses preimages to unlock funds.

1: **if** $H(x_S) = h_S$ **then**
2: $\quad$ **for** $(pid, h_S, h_R, M) \in in$ **do**
3: $\quad\quad x_{Sj} = x_S + Dec_{skR}(M), x_{Rj} = x_R + Dec_{skR}(M)$
4: $\quad\quad$ **for** $i \in [0, x_{Sj}]$ **do**
5: $\quad\quad\quad$ **if** $H((x_{Sj} + i \cdot N)\ mod\ P) = h_S$ **then**
6: $\quad\quad\quad\quad x_{Sj} = (x_{Sj} + N \cdot i)\ mod\ P$
7: $\quad\quad\quad$ **end if**
8: $\quad\quad$ **end for**
9: $\quad\quad$ **for** $i \in [0, x_{Rj}]$ **do**
10: $\quad\quad\quad$ **if** $H((x_{Rj} + i \cdot N)\ mod\ P) = h_R$ **then**
11: $\quad\quad\quad\quad x_{Rj} = (x_{Rj} + N \cdot i)\ mod\ P$
12: $\quad\quad\quad$ **end if**
13: $\quad\quad$ **end for**
14: $\quad\quad (cPay\text{–}unlock, pid, x_{Sj}, x_{Rj}) \longrightarrow F$
15: $\quad$ **end for**
16: $\quad wait\ for\ \Delta + 1\ rounds\ to\ return\ (x_{Sj}, x_{Rj})$
17: **end if**

F upon receiving $(cPay\text{–}unlock, pid, x_S, x_R)$ **from** I **or** R**:**

1: **if** x_S and x_R are valid preimages for the payment with pid **then**
2: $\quad$ increase I's capacity by $size + fee$
3: $\quad$ send $(cPay\text{–}unlocked, pid, x_S, x_R)$ to the former node of I
4: **end if**

Intermediary upon receiving $(cPay\text{–}unlocked, pid_0, x_S, x_R)$**:**

1: $Let\ u_j := (e_j, size_j, fee_j, fee_j^{revoke}, x_j, pid_j, pid) \in fw[T] \quad s.t.\ pid_j = pid_0$
2: $(cPay\text{–}unlock, pid, x_S - x_j, x_R - x_j) \longrightarrow F$
3: $fw[T] := fw[T] \backslash \{u_j\}$
4: **if** $\forall T, fw[T] = \emptyset$ **then**
5: $\quad wait\ for\ \Delta + 1\ rounds\ to\ return\ (x_S, x_R)$
6: **end if**

Sender upon receiving $(cPay\text{–}unlocked, pid_0, x_S, x_R)$**:**

1: $Let\ u_j := (e_j, size_j, fee_j, fee_j^{revoke}, x_j, pid_j, pid) \in fw[T] \quad s.t.\ pid_j = pid_0$
2: $out := out \backslash \{u_j\}$
3: **if** $out = \emptyset$ **then**
4: $\quad return\ (x_S - x_j, x_R - x_j)$
5: **end if**

Error handling

If some parties do not behave as expected, functions below makes sure honest parties always terminate and never lose money.

Sender in round T**:**

1: **for** $pid \in out$ **do**
2: $\quad (refund, pid) \longrightarrow F$
3: **end for**
4: $wait\ for\ \Delta + 1\ rounds\ to\ Terminate$

F upon receiving $(refund, pid)$ **from** I **or** S**:**

1: **if** the payment with pid expires **then**
2: $\quad$ increase I's capacity by $size + fee$
3: **end if**

Receiver in round $t_0 + 2 + 2|V|$**:**

1: $Terminate$

Intermediary in every round:

1: **if** $fw[now] \neq \emptyset$ **then**
2: $\quad$ **for** $(e_j, v_j, fee_j, pid_0, pid) \in fw[now]$ **do**
3: $\quad\quad (refund, pid_0) \longrightarrow F$
4: $\quad$ **end for**
5: $\quad$ **if** $fw = \emptyset$ **then**
6: $\quad\quad wait\ for\ 2(\Delta + 1)\ rounds\ to\ Terminate$
7: $\quad$ **end if**
8: **end if**