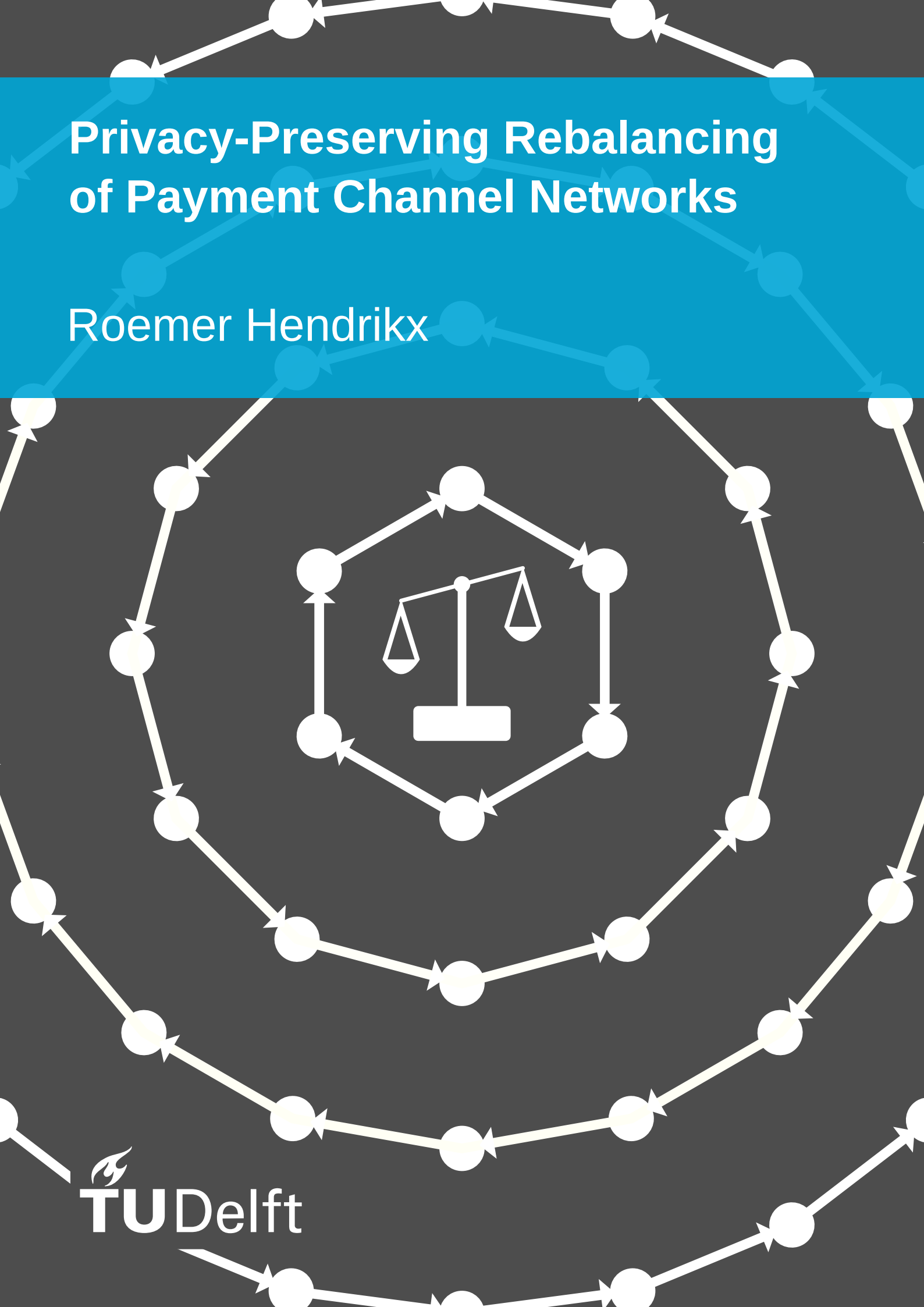
Privacy-Preserving Rebalancing of Payment Channel Networks

Roemer Hendrikx



TU Delft

Designing a Privacy-Preserving Rebalancing Algorithm for Payment Channel Networks

by

Roemer W.J. Hendrikx

to obtain the degree of Master of Science
at the Delft University of Technology,
to be publicly defended at 09:00 on Thursday the 27th of January 2022

FINAL VERSION

Student Number:	4346912
Project duration:	November 1, 2020 – January 27, 2022
Thesis committee:	Dr. Z. Erkin, TU Delft, supervisor
	Dr. S. Roos, TU Delft, co-supervisor
	Dr. S. Verwer, TU Delft

TU Delft

Abstract

In the past 8 years, Bitcoin has dominated the cryptocurrency markets and drawn attention from academia, developers and legislators alike. Bitcoin has been praised for its impact on decentralizing trust and currencies but also criticized for its volatility and energy-inefficient consensus mechanism. To improve its limitations, in 2016, payment channels and payment channel networks were introduced in the form of the Lightning Network. Payment channels allow for so-called off-chain transactions that, in case of dispute, can be published to an existing cryptocurrency blockchain, like Bitcoin, for arbitration. After its introduction, the concept of payment channels was quickly adopted by many cryptocurrency users. However, although payment channels remove the need for many on-chain transactions, some still remain. An on-chain transaction is required for the opening and closing of a channel. This happens during the initial setup between two users but is also required if one of the users runs out of balance on their side of the channel. The latter is a common occurrence as transactions are often unidirectional, say between a customer and a merchant. To limit the amount of closing and opening on-chain transactions required, a user can start or take part in a rebalancing. A rebalancing is a process with the aim of bringing a channel to a balance as desired by its owners. The state-of-the-art existing protocol to carry out a rebalancing is called Revive, which is a distributed protocol using leader election and a linear program to calculate the optimal rebalancing between its participants. Although effective, the protocol provides little privacy to its participants. We, therefore, designed a new, privacy-preserving peer-to-peer rebalancing protocol. Alongside it, we also introduce an accompanying participant discovery protocol that allows users in a network to find other users interested in running a distributed algorithm. We show that both protocols are secure and that our rebalancing protocol provides more privacy than Revive, at the cost of a suboptimal result and an increased message and time complexity. Finally, we compare our rebalancing protocol and Revive using a payment channel network simulator that simulates transactions taking place during the rebalancing. Using this simulation, we show that both protocols have a negative effect on the payment channel network as they lock the to-be-rebalanced channels while they are executing. We, therefore, conclude that an ideal rebalancing protocol should both be privacy-preserving and concurrent, and propose ideas to achieve this in future research.

Acknowledgements

As a young adult, I think that there are few things I have experienced up to this point that are as insightful and clarifying as writing a thesis. Throughout my many years at the TU Delft, starting with an aerospace bachelor and now (hopefully) finishing with a master in cybersecurity, writing a thesis is very different compared to any other report or document I have written over the years. All my life, I have liked telling stories as to me, it feels amazing that someone dedicates time and effort into reading and trying to understand something that I have created. Writing this thesis was like learning to tell a whole new type of story, for an audience that not only wishes to be entertained or intrigued but also requiring that every part of the story is thoroughly researched and factually true.

Learning to tell such a story would not have been possible without my supervisors Zeki and Stefanie, who provided me with the necessary feedback and motivation to write this thesis as a combination of the cybersecurity and distributed algorithms fields. I will never try again to make someone hearty pancakes without asking if they expected sweet ones instead. I also wish to thank Dr. Sicco Verwer for putting the time and effort into partaking in my defence beside my supervisors, and for already taking an interest in my topic all the way at my first master colloquium presentation.

I also wish to thank all fellow cybersecurity master students, PhD candidates and other specialists whom I have had many conversations with, some for research focused questions and sometimes just so I could rant about my problems. I want to thank my family, for their loving support during all those years I have been studying and who provided much-needed help in times of despair. Beyond that, I especially wish to thank my loving girlfriend Ásta, who kept me company throughout the writing of this thesis while writing a thesis of her own. I could not have done it without your motivation, support and confidence in my ability to write a thesis. Finally, I wish to thank my favourite artist Ramses Shaffy for creating his inspirational music that always cheered me up, gave me hope and allowed me to see the good side of life again during darker times.

Roemer Hendrikx
Reykjavík, December 2021

Contents

List of Figures

List of Tables

Chapter 1

Introduction

In 2008, a new era of distributed technology was launched with the introduction of blockchain by a developer going by the pseudonym of Satoshi Nakamoto [32]. Nakamoto created one of the first practical blockchains in order to introduce the cryptocurrency Bitcoin. Bitcoin has many features but one important feature in particular is its guarantee of the privacy and security for its users, while allowing everyone with a Bitcoin wallet to make payments to each other. These payments happen outside of conventional financial institutions like banks or governments. Since 2008, the adoption of Bitcoin has been steadily growing and inspired the creation of multiple other cryptocurrencies like Ethereum [3] and Dogecoin [34]. However, as can be seen in Figure 1.1, none of the alternative cryptocurrencies come close to the adoption of Bitcoin. Given its popularity, Bitcoin and other applications of blockchain technology have become a topic of interest for governments, developers and researchers alike over the last decade.

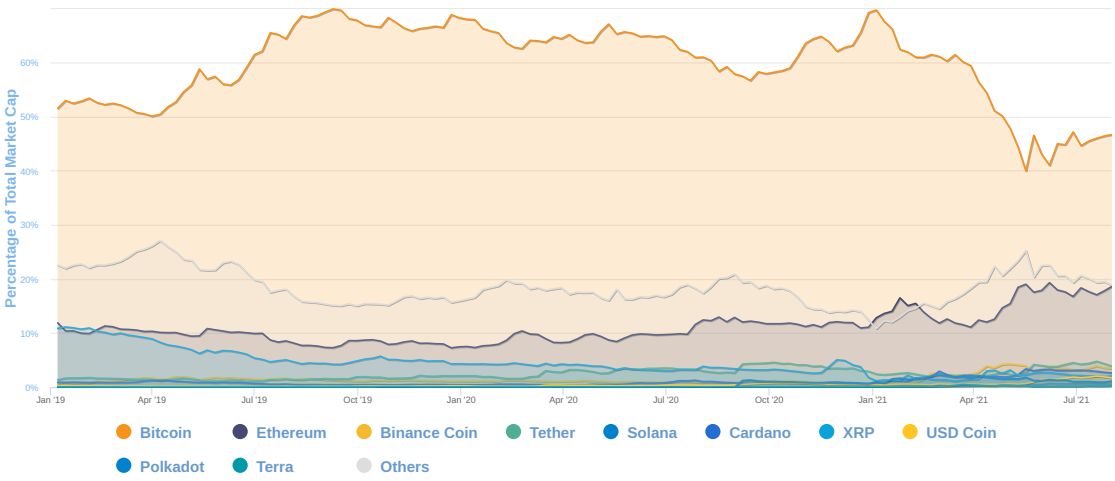


Figure 1.1: Visualization of the dominance of Bitcoin on the cryptoassets market over the period of January 2019 to August 2021 (CoinMarketCap)

Bitcoin uses a public blockchain as a distributed transaction ledger. Contrary to a normal ledger – where only the holder of the ledger can approve or deny transactions – a distributed ledger uses a peer-to-peer network of multiple entities who have to reach a consensus to approve a transaction before it is added to the ledger. If a consensus is not reached, the transaction is denied. In this way, no single entity holds all the power of approving or denying transactions. The presence of a distributed consensus mechanism is one of the necessary features that allows for the decentralization of a public blockchain. [39]

The consensus mechanism of Bitcoin uses a Proof-of-Work (PoW), a mechanism requiring anyone trying to add new transactions to the blockchain to expend a minimum amount of processing time. A PoW is effective in preventing an attacker from creating many fake entities in an attempt to influence the outcome of the consensus, as now the attacker has to pay a cost (i.e. processing time) for every entity it wants to use to influence the consensus [39]. However, on the scale of Bitcoin, this required extra processing power means that the energy consumption of Bitcoin is non-negligible. According to Jiang et al. [17], "...the annualized energy consumption of the Bitcoin industry in China will peak in 2024 at 296.59 TWh ... This exceeds the total energy consumption level of Italy and Saudi Arabia and ranks 12th among all countries in 2016."

Besides the problem of high energy consumption, Bitcoin also has a low transaction speed when comparing to other cryptocurrencies. As Bitcoin was the very first widely adopted cryptocurrency, it did not have the advantage of avoiding the problematic design choices of its predecessors. Bitcoin's primary design choices creating the low transaction speed problem consist of the PoW consensus mechanism and the chosen block size. The block size in Bitcoin determines the maximum amount that can be stored in a single block and thereby also the maximum number of transactions that can be stored in a single block. A Bitcoin block has a maximum size of only 1 MB. Together with the PoW[1], this limits Bitcoin transactions to 7 transactions per second [14] which is much lower than the 6500 transactions per second of Visa [52].

1.1 Payment Channels and Payment Channel Networks

To improve the slow transaction speed and high energy usage of Bitcoin, the concept of *off-chain transactions* was proposed. An off-chain transaction is a transaction that happens between two parties exchanging cryptocurrency without directly involving a blockchain, in contrast to *on-chain transactions* which always involve a blockchain. A well-known implementation of off-chain transactions is the Lightning Network, which was introduced in 2016 by Poon and Dryja [41]. The Lightning Network uses the concept of 'payment channels', which are blockchain transactions between two parties that have not yet been published on the blockchain. A transaction between the two parties is then simply an update of this blockchain transaction, signed by both parties. Opening a payment channel requires a *funding* on-chain transaction that removes funds from the participant's Bitcoin wallet and locks it in the channel. Closing a payment channel requires a *closing* on-chain transaction that unlocks the funds from the channel and deposits it in the participant's Bitcoin wallet. The collateral in the channel enables both parties to have a *balance* or *credit* in the channel, which represents the number of coins they can spend or move to the balance of the other party. See Figure 1.2 for an example of this concept.

In case of dispute, the transaction can be published on the blockchain similar to the closing transaction. The intuition behind this concept is that if both parties trust each other, there is no need for a third party (i.e. the blockchain) to verify their transactions and balances, but the option to do so is always available. This is akin to real-life contracts, of which most are never enforced by a court as both parties behave honestly.

Payment Channel Networks Extending the idea of payment channels, Poon and Dryja [41] also proposed the construction of a Payment Channel Network (PCN). In a PCN, a participant A pays another participant B via a chain of intermediary participants in a *multi-hop transaction*. To do this in a way that protects the intermediaries from losing any coins in the process, Poon and Dryja [41] also introduce Hash Time Locked Contracts (HTLCs). HTLCs guarantee that intermediaries and the sender can always recover their coins if a participant in a multi-hop transaction is malicious. HTLCs are a form of *conditional transactions* that can become void if proof of payment is not shown before a certain 'time' has passed. In a HTLC, the proof of payment consists of the preimage of a hash and the 'time' is measured in the amount of blocks that have been accepted by the blockchain since the start of the transaction.

Since the Lightning Network was first introduced, it has grown steadily with currently $\approx$ 12 000 nodes present on the network [24] and spawning numerous alternatives such as the Raiden

[1] The PoW causes block generation to take around 10 minutes

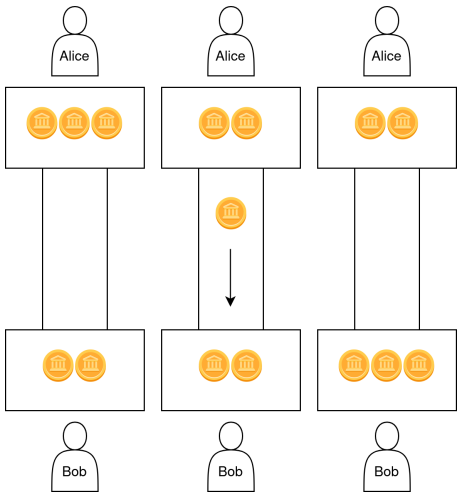


Figure 1.2: A diagram of a payment channel. From left to right: Alice and Bob fund the channel with 5 coins (3 for Alice, 2 for Bob) – Alice transfers 1 coin to Bob – Alice and Bob now have 2 and 3 coins as their balance, respectively. The second step of transferring the coins happens atomically in reality, i.e. the coin is either transferred or not and cannot be 'stuck in the channel'

network[2]. However, the Lightning Network does not solve all the problems with the use of Bitcoin[3]. Opening a channel, closing a channel and any dispute depends on on-chain transactions which suffer from the low transaction speed and high energy cost of Bitcoin.

Multiple methods have been proposed to limit the number of times a channel is opened or closed. Opening and closing of channels usually happen if one or both participants of the channel run out of balance due to many transactions going in one direction. Closing the channel and then reopening it with new funding allows for the channel to process transactions again. To remedy the times a channel has to be reopened and closed because of a lack of funding, Li, Miyazaki, and Zhou [22] propose a method to predict the right amount of funding a channel should be opened with. Another set of methods to limit the amount of opening and closing of a channel falls into the category of *rebalancing* and are the methods studied in this thesis.

Rebalancing Rebalancing is the process of moving the balance of a user in one channel to another channel. This improves the flow of transactions through a user's channels. In this thesis, a method called *on-demand rebalancing* is studied that takes advantage of the fact that in a payment channel network, a participant can pay themselves in a cycle via a chain of intermediaries[4]. This has the effect of moving the participant's balance in one channel to another one of its channels, up to a limit defined by both channels and the other channels in the cycle. In this way, the number of times a channel needs to be opened or closed if it runs out of funds can be reduced.

In a simple network, there might only be one such rebalancing cycle but with 12 000 nodes and many more edges, a network like the Lightning Network needs a protocol to find cycles and set up a cyclic transaction. One such protocol is called Revive [18], which is a distributed protocol in which a leader receives all the *rebalancing objectives* of the protocol's participants. A rebalancing objective takes the form of a value and a direction that specifies how the balance of the channel should be shifted. After receiving the rebalancing objectives, the leader uses a linear program to calculate the optimal set of cyclic transactions that are possible and distributes these to all the participants for validation and execution. As Revive produces an optimal set of cyclic transactions and is a relatively compact protocol, it is theoretically an effective and efficient protocol to carry out rebalancing with.

[2] https://raiden.network/

[3] The Lightning Network can also work with other cryptocurrencies but as Bitcoin is the largest cryptocurrency, it is expected that most transactions are backed by the Bitcoin blockchain

[4] See Figure 2.2 for an example of a PCN where cyclic payments are possible

1.2 Privacy considerations

A major drawback of Revive is its lack of *privacy*. In a PCN such as the Lightning Network, privacy can informally be quantified into the following components [54]:

- Sender/Recipient Privacy: An adversary should not be able to determine the sender/recipient of any payment between non-compromised parties.
- Value Privacy: An adversary should not be able to learn the exact value of any payment. Moreover, the adversary should learn as little information as possible about the range of value of any payment.
- Path Privacy: An adversary should not be able to learn the path(s) of any payment, other than the nodes it has already compromised.
- Channel Balance Privacy: An adversary should not be able to learn the exact balance of a payment channel at any given time, unless the channel connects to a node it has already compromised.

One can see that in Revive, if one of the participants is the adversary, the adversary would learn the sender, receiver, value and paths of all cyclic transactions as those are distributed to all participants. If the adversary happens to be the leader and knows the relation between the rebalancing objective and the channel balance of one or more participants, it is also possible for the adversary to figure out the current channel balance of those participants.

The authors of Revive briefly touch upon the privacy of their protocol, stating that they make no claims on the leaking of information in an adversarial setting. The authors suggest that if more privacy is desirable, a payment channel design should be used that allows the leader to only publish balance changes which limits the amount of information leaked.

We consider the lack of privacy of Revive to be a major issue for the adoption of rebalancing protocols by users of a PCN. We believe that for major adoption of such a protocol it would need to be effective in finding rebalancing cycles, be privacy-preserving, provide balance security and preferably be efficient in time and message complexity. Revive meets three of these four criteria as it is not privacy-preserving.

1.3 Research question

Based on the fact that Revive is not privacy-preserving, we deduced the following research question:

> How to construct a protocol that allows an arbitrary set of users in a payment channel network to securely rebalance their channels while achieving sender, receiver, value, channel balance and path privacy?

The objective of this thesis is to design such a protocol and compare it to Revive. Our contribution considers the problem both theoretically and practically. In our work, we analyse the privacy and security of our protocol in a malicious setting. We also analyse the performance of our protocol compared to Revive in multiple scenarios by using a dynamic simulation of the Lightning Network.

1.4 Contributions

We present in this thesis an on-demand rebalancing protocol that is a more privacy conscious alternative to Revive. Similar to Revive, our rebalancing protocol takes as an input a set of participants and their given demands, and finds a best-effort set of cyclic transactions. The main design goal of our protocol is to share the minimum of information required to find rebalancing cycles in a distributed setting.

By providing a privacy-friendly alternative to Revive, we hope to contribute to the more widespread adoption of rebalancing protocols, as they have the potential to positively impact

transaction routing in a payment channel network. If transaction routing is improved, payment channel networks become more practical for everyday use which in turn will make them a faster, cheaper and more environmentally friendly alternative to using Bitcoin directly.

In the construction of our rebalancing protocol we also introduce, for the first time to the best of our knowledge, a participant discovery protocol that allows nodes in a payment channel network to join the execution of a distributed algorithm. It outputs a list of anonymous participant identities that can be used as a starting point for protocols such as Revive or our rebalancing protocol.

To analyse the privacy of the participant discovery protocol, we also introduce and formally define *participation anonymity*, which is used to define if an adversary is capable of determining if a node and its edges are participating in a distributed algorithm.

1.5 Overview

In this thesis, background on related topics – such as the workings of a blockchain, payment channels and payment channel networks – is provided in Chapter 2. Related work, in particular about payment channel networks, transaction routing and rebalancing, is discussed in Chapter 3. The chapter also discusses Revive in more detail, given its direct relation to this thesis.

The discussion about the design of our rebalancing protocol and the participant discovery algorithm can be found in Chapter 4. Chapter 4 also covers the formal definition of a payment channel network and the breakdown of the research question into design requirements. At the end of this chapter, the privacy of both protocols is analysed. The actual implementation of the rebalancing protocol, the discovery protocol and the evaluation using the simulation of the Lightning Network are discussed in Chapter 5.

Our thesis concludes with the conclusion and future work in Chapter 6.

Chapter 2

Background

In this chapter, we introduce the foundations of our work. These consists primarily of blockchain, payment channels and payment channel networks.

2.1 Blockchain

An ideal blockchain is a distributed data structure that allows multiple users to read from and write to it while guaranteeing that many users consent to the writing [46].

A blockchain consists of multiple *blocks*. Each block holds the information that is intended to be stored in the blockchain while also holding a hash of the previous block in the chain. In this way, each new block is coupled to all the previous blocks before it, forming a chain. This chain continues to the first block in the chain, which is often called the Genesis block. An efficient way to represent this data for each block is by using a Merkle tree to create a single hash representing the block [23]. See Figure 2.1 for a visualization of a blockchain.

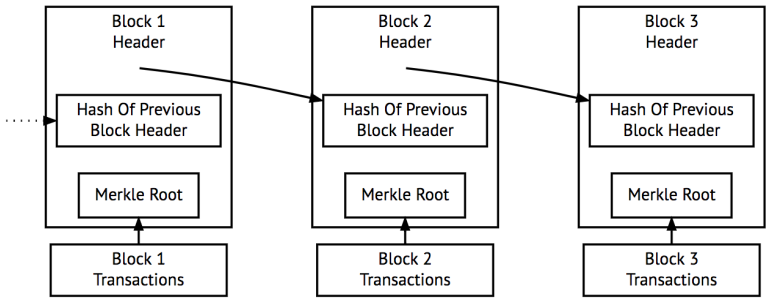


Figure 2.1: A simplified view of a blockchain [2]

As there exists a chain to the first block, it is possible for anyone who obtains a block to validate that it is indeed a part of the blockchain [23, 46]. This is useful when a blockchain is used as a distributed ledger for cryptocurrencies, as it allows anyone (say Alice) to validate a cryptocurrency transaction by checking if all the blocks up until the most recent one sum up to the wallet amount that the owner (say Bob) claims to have. If Bob speaks the truth, the transaction can commence.

2.1.1 Consensus

On its own, this validation property of a blockchain is only useful if the blocks in the blockchain itself can be trusted. If anyone would be able to simply add a block to the chain, Bob could add a block stating that he received a transaction from Eve and that would be considered the truth according to the validation strategy as discussed above. To prevent any one user from adding a block to the chain, one could consider allowing only a subset of trusted users (a.k.a. 'owners' or 'validators') to add blocks to the chain. This setup is also known as a permissioned blockchain

since a user needs to have permission from the trusted users to become trusted as well [46]. In a permissioned blockchain, there might also be restrictions on creating transactions or reading the blocks in the chain.

In contrast to permissioned blockchains, more well-known are the public blockchains that are in use for cryptocurrencies such as Bitcoin or Ethereum. Public blockchains cannot rely on a set of trusted users and therefore require a consensus mechanism such as Proof-of-Work or Proof-of-Stake, both of which theoretically allow anyone to act as a validator of the blockchain and add blocks [46]. The goal of the consensus mechanisms in a public blockchain is to make it very difficult or expensive for an adversary to hold a majority of the validators of the blockchain. If this happens, the adversary can arbitrarily add any block to the chain they like given that all its validators approve the addition. This is also called a 51 % attack [46].

Proof-of-Work

A Proof-of-Work consensus mechanism has as its most distinguishing feature a mathematical challenge that is hard to solve but easy to verify. For example, in Bitcoin, the Proof-of-Work mechanism is the challenge of finding a preimage of a SHA-256 hash that has a specified number of zero bits. The preimage must also be a combination of a nonce and the hash of the block that is to be added [39]. Only the nonce can be varied to produce the required preimage of the hash. As this is a hard problem (and is kept hard by slowly increasing the number of zero bits required), it takes about 10 minutes in Bitcoin before a new block is added to the blockchain. In the unlikely event that two validators (or 'miners', as they are known in Bitcoin) solve the challenge simultaneously, two branches of the blockchain will temporarily exist, one with block A and one with block B, both with equal length. The next block to be added to the blockchain will then most likely break the tie by either increasing the chain with block A or block B, causing the validators to discard the shortest branch. This is the reason that users of a blockchain with such a consensus mechanism are requested to only consider their transaction valid after the block which holds their transaction has been inherited from by one or more other blocks. This reduces the likelihood that their block ends up in the shortest branch and gets pruned, which would make their transaction invalid [39].

Proof-of-Stake

Although the Proof-of-Work mechanism is battle-tested in many blockchains, it has the drawback that if a user wishes to become a validator, a huge investment in hardware is required to be able to solve the challenge. Some other important drawbacks are that the powerful hardware required to do so consumes a lot of energy [17] and that solving the challenge takes a lot of time, slowing down the speed with which transactions are processed. The other consensus mechanism, Proof-of-Stake, differs from Proof-of-Work in that it does not depend on solving a challenge. Instead, Proof-of-Stake allows any user to become a validator simply by staking a predefined number of coins. The validator can then lose (part of) this stake for a variety of reasons, such as colluding with other validators or by going offline. As Proof-of-Stake does not depend on solving a difficult problem, blockchains that implement it use much less energy than ones using a Proof-of-Work and are predicted to be faster in executing transactions [46].

A newer version of Ethereum, Ethereum 2.0, plans to use the Proof-of-Stake mechanism. Its authors argue that a 51 % attack is still possible with a Proof-of-Stake – although impractical – as an adversary would have to control 51 % of the staked ETH to do so. The authors argue that this would represent such a large amount of the total ETH available that this would most likely drop the value of ETH, which would be a negative incentive for an adversary willing to control the Ethereum blockchain. [53]

2.1.2 Smart contracts

The final topic to be discussed here are *smart contracts*, as they are used in the construction of payment channels. Smart contracts are a way for the users of a blockchain to execute arbitrary code on the blockchain. A smart contract is deployed to a blockchain with a transaction that also

includes the code that defines the contract. Once deployed, the contract itself has a wallet and can be called by users to execute its functions which may or may not update its wallet. Smart contracts have access to almost all the information on the blockchain and can act similarly to a user by executing transactions or even creating other contracts [23]. This is useful as it allows smart contracts to distinguish between its owner (the user who deployed it) and other users interacting with it. A downside of the flexibility of smart contracts is that execution can take an arbitrary time and the execution happens on the hardware of the validator. To reimburse the validator for the added execution time, all blockchains that support smart contracts require a cost that the user using a smart contract has to pay to invoke it. The cost is often calculated differently per operation of the smart contract, e.g. adding may be cheaper than multiplying.

Smart contracts have a multitude of uses but a relevant example of a smart contract is the one used in the construction of payment channels. This contract defines a multisignature account, i.e. a wallet that requires more than one user to do transactions with. The deployed contract validates that every transaction request it receives has been signed by the public keys it was deployed with. If the transaction validates, the contract creates the requested transaction and publishes it to the blockchain.

2.2 Payment channels

A payment channel is a protocol that allows its users to carry out transactions of coins outside of the blockchain, so-called *off-chain transactions*. These off-chain transactions are still bound to the blockchain but are not yet published on the blockchain. Intuitively, payment channels work similarly to a physical contract for which, before signing, both parties need to agree to the terms of the contract. Nobody needs to enforce the contract as long as both parties act out the contract truthfully. If one of the parties acts dishonestly, only then does a party need to go to court and enforce the contract. Similarly, with a payment channel, both parties sign a contract representing the initial balance of a payment channel. For every transaction that is executed afterwards, both parties sign a new contract representing the new initial balance. This contract supersedes the older contract with the old balance. Nothing happens with the contract unless either: 1) the parties wish to end the channel, in which case it is published to the blockchain to be executed or 2) there is a dispute about the current balance of the channel, in which case the latest contract signed by both parties is published on the blockchain. A diagram of a payment channel is provided in Figure 1.2. [41]

The largest benefit of the use of payment channels is a reduction of time and cost for executing transactions. If two users establish a payment channel with each other, there is only the cost and blockchain processing overhead for the initial funding transaction and the closing transaction, technically allowing an infinite number of transactions between the two users. [41]

In this thesis, the payment channel design that is most referred to is the Lightning Network or Lightning for short, initially created by Poon and Dryja [41]. The design of Lightning's channels is also the basis for the background given in this chapter concerning payment channel networks. Similarly to what we explained earlier, Lightning's payment channels allow users to carry out off-chain transactions after establishing the initial balance of a channel. It is important to note that in Lightning, if a user does not abide by the contract rules (i.e. claims a different balance than the real balance of the channel), the design is such that all the funds of the channel are awarded to the honest user as a penalty. This property serves as an incentive for a user to act honestly and according to the protocol, and is also called: "An honest user will never lose coins". However, the mechanism that allows a penalty to be enforced requires a certain dispute time (around 1000 blocks) and during this dispute time (also called a time-lock), the funds of the honest party are locked up. This allows for denial-of-service attacks where a user acts dishonestly on purpose to lock the funds of the honest party, although this is expensive for the dishonest party.

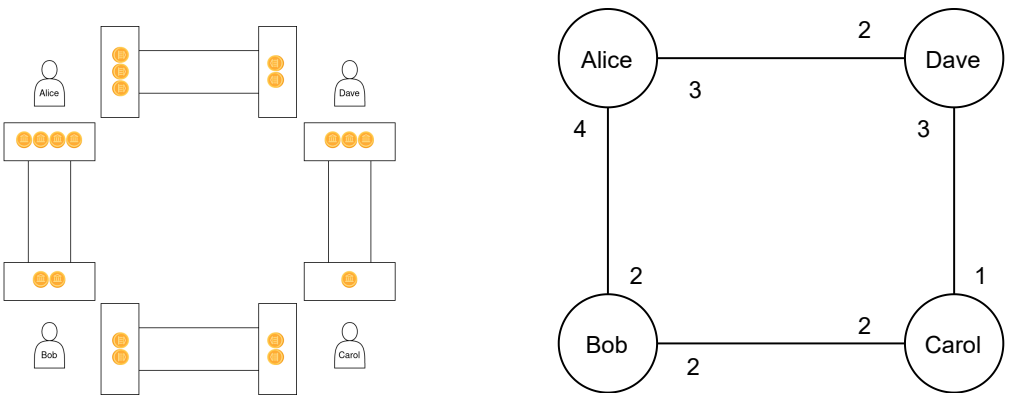
Alternative designs of payment channels and payment channel networks can be found in Chapter 3.

2.3 Payment channel networks

Payment channel networks (PCNs) are networks created out of multiple individual payment channels. A simple example of such a network is Subfigure 2.2a, where Alice, Bob, Carol and Dave all have two channels that together form a network. For simplicity, in this thesis, only PCNs that look similar to Subfigure 2.2b are used where the balance of each user in a channel is denoted by a number adjacent to the channel edge.

2.3.1 HTLC

The promise of a PCN is that it allows Alice to pay Carol without the need for Alice to set up a channel to Carol, saving blockchain transaction costs. Such a multi-hop payment or payment route works as follows: If Alice wishes to pay Carol for a service, she would first need to pay Bob x coins after which Bob transfers x coins to Carol. If all users act honestly, this process works. However, if Bob is evil, Bob could take the payment from Alice and never pay Carol. If Alice is evil, she could never pay Bob and claim to Carol that she has paid Bob, shifting the blame to Bob. To prevent evil senders and intermediaries from acting dishonestly, in the Lightning Network multi-hop payments are set up using Hash Timelocked Contracts (HTLCs) [41].



(a) Actual (b) Diagram

Figure 2.2: Two different representations of the same PCN

The reason for the existence of HTLCs is that one wishes for the sender and receiver of a multi-hop transaction to have a guarantee that their coins will not be stolen by intermediaries without having to trust all of them [41]. It is unreasonable to expect that any user has vetted all intermediaries of a payment route given that routes change from one transaction to the next and their length varies. HTLCs work by having the receiver of a transaction generate a preimage s and a cryptographic hash $H(s)$. A common hashing algorithm used for this purpose is SHA-256[1]. The receiver then sends *only* the hash to the sender of the transaction. After receiving the hash, the sender (say Alice) then pays the first intermediary I_1 the required amount, with the added condition that the amount can only be claimed if I_1 produces the preimage s of the hash within t blocks added to the blockchain. I_1 then pays the next intermediary I_2 under almost the same condition but must change the t to something smaller. If the time-lock is larger than t, I_2 could produce s after t blocks and claim the funds from I_1. However, as t blocks have gone by, I_1 cannot claim the funds from Alice which would result in I_1 losing coins in this transaction. To counteract this risk, every intermediary after the sender should have a time-lock with a time smaller than the previous step in the payment route [41].

This process of conditional payments repeats until the last intermediary I_n has paid the receiver (say Carol). Carol then reveals the preimage s to I_n, finalizing the conditional payment. I_n can then show s to I_{n-1} to claim its funds, etc. until I_1 claims its funds from Alice which completes the transaction. In this way, HTLCs allow a form of atomic payments in a PCN.

[1] See https://helix.stormhub.org/papers/SHA-256.pdf for more information about SHA-256

2.3.2 Fees

Although HTLCs allow for multi-hop transactions, they do not provide any incentive for intermediaries to participate in such a transaction. The existence of an incentive for intermediaries is important as intermediaries have to lock part of their balance in a multi-hop transaction as part of the HTLC, and are required to continuously watch the blockchain to see if their direct neighbours uphold the channel contract. To provide this incentive, the Lightning Network allows intermediaries to ask for a *base fee* and a *fee rate*, where the latter is also known as a *liquidity fee*. The base fee stays the same regardless of the transaction amount and the fee rate changes based on the transaction amount, recognizing that large transactions lock more coins during multi-hop transactions. In a multi-hop transaction, fees are paid by the sender and are added to the total transaction amount of which every intermediary deducts his/her fee along the route.

2.3.3 Routing

Another component of multi-hop payments is routing. Going back to Figure 2.2, if Alice wishes to pay 2 coins to Carol, she can either go via the route Alice - Bob - Carol or Alice - Dave - Carol. In the Lightning Network, the sender of the transaction decides the full route from start to finish (BOLT #4, [25]). This is necessary as the sender also has to pay the transaction fee in advance, which requires knowing which intermediaries a transaction goes through as each intermediary asks for different fees. This is also called source routing. The following are important parameters for Alice to consider when deciding on a route:

- *Fees*: A route A can have fewer fees in total than a route B. From an economical perspective, it makes sense for Alice to pick a route with the least fees.
- *Timelock duration*: As each intermediary can specify its own minimum time-lock duration, a route A can have a larger or smaller total time-lock duration than a route B. It makes sense to pick a route with the smallest time-lock such that if something goes wrong, recovery takes little time.
- *Capacity*: A route must be able to support the payment. Taking Figure 2.2 as an example, if Alice would have wished to pay 3 coins instead of 2, only the route of Alice - Dave - Carol would be suitable as Bob does not have enough balance to pay Carol. When considering which route to pick, Alice must also take into consideration the available capacities of all channels between her and the receiver.

In the Lightning Network, a gossiping protocol is used to distribute information about active nodes, available channels, fees, time-lock durations and maximum channel capacities (BOLT #7 [25]).

The real-time capacity of a channel is not distributed as it is related to the funding of the channel and its current balance, which is considered private information belonging to the channel owners. If the real-time capacity of every channel would be publicly known and is updated because of a transaction, an adversary could subtract the new capacity from the old capacity to figure out the value of the transaction that has taken place on the channel. To protect against such attacks, real-time capacities are not published on the Lightning Network's gossip channels[2].

Using all the gossip information, the specific choice for which routing algorithm to be used is left to client implementations compatible with the Lightning Network. This is because the Lightning RFC [25] only specifies that a routing algorithm must use source routing and that the route must be distributed using a protocol similar to onion routing[3]. The algorithm used by *lnd* [26] – the reference Lighting Network client – is a modified version of Dijkstra's algorithm.

[2]This still allows an attack known as balance probing where an adversary sends payment requests in decrementing sizes until the victim accepts the request. The attacker then roughly knows the current capacity of a channel. This attack is expensive to perform as every payment request locks up coins of the adversary and the adversary has to perform the attack faster than the frequency with which the channel updates its capacity. If a lot of transactions take place over the channel, the capacity might change faster than the attacker can deduce it.

[3]Known from the TOR network and its accompanying browser

Not knowing the real-time capacity of a channel during the sender's route planning also introduces the possibility of multi-hop transactions to fail. This failure mode is different than the earlier discussed failure modes which required dishonest users. Not knowing the real-time capacity of all the channels in all possible routes means that any route can fail as one channel with insufficient capacity can cause failure in the whole route. To improve the success rate of transactions, *lnd* keeps track of all past transactions to calculate the probability that a route may fail or succeed. Alternative routing algorithms try to find different ways to deal with the limited knowledge on channel capacities, and are discussed in Chapter 3.

Chapter 3

Related Work

Payment channel networks (PCN) are a relatively new development in the field of blockchain technology. A well-known example of a PCN was introduced by Poon and Dryja [41] in 2016, named the Lightning Network. As of February 2021, the Lightning Network is the largest example of a deployed PCN. The Lightning Network has around 12 000 active nodes [24] compared to around 36 for the Raiden network [42]. As one of the first PCNs, Lightning functioned as a proof-of-concept that PCNs can become a reality. Since Lightning, academic efforts have increased to improve the security, privacy, concurrency, availability and routing of PCNs. This chapter introduces the literature on a selection of topics relevant to this thesis.

3.1 Security and privacy

Proofs and definitions Before the introduction of Lightning, Moreno-Sanchez et al. [38] introduced PrivPay, which is a payment protocol for credit networks. Although not applicable to decentralized PCNs because of the centralized nature of PrivPay, the work contains a formal definition of *value privacy* and *receiver privacy* in credit networks using security games. According to the authors, informally a credit network has *value privacy* if the adversary cannot obtain the value of a transaction between two non-compromised users. The authors informally define *receiver privacy* as when the adversary cannot determine the receiver of a transaction from an uncompromised sender.

In a later work by Malavolta et al. [30], the authors present a formal security and privacy treatment of *balance security*, *value privacy* and *sender/receiver anonymity* using the Universal Composability (UC) framework [5]. The authors state that balance security "...guarantees that any honest intermediate user taking part in a `pay` operation ... does not lose coins even when all other users involved in the `pay` operation are corrupted." Although the authors define *value privacy* the same as [38], they phrase *sender/receiver anonymity* differently than *receiver privacy*. The authors informally define sender/receiver anonymity as: Given two simultaneous successful `pay` operations with the same sender, receiver and intermediaries, and at least one honest intermediate user, corrupted intermediate users should not be able to determine the sender or the receiver for a specific `pay` operation with a probability better than 1/2. The work of the authors continues by providing two designs of PCNs called Fulgor and Rayo. Both designs utilize Multi-Hop Hash Time-locked Contracts (Multi-hop HTLCs) but differ in that Rayo orders transactions in a global state to provide concurrency at the cost of anonymity while Fulgor preserves anonymity at the cost of dropping all concurrent transactions.

A multi-hop HTLC is different from a normal HTLC as used in multi-hop transactions. In normal multi-hop transactions, one hash and one preimage is used along the entire route which, according to the authors, breaks the sender/reciever anonymity. In a Multi-Hop HTLC transaction, for the receiver and all intermediate nodes i, the sender creates a random string x_i and $y_i = H(\oplus_{j=i}^{n} x_j)$, utilizing the previously generated x_i. The sender then sends (y_n, x_n) to the receiver and (y_{i+1}, y_i, x_i) to each node i over an anonymous channel. Once this is done, each node i sets up a standard conditional HTLC payment with the next node $i+1$, using y_{i+1} as the

hash condition. Once the conditional payments are set-up all the way to the receiver, the receiver reveals x_n which is the preimage of y_n. The intermediary $n-1$ can then use x_n to construct the preimage of $y_{n-1} = H(x_{n-1} \oplus x_n)$. This process continues until the first intermediary has claimed its payment from the sender.

Kiayias and Litos [19] present another security treatment for the functionality of the Lightning Network in the UC framework, proving that honest users cannot lose funds in the Lightning Network. Their treatment differs from the proof attempts by [9, 30, 37] as the authors of those works assume a ledger with instant finality, which Kiayias and Litos [19] proof is unrealisable in practice even with strong network assumptions. The authors note that proving the balance security property of the Lightning Network "...acts as a guarantee to the almost 900 bitcoins currently in circulation in the layer-2 protocol".

Balance privacy attack In the work of Herrera-Joancomartí et al. [15], the authors explain a privacy attack on the Lightning Network utilizing multiple transactions that do not finalize in order to obtain the exact balance of a channel. The authors vary the transaction amount in such a way that the balance of a channel can be probed using a limited amount of transactions. According to tests done by the authors, all the channels of 1432 nodes can be attacked in a minute with an accuracy of 10 USD, or slower if higher accuracies are desired. The authors note that for a worst-case scenario (where the initial balance guess is far off from the real balance) with 624 nodes, the costs of the attack are roughly 50 USD. As a mitigation to this attack, the authors suggest multiple approaches: The first option is that the nodes deny payment requests based on a (possibly random) heuristic. The second option is to extend Lightning to adopt a differential privacy technique as used in smart power meters, s.t. only a usable derivative of the real channel balance is revealed.

Bolt: Blind Off-chain Lightweight Transactions In order to improve the privacy of payment channels, Green and Miers [13] introduced Bolt in 2017. The authors claim that Bolt improves upon the privacy of Lightning (and other payment channel designs) by using cryptographic primitives to prevent a malicious user from learning the channel balance of its channel partner. In Lightning, the balance of a channel is known to both partners. To create a balance-anonymous unidirectional payment channel between a customer and a merchant, Bolt uses a modified version of *Compact E-cash* by Camenisch, Hohenberger, and Lysyanskaya [4]. Compact E-cash provides a cryptographic construction for allowing a customer to withdraw electronic coins at a bank B, and later spend these coins without the bank B being able to trace who withdrew these coins (i.e. spending unlinkability). The *compact e-cash* scheme by itself works for spending coins but has the limitation that if the customer wishes to close the channel without spending all the coins, it would be possible for the merchant to identify the source of all previously spent coins. After modifying the scheme to allow early closure, the authors extend it to support bidirectional channels by allowing the customer to exchange its old wallet for a new wallet with the merchant, which can hold either less or more coins than the old wallet. The authors note, however, that a dispute about the channel balance requires revealing the final balance of the channel. Finally, the authors emphasize that the design does not yet allow for creating a network of payment channels in order to create a PCN.

3.2 Transaction routing

Transaction routing in PCNs is an active subfield of PCN design. In the period between 2016 and 2021, more than 13 works were published that only focused on transaction routing. A common theme of the transaction routing research is to improve the *probability of transaction success*, i.e. how many transactions complete on the first try. In a PCN like Lightning, an equally balanced channel (i.e. both users have the same amount of coins on each side, say 5-5) can become fully unbalanced (i.e. 0-10 in our example) when transactions are unidirectional. This prevents any further transactions from taking place in that direction unless the channel is refunded (which often involves closing and reopening it) or rebalanced. Research into transaction routing tries to deal with a network where previously available channels become unavailable because of this. Other

researchers prefer to focus on transaction privacy instead (i.e. keeping one or more of the following private: sender, receiver, amount, channel balance, path/direction) or concurrency of transactions (i.e. multiple transactions going simultaneously over the same channel).

SilentWhispers SilentWhispers is a routing algorithm proposed by Malavolta et al. [31] in 2017 to address privacy concerns with PCNs that allow everybody to "...know who paid what to whom." The authors note that this conflicts with users who prefer to hide their operations or do not want to reveal any personal information such as the height of medical bills or salary.

SilentWhispers is a *landmark* routing algorithm. Such an algorithm uses landmarks in a network, which are nodes that are known to every other node in the network. Using landmarks requires a directed weighted network, which the authors construct by creating two directed edges for each payment channel. Each directed edge between users u_1 and u_2 has a weight that "...indicates the *unconsumed* credit that a user u_2 has extended to u_1". Each landmark node carries out a Breadth-First-Search (BFS) of only forward edges which allow it to find the shortest paths from the landmark to each node, creating a spanning tree. It then does another BFS considering only backward edges which allows it to obtain all the shortest paths from each node to the landmark, creating another spanning tree. During this process, each visited node learns their parent in the path to and from each landmark. The authors note that using landmarks is efficient as when the moment comes that a sender needs to find the route to a receiver, the sender and receiver pick a common landmark L after which a path can be stitched together from the sender to L and from L to the receiver, given that both the sender and receiver know the shortest path to and from L. As every route of SilentWhispers contains a landmark, SilentWhispers is also said to be a *landmark-centered* algorithm [44]. The authors say that a drawback of using landmarks is the fact that the routing information needs to be updated regularly as the credit information of the PCN changes.

To calculate the unconsumed credit over each path without all the visited nodes having to share it with the landmark (which is a privacy problem), the authors propose to use Secure Multi-Party Computation (SMPC) such that the landmark node can run multiple `min` functions combining all the credit information of a path while only learning the outcome and not the inputs. As this requires shares to be sent to the landmarks by the nodes on the path, the authors introduce chained digital signatures to prevent nodes outside the path from influencing the calculation of the `min` by the landmark. The authors construct this chain of digital signatures using long term keys and fresh keys, where the fresh keys are used for the signing of the credit shares that are sent to the landmark node. This prevents the landmark node from learning which node sent what share. The authors also introduce a dispute resolution mechanism in case the two parties involved in a channel disagree about what the value of the unconsumed credit may be. Finally, the authors prove the security of their scheme in the Universal Composability framework [5] under the assumption of the existence of a secure secret sharing scheme, an existentially unforgeable digital signature scheme and only passive corruption of a proper subset of landmarks.

SpeedyMurmurs SpeedyMurmurs is a routing algorithm introduced in 2017 by Roos et al. [44]. The authors introduce SpeedyMurmurs to solve existing problems with routing on the topics of privacy, efficiency and scalability. For example, the authors claim that Flare leaks sensitive information about users, as every user along a transaction path has to share their current balance with the sender. The authors also studied SilentWhispers and found it promising but lacking efficiency. The authors give some examples as to why; SilentWhispers updates its spanning trees only periodically, reacting late to changes in the network. Another example is that SilentWhispers routes always include the landmark and may therefore be longer than the true shortest path in the PCN.

SpeedyMurmurs extends VOUTE, which is "...a privacy- preserving embedding-based routing algorithm for message delivery in route-restricted P2P networks" [44]. VOUTE cannot handle directed and weighted links, which is why the authors extend it to work in a PCN. The authors note that PCNs are similar to the type of network VOUTE was designed for as both networks are *route-restricted* (i.e. channels are not created to improve routing quality). As it is based on VOUTE, SpeedyMurmurs uses a greedy embedding-based routing technique called Prefix Embedding. The

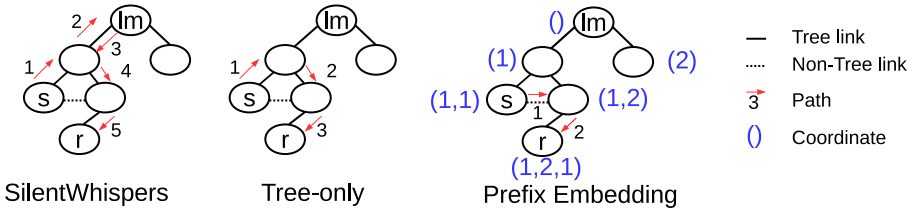


Figure 3.1: Example of differences between routing with SilentWhispers, tree-only routing and Prefix Embedding where lm is the landmark, s the sender and r the receiver (Roos et al. [44])

authors explain that embedding-based routing relies "...on assigning coordinates to nodes in a network and having nodes forward packets based on the distances between coordinates known to that node and a destination coordinate." The authors note that *greedy* embedding-based routing is similar to tree-only landmark routing as it assigns coordinates based on the position from a landmark node in a tree. However, this kind of routing is not limited to the existing tree but can also take shortcuts between leaves of the tree which allows for more efficient routing than compared to tree-only landmark routing, as can be seen in Figure 3.1.

SpeedyMurmurs contains three distributed sub-algorithms: `setRoutes`, `setCred` and `routePay`. `setRoutes` is a modified version of VOUTE's coordinate assignment to allow for weighted and direct links. `setCred` is an algorithm that reacts to nodes that wish to change the value of their link and decide if a coordinate change should occur. The authors decided to allow only coordinate changes if a new link is added with non-zero weight or if a non-zero link is removed from the network. Finally, `routePay` allows a sender and a receiver to find a route for a multi-path transaction. It takes into account available funds on each possible route and splits payments over multiple routes if only a combination of routes has the necessary capacity for routing the transaction amount.

According to simulations done by the authors, SpeedyMurmurs outperforms SilentWhispers and a reference Ford-Fulkerson implementation in terms of efficiency and probability of transaction success. In later work by Eckey et al. [10], SpeedyMurmurs is used as a sub-algorithm for an algorithm that introduces splitting payments by intermediaries along the multi-hop transaction path, further increasing the probability of transaction success.

3.3 Distributed cycle finding

Rocha-Thatte Distributed Cycle Detection In 2015, Rocha and Thatte [43] introduced a distributed algorithm for detecting cycles in directed graphs. The algorithm is synchronous and intended for use in computing systems. Although cycle detection was already possible using a depth-first search algorithm, the authors reference another work that shows that it is hard to parallelise and therefore a new *distributed* cycle detection algorithm might be necessary.

The algorithm works as follows, under the assumption that each node has a unique numerical identifier: In the first iteration i of the algorithm, a node $u \in V$ sends a message containing only its own identifier ($\mathrm{ID}(u)$), wrapped in a sequence s of length 1, on its outgoing edges E_u^o. In all following iterations, u adds its own identifier to the end of the received sequences and forwards them all on E_u^o. If u does not receive any sequences during the iteration, it deactivates. If all nodes are deactivated, the algorithm terminates.

A received sequence s is not forwarded if the first ID in s is equal to $\mathrm{ID}(u)$. In that case, u has detected a cycle. To prevent multiple nodes from reporting the same cycle, a cycle is only reported if $\mathrm{ID}(u) = \min(s)$. The other reason to discard s is if $\mathrm{ID}(u) \in s$ but not at the beginning of s. If this happens, another node must have already detected the specific cycle in an earlier iteration. See Figure 3.2 for an example of the execution of the algorithm.

The authors prove the algorithm's correctness and theoretically analyse the number of messages and number of iterations required to run the algorithm in various graphs. If G is a complete directed graph with cycles, the authors proof that the total number of messages $Y_t = n\, n^{\underline{t+1}}$, where t is the current iteration, n the number of nodes and $n^{\underline{t}}$ the falling factorial $n(n-1)\cdots(n-t+1)$.

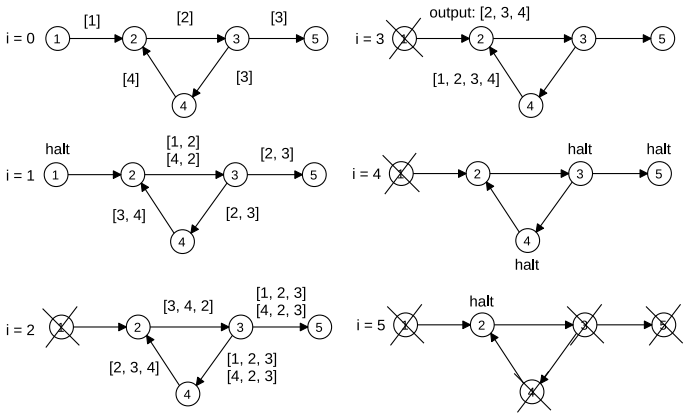


Figure 3.2: Example of an execution of the Rocha-Thatte distributed cycle detection algorithm (Rocha and Thatte [43])

The authors consider a complete graph the worst-case scenario. If G is a random directed graph in which each possible directed edge has probability p to exist, then $\mathbb{E}[Y_t] = n\, n^{\underline{t+1}} \cdot p^{t+1}$. The authors also prove that the number of iterations required is equal to the longest path in G. Finally, the authors provide a way to use their cycle detection algorithm to detect strongly connected components.

3.4 Rebalancing

Rebalancing is the process of taking an *imbalanced* payment channel of a node and restoring its balance without carrying out on-chain transactions (e.g. closing and opening a channel with new funds). What it means for a channel to be *imbalanced* or *fund skewed* is a topic of discussion among researchers and is explored in this section among solutions to solve this imbalance through rebalancing. For ease of analysis, one can distinguish two types of rebalancing algorithms: *on-the-fly* and *on-demand*. On-the-fly rebalancing algorithms are algorithms that try to influence the transaction routing in such a way that the balance of channels improves. They differ from on-demand rebalancing algorithms because those are specifically *invoked* if a node wishes to balance a certain subset of its channels.

FSTR An on-the-fly protocol for rebalancing is *FSTR* or Fund Skewness Aware Transaction Routing by Lin, Zhang, and Wu [27]. In their work, the authors define the *fund skewness* in a certain direction as Equation 3.1a.

$$\varphi_{uv} = \frac{f(u,v) - f_a(u,v)}{f_s(u,v)} \tag{3.1a}$$

$$\varphi_{vu} = -\varphi_{uv} \tag{3.1b}$$

In Equation 3.1a, u and v are two users sharing a channel, $f(u,v)$ represents the coins that can be transferred from u to v over the channel, and $f_a(u,v)$ and $f_s(u,v)$ are the average and sum of $\{f(u,v), f(v,u)\}$ respectively. The authors then present three methods to calculate a *path skewness* metric that is used to determine the skewness of a multi-hop transaction. The routing method used in FSTR uses a modified BFS and stores found routes in a routing table. Routes are selected based on the path skewness metric. Simulating the algorithm and comparing it to a reference Ford-Fulkerson max flow and SpeedyMurmurs [44] implementation, the authors decide to choose a non-linear path skewness metric for their algorithm. Using this skewness metric, the authors experimentally show that the probability of transaction success using their algorithm for transaction routing is larger than compared to using Ford-Fulkerson or SpeedyMurmurs.

A drawback of FSTR is that its route discovery uses a REQ message that contains, among other things, the full path of the transaction and all the skewness values for the channels the REQ

message has passed. This REQ message is sent to all the nodes who might potentially have a path to the receiver. Compared to the current routing algorithm of the reference Lightning client `lnd`, which uses source-routing to prevent intermediaries learning the path of the transaction, FSTR fully reveals the path of the transaction to all potential intermediaries. In addition, an adversary can use the skewness values in the REQ to derive the channel balance of all the channels the REQ passed through using Equation 3.1a.

CLoTH Another work that investigates on-the-fly and on-demand rebalancing is presented by Conoscenti, Vetrò, and Martin [6] and is a continuation of [7] where a simulator called CLoTH is presented based on the Lightning Network client `lnd` (the reference Lightning client). In [6], the authors use CLoTH to investigate the effects of different payment rates, payment amounts, rebalancing strategies and the removal of hub nodes. The authors find that higher payment rates do not majorly influence the probability of transaction success although higher payment amounts have a visible negative effect on the probability. The authors note that higher payment amounts cause transactions to fail as there is no route available with sufficient capacity. For the rebalancing investigation, the authors considered both on-the-fly as well as on-demand rebalancing strategies. For the on-demand strategy, the authors triggered nodes to rebalance as soon as one of its channel balances is below 20 % of the total channel capacity. If a node is triggered to rebalance, it tries to transfers its coins from a highly balanced channel in a circular transaction to the lowly balanced channel that caused it to rebalance. A channel is considered 'highly balanced' if i), the node has a balance in that channel that is greater than half of the total channel capacity and if ii), the node's balance covers the amount required to bring the lowly balanced channel up to 20 % of the total channel capacity. Additionally, a suitable cyclic route needs to be found. The authors use the routing algorithm of `lnd` to find suitable cyclic transactions.

For the on-the-fly strategy, the authors adjust the fee policies of the nodes such that the "...fee amount is inversely proportional to channel balance". As CLoTH is build to simulate `lnd`, the routing algorithm used ranks its routes on the lowest amount of fees required[1]. Lowering fees if a channel is imbalanced can therefore result in the routing algorithm including a channel more often. The authors argue that this might change the flow of transactions in such a way as to rebalance the channel. After simulating both strategies, the authors conclude that the on-demand strategy is ineffective as many attempts at circular payments fail due to a lack of capacity in the network and because a node does not always have a sufficiently highly balanced channel to correct the lowly balanced channel with. The on-the-fly strategy is more effective and reduces the probability of transaction failure by one fourth. However, the authors do not yet know if this on-the-fly strategy can reduce the probability even further or if this is the maximum achievable with such a strategy.

Imbalance measure and proactive channel rebalancing In a work by Pickhardt and Nowostawski [40], an on-demand rebalancing algorithm is presented based on using the Gini coefficient as a definition of imbalance of a node. The authors define the *channel balance coefficient* as in Equation 3.2.

$$\zeta_{(u,v)} = \frac{b(e_u)}{c(e)} \tag{3.2}$$

In Equation 3.2, u and v are two nodes sharing an edge e, e_u represents edge e of node u, $b(e_u)$ is the balance of u in edge e and $c(e)$ is the capacity of edge e. The authors also define the capacity as the sum of the balances, i.e. $c(e) = b(e_u) + b(e_v)$. Because of this, it also holds that $\zeta_{(u,v)} + \zeta_{(v,u)} = 1$. Based on this, the authors state that a node u is *balanced* if all its local channel balance coefficients $\{\zeta_{(u,v_1)}, \ldots, \zeta_{(u,v_d)}\}$ have the same value. The authors consider a node *unbalanced* when the local channel balance coefficients are unequal. The authors formalize this definition by defining the Gini coefficient in Equation 3.3. Note here that $U = n(u)$, i.e. the neighbours of node u.

[1] Among other factors, see Subsection 2.3.3.

$$G_u = \frac{\sum_{i \in U} \sum_{j \in U} |\zeta_i - \zeta_j|}{2 \sum_{i \in U} \sum_{j \in U} \zeta_j} \tag{3.3}$$

If $G_u = 0$, it means the node u is balanced while if $G_u = 1$, the node u is balanced in the most unequal way. Finally, for their algorithm the authors also define the *node balance coefficient* as Equation 3.4.

$$\nu_u = \frac{\sum_{e \in U} b(e_u)}{\sum_{e \in U} c(e)} \tag{3.4}$$

The rebalancing algorithm as defined by the authors runs on every node and starts as soon as G_u rises above a certain threshold. A node u then uses ν and ζ to select a set of candidate channels C for which $\{(u, v_i) | \zeta_{(u,v_i)} - \nu_u > 0\}$ and selects a random channel e from C. u then finds a circular transaction to itself with as outgoing edge e and as incoming edge an edge from the set of $\{(u, v_i) | \zeta_{(u,v_i)} - \nu_u < 0\}$. The participants of this circular transaction only participate if this also improves *their* Gini coefficient G.

The authors state that the amount of the transaction should be $a = c(e) \cdot (\zeta_{(u,v)} - \nu_u)$ as this has the effect of decreasing the value of $\zeta_{(u,v)}$ to ν_u. If a is not possible due to a capacity bottleneck, u settles for any amount smaller than a. The algorithm is repeated as long as G_u is above the threshold and transaction paths are available.

To find circular transaction paths, the authors implemented four different strategies. The first two strategies test all cycles in the neighbourhood of lengths smaller than four or five. The second-to-last strategy `foaf` tries to find cycles inside the friend-of-a-friend network and the last strategy `mpp` does the same although with a 20^{th} of the amount that `foaf` does, as it is supposed to spread out the rebalancing over multiple iterations of the algorithm. Running static simulations on a snapshot of the Lightning Network, the authors show that all strategies are successful at reducing the network imbalance (i.e. the average Gini coefficient of all nodes $\bar{G}$) although cycles of length 5 and `foaf` are most efficient in doing so. The authors do not provide a dynamic simulation to show the interaction with their algorithm and transactions taking place at the same time.

Other works related to rebalancing Engelshoven [11] presents a work containing two algorithms that, in conjunction with the *SpeedyMurmurs* [44] routing algorithm, allow a node to determine the fees of each channel to stimulate the usage or avoidance of using the channel in a multi-hop transaction. This has the net effect of rebalancing channels on the fly. Mercan, Erdin, and Akkaya [35] also introduce an on-the-fly rebalancing and routing algorithm but specifically focus on IoT devices and introduce the concept of *smart gateway selection*. This concept has IoT devices use multiple gateways to transact on a PCN, resulting in a more balanced network. Subramanian, Eswaraiah, and Vishwanathan [48] present an on-demand rebalancing algorithm for acyclic payment networks.

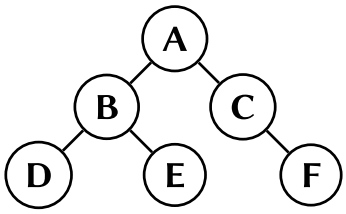
3.5 An introduction to Revive

Revive is an on-demand rebalancing algorithm presented by Khalil and Gervais [18] in 2017. As Revive is referred to often in this thesis, this section serves as an introduction to the work.

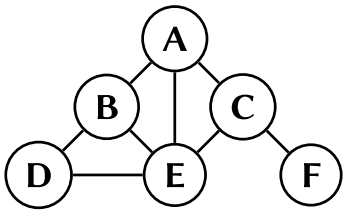
The intention of the authors with Revive was to introduce an on-demand rebalancing algorithm that requires no transaction fees and is PCN agnostic. They believe it is important to have a rebalancing algorithm without fees as the alternative strategy, which consists of lowering fees to have normal transactions rebalance a node's channel, can be considered a sacrificial strategy and reduces the incentive of users to take part in a PCN.

System model For the functioning of Revive, the authors require a blockchain supporting smart contracts, a PCN with payment functionality and an underlying, secure communication model between participants such as TLS over TCP. The authors note that the topology of the

PCN influences the success of their algorithm. This is because Revive finds *rebalancing cycles*, which do not exist in all networks (see Figure 3.3 for an example). The authors also note that some edges used in cycles can overlap if the channels visited multiple times have sufficient capacity, as might possibly be the case with edge B-D in Subfigure 3.3b.



(a) An example network with a tree structure. Since no cycles exist in the graph, no viable rebalancing transactions can be found.



(b) An example network containing cycles. Rebalancing payments can take paths such as: (A, B, D, E, A) and (C, E, B, D, E, A, C).

Figure 3.3: Two networks with different topologies, influencing the available rebalancing transactions. (Images and captions from Khalil and Gervais [18])

Threat model The authors state that Revive is designed to prevent any honest participant from losing any funds (i.e. provide *balance conservation*) despite the presence of an irrational adversary that is willing to lose all of their committed funds. This adversary can "...cause parties under its control to sign and authorize any set of messages using their identities, or front-run any user input, but may not violate the integrity of the keys honest protocol participants use. In addition, we assume an adversary can cause denial of service attacks that abort the protocol at any given point". [18]

3.5.1 Algorithm

The rebalancing algorithm starts with a leader election from a fixed set of participants. The leader does not have to be a participant as the leader only serves as a shared platform for synchronisation and computation. At the start of the algorithm, the leader is chosen as the participant with the smallest public identifier $\mathrm{ID}(p)$. For every other execution of Revive after the first, the participant with the next smallest ID is chosen until there are no new participants, at which point the first leader is elected again. If we assume that the public identifier cannot be changed after the set of participants has been fixed, the adversary is unable to always be selected as the leader, as this strategy makes every participant a leader an equal amount of times. The authors note that the other parts of Revive do not depend on the exact mechanism of the leader election and it can therefore be changed to suit the use-case of the participants, e.g. the participants might have a specific node that they all trust who is always elected as the leader.

After the initial leader election phase, the algorithm continues as in Figure 3.4. At first, the leader waits until enough participants have signalled that they wish to rebalance. The authors argue that this is useful for scalability as this allows one to set a threshold for how many participants are required to continue the rest of the algorithm. Once this threshold is passed, the leader sends a rebalancing initiation request to all participants. The participants which wish to partake then send a confirmation to the leader, who compiles a list of all participants P_t. After compiling P_t, the leader sends P_t to all $p \in P_t$ such that each p knows who is partaking in this iteration of the algorithm. Additionally, the leader requests all $p \in P_t$ to freeze transactions on channels they wish to rebalance.

The authors expect participants who wish to freeze their channels to do so in cooperation with the other channel owner, i.e. one of their neighbours. Participants also have to cooperate on defining a suitable *rebalancing objective* or *rebalancing demand*, which the authors write as $\Delta_{u,v}$. $\Delta_{u,v}$ represents the balance that node u wishes to gain in its channel with v. In contrast, $\delta_{u,v}$ represents the balance that node u *is going to* gain in its channel with v as a result of the rebalancing algorithm. Once both channel owners decide on a Δ, it holds that $\Delta_{u,v} = -\Delta_{v,u}$.

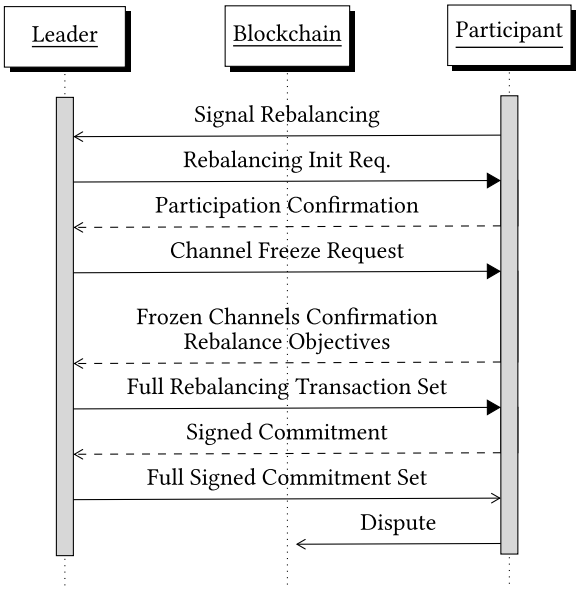


Figure 3.4: Protocol sequence diagram of Revive (after leader election). (Khalil and Gervais [18])

Linear Program

The next step of the algorithm involves the participants sending back their $\Delta_{u,v}$ to the leader. Each Δ represents both a value *and a direction* in which the coins should flow through a channel. This allows the leader to create a directed graph of the sub-network of participants, replacing each channel with a directed edge with weight and direction representing Δ. In this directed graph, a linear program solver[2] then tries to find a set of transactions that solve the specified Linear Program:

> **Linear Program**: Maximize: $\sum_{u,v} \delta_{u,v}$ Subject to:
> (1) $\forall u, v : \Delta_{u,v} > 0 \wedge \Delta_{v,u} < 0 \iff 0 \leq \delta_{u,v} \leq \min(\Delta_{u,v}, -\Delta_{v,u})$
> (2) $\forall u : \sum_v \delta_{v,u} = \sum_v \delta_{u,v}$

The Linear Program tries to maximise the number of coins moved between channels under the constraint that the amount of coins going through a channel has to be equal or less than $\min(\Delta_{u,v}, -\Delta_{v,u})$ and larger or equal to zero (1), and the total balance of each node stays the same s.t. no node can lose or gain coins as a result of the rebalancing (2).

The authors remark that if $\delta_{u,v}$ is defined, $\delta_{v,u}$ is not. This is because if both were defined, it would break the semantics of the linear program. The authors also assume that all participants are honest about their Δ being smaller than the total funding of the channel. As a final remark, the authors warn that linear programs can produce non-integer solutions. This is a problem as when the lowest unit of a global ledger is in the range of 0–9, a fractional result might not be something that the ledger may be able to represent. The authors argue that this can be solved by rounding down each fractional result to an integer, as the US dollar price of the lowest currency unit of Bitcoin (1 Satoshi = 10^{-8} BTC) and Ethereum (1 Wei = 10^{-18} ETH) is very small.

Finalizing the algorithm

At this point, the linear program has produced a transaction set. The transaction set, a list of participating members and a commitment by the leader is then sent to all $p \in P_t$ for verification and signing. The commitment takes the form of a Merkle-tree [36] that covers all transactions in the transaction set as well as a hash of the public addresses for all $p \in P_t$. Each participant p then verifies the commitment and transaction set, signs the commitment and sends it back to the leader. Once the leader obtains all signatures, it multicasts these to all $p \in P_t$. At that point, the

[2]Many Linear Program solvers exist. See for example lp_solve.

authors state that all channels can be unfrozen because the signed transaction set is considered binding. Using the fully signed commitment of the transaction set as proof, each node can then claim the coins of incoming transactions.

In a later analysis of their algorithm, the authors argue that the linear program produces more optimal results if run by one leader for the entire network. However, a global leader requires extensive coordination by the entire network and so this scenario is probably unrealistic. Instead, the authors propose that a PCN should run many local, sub-optimal instances of Revive to hopefully produce a local optimum close to the global optimum.

3.5.2 Performance evaluation

Although the authors provide a proof-of-concept implementation and scaling issues are discussed, no experimental evaluation is done on a PCN in order to study the effect of Revive on the probability of transaction success in a PCN. This makes it difficult to evaluate if it is worthwhile to scale Revive to a point of it being used by the majority of the PCN to rebalance channels or whether at that point it would be better for the probability of transaction success if users do not use on-demand rebalancing and simply close and open channels with new funding once a channel runs out of funding.

3.5.3 Balance conservation

An important property for any on-demand rebalancing protocol is that it provides *balance conservation*. Balance conservation is the property that the total balance of participant u owned in all its channels E_u is the same before and after the, potentially unsuccessful, execution of the protocol. More informally, guaranteeing balance conservation ensures that a participant can never lose its coins or gain any coins by taking part in the execution of the protocol.

In Revive, the authors guarantee balance conservation by making each participant verify and sign the transaction set that the leader has generated. If the transactions in the transaction set do not result in a net sum of zero for the participant that is verifying it, the participant can refuse to sign and commit to the transaction set. Then, only when the participants receive a transaction set signed by all participants, they can be assured that all other participants have committed to the rebalancing. If a participant commits but does not execute the transactions it committed to, the authors provide a smart contract dispute settlement mechanism in order to guarantee that honest participants always receive what the dishonest participant committed to, preserving balance conservation or resulting in a loss in coins for the dishonest participant and a gain in coins for the honest participant.

This dispute settlement mechanism also allows for an alternate problem where a participant who committed never receives the fully signed transaction set because the adversary controls all the other participants and the leader. The adversary can then claim all outgoing coins of the victim while the victim cannot claim any of the incoming coins, breaking the balance conservation property. In that case, the victim can issue an on-chain availability challenge for the fully signed commitment set using the smart contract. If the signed transaction set is not provided within a certain amount of time, all rebalancing transactions will become invalid. The authors note that for this to work, transactions can only be finalized after a certain deadline t. After t, no availability challenge can be started. As the solution involves an on-chain transaction, the authors state that this has the effect of introducing a monetary cost for the victim when protecting its coins. According to the authors, the total cost in the worst-case scenario where all honest participants have to issue an availability challenge increases proportionally to the number of participants.

3.5.4 Objective satisfiability

The authors state that there exists a risk that if the leader is the adversary and also a participant, it can generate a transaction set using a modified linear program that favours its own channels or interests. As a solution, the authors suggest using a modified version of Revive in which *each* participant $p \in P_t$ solves the linear program as a form of multi-party computation, instead of only the leader. This would require all information regarding rebalancing objectives to not only be sent

to the leader but also to all P_t, along with a random seed. The authors note that this solution comes at the cost of privacy and efficiency but might be a worthwhile trade-off in some use cases.

3.5.5 Privacy

A major part of the design of Revive is that the leader needs to know the latest state of each payment channel in order to calculate the transaction set. The authors acknowledge that this is a privacy leaking component of Revive and that a proper privacy analysis has not been done. We provide such a privacy analysis in Subsection 4.2.1 after defining the relevant security and privacy definitions in Section 4.1.5.

Chapter 4

Design

In this chapter, our payment channel network (PCN) model is formally defined after which we introduce our notation and security and privacy definitions. We then discuss the problems with Revive and formulate design requirements for our on-demand rebalancing protocol based on these problems. In the later sections of this chapter, we first present the design of the participant discovery protocol and then the design of the rebalancing protocol. Finally, in the last part of this chapter we analyse the security and privacy of our designs.

4.1 Formal definitions

Let a PCN be defined as an undirected multigraph[1] $G = (V, E)$ with a finite set of nodes. Each node represents a user of the PCN. Each payment channel is represented by an undirected edge $e_{u,v}$ between two nodes u and v such that $E \subset V \times V$. For each edge with associated nodes u and v, we define a balance function $b(e_{u,v}, v)$ that returns the balance of v in channel $e_{u,v}$ and a capacity function $c(e_{u,v}) = b(e_{u,v}, u) + b(e_{u,v}, v)$. We denote with $N(u)$ the set of all the neighbouring nodes of u where a neighbour i is defined as a node with which u shares an edge $e_{u,i}$. We define E_u as the set of edges involving node u. Furthermore, we define a path p as a sequence of edges $e_1 \dots e_n$ where $e_i = (v_i^1, v_i^2)$ and $v_{i+1}^1 = v_i^2$ for $1 \leq i \leq n-1$ (similar to [44]). See Subfigure 4.1a for a visualisation of G.

4.1.1 Functionality

In general, we assume the functionality of our PCN model to be similar to existing payment networks such as Lightning or Raiden. We assume there exists an interface provided by our PCN model that consists of three functions: $\texttt{pay}$, $\texttt{pay}_{\text{cond}}$ and $\texttt{pay}_{\text{exec}}$. We define $\texttt{pay}$ as $\texttt{pay}(s, r, x)$ where s is the sender, r the receiver and x the amount to be transferred over the edge $e_{s,r}$. We extend $\texttt{pay}$ to conditional payments by defining $\texttt{pay}_{\text{cond}}(s, r, x, \varphi_{\text{setup}})$ and $\texttt{pay}_{\text{exec}}(r, s, x, \varphi_{\text{exec}})$. In $\texttt{pay}_{\text{cond}}$ and $\texttt{pay}_{\text{exec}}$, φ_{setup} represents the necessary information to setup a conditional transaction and φ_{exec} represents the information required to fulfil the condition of the transaction and execute it. If HTLC transactions are used, φ_{setup} would include a cryptographically hashed preimage and the time-lock for the transaction. φ_{exec} would then represent the preimage that is to be revealed to the payee. For a sender s to do a conditional payment to a receiver r, it must execute $\texttt{pay}_{\text{cond}}$ to create a conditional transaction y on the condition that it may only be executed if r presents φ_{exec}. If r then wishes to fulfil the condition, it must call $\texttt{pay}_{\text{exec}}$ with the correct φ_{exec} to complete the transaction.

We also define functions $\texttt{pay}_{\text{condP}}(s, r, x, p, \varphi_{\text{setup}})$, $\texttt{pay}_{\text{execP}}(r, s, x, p, \varphi_{\text{exec}})$ and $\texttt{pay}_{\text{condComp}}(s, r, x, p)$ that represent conditional payments over a path p with i intermediaries between s and r. $\texttt{pay}_{\text{condP}}$ consists of the repeated execution of $\texttt{pay}_{\text{cond}}$, e.g.

$$\{\texttt{pay}_{\text{cond}}(s, i_1, x, \varphi_{\text{setup}}), \texttt{pay}_{\text{cond}}(i_1, i_2, x, \varphi_{\text{setup}}), \dots, \texttt{pay}_{\text{cond}}(i_n, r, x, \varphi_{\text{setup}})\} \quad ,$$

[1] Meaning that there might be more than one edge between two nodes

If it happens that a $\texttt{pay}_{\text{cond}}$ fails because of a lack of funds in the channel, any subsequent $\texttt{pay}_{\text{cond}}$ will not be called. It is also important to note that although φ_{setup} does not appear to change from one $\texttt{pay}_{\text{cond}}$ to the other, in practice small variations might exist between each φ_{setup}. This is the case with HTLC payments where the time-lock gets smaller from one conditional payment to the next. Finally, we assume intermediaries ask no fees for facilitating transactions. Even though fees are an essential part of the functioning of normal multi-hop transactions, we consider nodes that participate in our rebalancing protocol to require no fees for transactions that are done as part of our protocol. This is because in our protocol, a node only needs to facilitate transactions if one of its channels gets rebalanced, thereby always providing an incentive for the node to facilitate the transaction.

$\texttt{pay}_{\text{execP}}$ functions similarly to $\texttt{pay}_{\text{condP}}$, but with $\texttt{pay}_{\text{exec}}$ instead of $\texttt{pay}_{\text{cond}}$. Finally, $\texttt{pay}_{\text{condComp}}$ is defined as a function that generates a φ_{setup} and φ_{exec} and invokes $\texttt{pay}_{\text{condP}}$ and $\texttt{pay}_{\text{execP}}$ to do a complete conditional transaction over a path p between s and r. The functionality of $\texttt{pay}_{\text{condComp}}$ is similar to the experience of a user that leaves the handling of the conditional payments to a client interacting with the PCN and only specifies the path[2] p, amount x and receiver r (the user itself being s).

4.1.2 Rebalancing

We denote the *demand* or *rebalancing objective* of an edge $e_{u,v}$ as $\vec{\Delta}_{u,v}$. Note that contrary to $e_{u,v}$, $\vec{\Delta}_{u,v}$ is directional meaning that it defines both the amount of balance shift and which direction the channel balance should be shifted. We say that $\vec{\Delta}_{u,v}$ represents a balance shift of amount Δ *from* u *to* v. We define an interface of a negotiation protocol $\texttt{negotiateObjective}(e_{u,v})$ that is run by the two nodes u and v of an edge $e_{u,v}$ when a $\vec{\Delta}$ is required and returns either a $\vec{\Delta}$ if it succeeds or $\perp$ if it fails. If the negotiation protocol does not fail, it holds that $\vec{\Delta}_{u,v} \neq \vec{\Delta}_{v,u}$ and $\vec{\Delta} \geq 0$.

We define a *rebalancing graph* as a directed weighted multigraph $R = (V, Q)$ where Q represents a set of q rebalancing edges. R is a sub-graph of G where all undirected edges in E with a zero or $\perp$ value for $\vec{\Delta}$ are removed. All $e \in E$ with a non-zero $\vec{\Delta}$ are replaced with a single directed edge q with a direction and weight $w(q)$ taken from $\vec{\Delta}$. See Subfigure 4.1b for a visualisation of R. We define Q_o^u as all outgoing edges $q_{u,i}$ for node u, Q_i^u as all incoming edges $q_{i,u}$ for node u and $Q^u = Q_o^u \cup Q_i^u$. We also define an opposite function $\texttt{Opp}(\{e|q\}^1, u)$ that takes as input an edge e or q and a node u which is one of the owners of the edge, and returns the other owner of the edge.

We define a *rebalancing cycle* as a cycle of q rebalancing edges in R. We define the incoming edge of a cycle c in a node u as $q_i^{u,c}$ and the outgoing edge as $q_o^{u,c}$.

4.1.3 Pseudocode utilities

In this section, we define the special notation we use in our pseudocode to denote certain data types and structures.

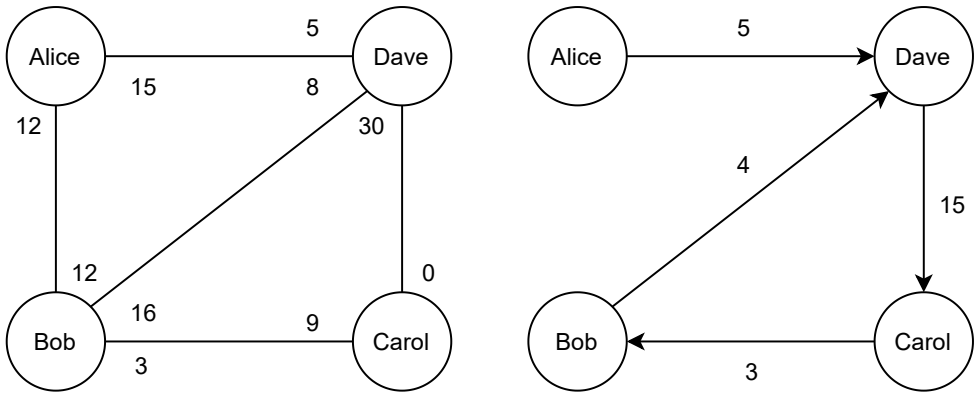
First, we define a primitive $\perp$ which represents a null character often found in many programming languages. We also define a *boolean* which is a variable that can either take the values true or false.

We also define a *dictionary* or *map* as a set D_i consisting of (k, v) key-value pairs where K represents the set of keys and $\mathcal{V}$ the set of values. The i refers to a specific dictionary or map. For example, $D_{t_r,q}$ refers to a map mapping $T_r \to Q$. We define a function $\texttt{get}_i : K \to \mathcal{V} \cup \{\perp\}$ where i denotes a specific set D and $\perp \notin \mathcal{V}$. The functionality of $\texttt{get}_i$ is defined as

$$\texttt{get}_i(k) = \begin{cases} v & (k, v) \in D_i \\ \perp & \text{else} \end{cases} . \tag{4.1}$$

We define two other functions $\texttt{put}_i : K \times \mathcal{V} \to \perp$ and $\texttt{remove}_i : K \to \perp$. Calling $\texttt{put}_i(k, v)$ causes $D_i \leftarrow D_i \cup \{(k, v)\}$ to take place. Calling $\texttt{remove}_i(k)$ causes $D_i \leftarrow D_i - \{(k, v)\}$ to take

[2]Client implementations for PCNs such as Lightning typically also provide the path, see Subsection 2.3.3



(a) An example of a payment channel network with undirected edges representing payment channels. The number on the right of an edge as seen from the perspective of the node represents the balance of a node in that edge, e.g. Bob and Dave share an edge in which Bob has a balance of 16 and Dave a balance of 8.

(b) An example of the rebalancing graph as based on 3a). In this example, `negotiateObjective` is implemented such that it produces a $\vec{\Delta}$ that causes the channel to become equally balanced, e.g. if Alice and Dave have a balance of 15–5 in their channel, Alice needs to transfer 5 coins to Dave such that the channel has a balance of 10–10. The channel between Alice and Bob disappears as their channel is already equally balanced.

Figure 4.1: An example of a PCN as an undirected graph G and a rebalancing graph R.

place. We also define a shorthand $k \in D_i$ which is equivalent to $k \in \{k' \mid (k', v') \in D_i\}$. The functionality provided by these functions is a limited version of the functionality provided by a `dict` in Python or a `HashMap` in Java.

We denote a *sequence* or *list* of values $a \in \mathcal{V}$ as $L_i = [a_0, a_1, \ldots, a_{n-1}]$. A sequence has an *order* that equals the order of value addition to the sequence. Contrary to a set J, duplicates are allowed in a sequence.

We denote with $|L_i|$ the size of the sequence L_i and with $L_i[j, k]$ a specific range $[j, k)$ of values $[a_j, a_{j+1}, \ldots, a_{k-1}]$ in L_i. If we write $L_i[j]$, we mean to indicate the j^{th} value of the sequence L_i. We denote with $L_i \leftarrow L_i * a$ the addition or concatenation of a to L_i. We write subtraction or removal of a from L_i as $L_i \leftarrow L_i - a$, where we clarify that this removes *all* instances of a in L_i. We also define a function $\texttt{indexOf}_i : \mathcal{V} \to \mathbb{Z}^+ \cup \{\perp\}$ for L_i. Calling $\texttt{indexOf}_i(v)$ returns the index j of the first value that matches $L_i[j] = v$ starting from index 0 until $|L_i|$.

We denote a predicate as $Pr : \mathcal{V}^k \to \{\text{true}, \text{false}\}$ where $\mathcal{V}^k$ represents k inputs. It is required that $k \geq 0$. A predicate is a predefined statement such as $Pr(v) \leftarrow a \wedge v$ which can be evaluated to produce a boolean.

Finally, we define `check-action` notation for the repeated invocation of predicates that is common throughout the pseudocode of the transaction generation protocol. The `check-action` notation defines an ordered sequence of predicate-action pairs and is written as: '**check-action**: $Pr_i \to$ **action**$_i$, $Pr_j \to$ **action**$_j$, ...' It should be read as 'If Pr_i is true, carry out **action**$_i$, else if Pr_j is true, carry out **action**$_j$, etc.'.

4.1.4 General assumptions

During the design and simulation, we assume a static G. This means that no nodes and channels are removed or added to G during the running of the simulations. Little research has yet been done into simulating the dynamic nature of nodes joining a PCN and creating channels [11] and we consider this outside the scope of this thesis.

As we take our PCN G to have functionality `pay`, $\texttt{pay}_{\text{cond}}$ and $\texttt{pay}_{\text{exec}}$, we acknowledge that for its secure operation we also have to assume the existence of a global ledger $\mathcal{G}_{\text{Ledger}}$ and the existence of cryptographic primitives (such as digital signatures and key generation) that are necessary for the functionality to be implemented.

We assume an asynchronous network model where a node has one communication channel per edge it owns. We assume no global ordering of messages but require that messages are received from a channel in the order that they were send by the sender. We also assume there are no lost and corrupted messages between honest users.

4.1.5 Security and privacy

PCNs and the algorithms running on them have certain security and privacy properties that are often referred to in literature discussing the topic [30, 31, 54]. In this section, we define these security and privacy properties and provide a threat model and the assumptions with which we evaluate them.

Threat model

We consider a malicious rational computationally bounded adversary that is a participant of G. The adversary can shape G at will by creating or corrupting k nodes in G and by establishing new channels from corrupted nodes to other nodes. The set of corrupted nodes is fixed from the moment the protocol starts. If a node is corrupted by the adversary, the adversary gains access to its state, message history and receives all new messages intended for them. Once a node is corrupted, the adversary can create arbitrary messages that originate from the corrupted node. We call a node that has not been corrupted by the adversary a non-corrupted or *honest* node.

Assumptions

We assume that the implementations of $\texttt{pay}$, $\texttt{pay}_{\text{cond}}$ and $\texttt{pay}_{\text{exec}}$ leak no information to anyone outside of the sender s and receiver r.

We also assume that $\texttt{pay}_{\text{condComp}}$ provides *balance security* for all honest intermediaries i in a path p between s and r. We argue that this is a valid assumption as this is the same functionality as provided by Lightning HTLC multi-path transactions, which are proven to guarantee *balance security* [19]. This assumption also implies that the underlying implementation of $\texttt{pay}_{\text{cond}}$ and $\texttt{pay}_{\text{exec}}$ contain safeguards that allow for the assurance of balance security. In the case of Lightning, this would imply that executing $\texttt{pay}_{\text{exec}}$ reveals φ_{exec} = preimage to both channel owners and that conditional transactions expire in sequence such that the an intermediary always has time to claim its coins. See Subsection 2.3.1 for a full explanation of how multi-hop transactions work in Lightning.

We also assume that all the communication channels of the nodes are made confidential by using a secure communication scheme such as TLS. This reflects current PCN implementations such as Lightning (BOLT #8, [25]).

Definitions

The following section provides the formal definitions of the security and privacy properties discussed in this thesis. Many of these properties we define have already been defined in related work [30, 31, 44, 54] (either formally or informally) but we reformulate them here in the context of our assumed functionality of the PCN and its methods.

Please note that after each definition, we cite the works that helped shape it.

Definition 1 (Balance security)**.** *For a $\texttt{pay}_{condComp}(s,\ r,\ x,\ p)$ where s is the sender, r the receiver, x the amount to be paid and the path p holds i intermediaries, a $\texttt{pay}_{condComp}$ operation provides balance security if there exists a negligible probability that the adversary can make any honest intermediary lose its coins that are staked as part of the $\texttt{pay}_{condComp}$ operation. [30]*

Definition 2 (Balance conservation)**.** *For an honest participant i partaking in the execution ν of a rebalancing algorithm Λ_{re} with only honest participants, we say that Λ_{re} guarantees balance conservation to i if Equation 4.2 holds, where Q^i_{before} are the rebalancing edges of i before the execution ν and Q^i_{after} the rebalancing edges of i after the execution ν.*

$$\sum_{q \in Q^i_{before}} w(q) = \sum_{q \in Q^i_{after}} w(q) \tag{4.2}$$

Definition 3 (Participation anonymity). *For a set of honest nodes $V_h \subseteq V$ partaking in the execution ν of a distributed algorithm Λ_d, we say that a node $i \in V_h$ has participation anonymity if i has node participation anonymity and all its edges E_i have edge participation anonymity.*

Definition 4 (Node participation anonymity). *For an honest node i partaking in the execution ν of a distributed algorithm Λ_d with anonymous identity A_i and identity $ID(i)$, we say that i has node participation anonymity if the adversary can determine with negligible probability if i is partaking in execution ν of Λ_d.*

Definition 5 (Edge participation anonymity). *For an edge $e_{u,v}$ between two honest nodes u and v where both nodes are partaking in the execution ν of a distributed algorithm Λ_d, we say that $e_{u,v}$ has edge participation anonymity if the adversary can determine with negligible probability if $e_{u,v}$ is part of execution ν of Λ_d.*

Definition 6 (Channel balance privacy). *For a channel $e_{u,v}$ in G with balance functions $b(e_{u,v}, u)$ and $b(e_{u,v}, v)$, and u and v being honest nodes, we say that channel $e_{u,v}$ has channel balance privacy if there exists a negligible probability that the adversary can determine $b(e_{u,v}, u)$ or $b(e_{u,v}, v)$. [54]*

Definition 7 (Path privacy). *For a $\texttt{pay}_{condComp}(s, r, x, p)$ where s is an honest sender, r a receiver, x the amount to be payed and path p holding n intermediaries of which at least one is honest, a $\texttt{pay}_{condComp}$ operation provides path privacy if the adversary can determine (parts of) the path p outside of the nodes it already corrupted with a probability less than or equal to $1/\left|\bigcap_{i \in C_p} P_i\right|$, where C_p is the set of corrupted nodes which lay in the path p and P_i represents all possible paths going through node i. [54]*

Definition 8 (Value privacy). *For a $\texttt{pay}_{condComp}(s, r, x, p)$ where s is an honest sender, r an honest receiver, x the amount to be payed and p being a path consisting of* only *honest intermediaries, a $\texttt{pay}_{condComp}$ operation provides value privacy if there exists a negligible probability that the adversary can determine x. [31, 44, 54]*

Definition 9 (Sender/receiver privacy). *For a $\texttt{pay}_{condComp}(s, r, x, p)$ where s is an honest sender, r an honest receiver, x the amount to be payed and the path p holds n intermediaries of which at least one is honest, a $\texttt{pay}_{condComp}$ operation provides sender/receiver privacy if the adversary can determine the sender/receiver of $\texttt{pay}_{condComp}$ with a probability less than $1/|V - C|$ where C is the set of corrupted nodes. [31, 44, 54]*

Definition 10 (Relationship anonymity). *For two simultaneous $\texttt{pay}_{condComp}(s, r, x, p)$ operations between sender/receiver pairs (s_1, r_1) and (s_2, r_2) with the same amount x and the same path p holding n intermediaries of which at least one is honest, we say a $\texttt{pay}_{condComp}$ operation provides relationship anonymity if an adversary can determine if s_1 pays r_1 or s_2 pays r_2 with a probability less than half. [30, 49]*

4.2 Problems of Revive

The major goals of on-demand rebalancing protocols are to allow its participants to rebalance their channels effectively, efficiently and securely. If these were the only goals, a protocol such as Revive [18] can possibly be a good fit for the problem. However, we argue that besides effectivity, efficiency and security, an on-demand rebalancing protocol should also protect the privacy of its users. This is in line with research into making PCNs more privacy-preserving (see [13, 15, 30]) and is stated to be a goal of choices made in the construction of the Lightning Network (BOLT #4, [25]). We believe that it is, therefore, strange to use an on-demand rebalancing protocol that does not provide appropriate privacy guarantees to its users on a PCN that is built to do so.

In this section, to improve upon Revive, we analyse the privacy properties and guarantees of Revive. We furthermore highlight the properties Revive does not achieve. In addition, we argue

that if it is to be implemented in an actual PCN, its dependency on a fixed set of participants needs to be reconsidered to support a varying set of participants, which would require a participant discovery protocol. We provide an overview of how Revive works in Section 3.5.

4.2.1 Privacy

Revive is designed in such a way that it can achieve *balance conservation* (Definition 2) despite relying on a single non-trusted leader. This requires that participants validate the transaction set generated by the leader. If the transaction set passes validation, it is signed by the participants, which acts as a commitment to the rebalancing transaction. The participants execute their transactions after receiving a transaction set signed by all participants. In our PCN model, this last step is equivalent to a $\texttt{pay}(s, r, x)$ operation where s, r, and x are taken from the transaction set.

Although this scheme achieves balance conservation, it also exposes many details about a participant u and its channels Q_u to the leader and other participants even though u might not want to share this information. The authors acknowledge that this is a problem with their protocol. If the adversary is one of the participants, it can derive the following information from the transaction set sent around by the leader for validation: i) transaction sender, ii) transaction receiver and iii) transaction amount.

As the adversary knows the sender and receiver of every transaction, this prevents the protocol from obtaining *sender/receiver privacy* (Definition 9) and *relationship anonymity* (Definition 10). As the adversary always knows the transaction amount, there is no *value privacy* (Definition 8). Because the transaction set only contains `pay` operations, it must be that there are no intermediaries between the sender and receiver. This breaks *path privacy* (Definition 7) as it is known to the adversary that the sender pays the receiver through a path $\{s, r\}$.

If the adversary is the leader or if all participants calculate the linear program in order to guarantee a fair solution (as suggested by the authors but with a decrease in privacy), the adversary would also have access to all the rebalancing demands $\vec{\Delta}$ of every participating channel e. If we then assume that the adversary knows the `negotiateObjective` protocol of each pair of channel owners, the adversary can use the workings of the protocol and the channel's $\vec{\Delta}$ to derive the original channel balance of e, breaking *channel balance privacy* (Definition 6). The assumption that the `negotiateObjective` protocol is known to the adversary is in line with our expectation of negotiation protocols to be relatively simple, such as one that produces equal balances.

A possible improvement on the leader-focused design of Revive is to augment it with a Secure Multi-Party Computation (SMPC) linear program solver as described in [50]. With an SMPC linear program solver, each participant could participate in the SMPC protocol to obtain the final transaction set with the leader only coordinating the computation but being unable to see the actual inputs to the program. However, this would not solve the privacy problem concerning the distribution of the generated transaction set for validation and signing.

4.2.2 Leader election and finding participants

Revive requires a fixed set of participants in its leader election in order to ensure that all participants are chosen as often as any other participant to become the leader. We consider this requirement to be unrealistic as in a PCN, channels may close or open, nodes may disappear, etc. This would prevent the execution of Revive. However, the authors suggest that the leader election phase of the protocol can be changed to incorporate different scenarios than they envisioned. For example, if we assume a set of participants that varies every execution and use the same leader election as specified by the authors, the participant p with the smallest public identity $\text{ID}(p)$ will always be chosen. However, if then the adversary manages to obtain or create a node with the smallest $\text{ID}(p)$, they will always be elected as the leader in any executions of Revive they participate in. As the leader can influence the outcome of the obtained rebalancing solution or deny participation to certain users, it is important to avoid malicious influence on the leader election. In their work, the authors provide no solution for this problem.

Switching from a fixed set of participants to a varying one also raises the question how such a set is found each execution of the protocol, implicitly adding the requirement for another protocol that carries out such a task. Such a protocol was not included in the authors work for Revive.

4.3 Requirements

To improve upon Revive and answer our research question, we can deconstruct a design of a suitable alternative into a set of requirements that such a design should follow. These requirements guide the design process and allow for later evaluation of the protocol after its design. The requirements are divided into MUST HAVES and SHOULD HAVES as follows:

1. **Must haves:**
   (a) **Transaction set**: The protocol must find a non-trivial (i.e. non-zero) transaction set among its participants that meets some rebalancing demands
   (b) **Balance security and conservation**: The protocol must have a negligible probability that an honest participant can lose coins in a channel without its consent
   (c) **Privacy** The protocol must have the following privacy properties:
      i. Sender/receiver privacy
      ii. Relationship anonymity
      iii. Value privacy
      iv. Path privacy
      v. Channel balance privacy
   (d) **Participation**: Nodes must be able to discover other nodes willing to participate in the protocol
2. **Should haves:**
   (a) **Optimality**: The protocol should find the optimal transaction set among its participants that meets as many rebalancing demands as possible
   (b) **Concurrency**: The protocol should allow participants to continue accepting transactions on channels participating in the protocol

4.3.1 Motivation

Requirement 1a embodies the minimum functionality of the new rebalancing protocol. The protocol is an on-demand protocol and when invoked, similarly to Revive, it produces a set of transactions that can be executed to rebalance channels among this set of participants. It is, therefore, a must-have that our protocol produces a similar result to Revive while explicitly not specifying how this transaction set must be *generated* or how it must be *executed*, given that these aspects are not required to be constrained and provide design freedom and potential privacy improvements over Revive.

Requirement 1b embodies the balance security and conservation property of the protocol. We consider this an essential part of any protocol running on a PCN that deals with multi-path transactions, as the Lightning Network itself already guarantees balance security with their current implementation [19] and as it is also a property that is covered in many works concerning the secure construction of multi-path transactions [28, 30, 51].

Requirement 1c embodies almost all privacy aspects as defined in Subsection 4.1.5. Many of these privacy properties have been a requirement in the design of transaction routing protocols [13, 31] and are mentioned as motivation for the Lightning RFC to adopt an onion-like source routing protocol (BOLT #4, [25]). The inclusion of these privacy aspects into our design is the differentiating factor between Revive and our rebalancing protocol and is, therefore, the reason we include them as a must-have. We are also interested to see what impact including more privacy has on the ability of the design to meet the rebalancing demands and as a result, the probability of transaction success.

Requirement 1d embodies the need for a protocol that can assemble a set of participants for an execution of another distributed protocol. In Revive, the authors assume this set of participants is fixed but in Subsection 4.2.2 we argue that this is not a realistic requirement. We, therefore, find it important to include such a participant discovery protocol in our design as a must-have, as it is

necessary for a complete protocol design as well as for implementing Revive in a fair comparison with our design.

Requirement 2a embodies one of the distinguishing features of Revive which is its optimality of the generated transaction set. As Revive uses a linear program solver with a linear program, it is essentially solving an optimization problem [21]. Assuming that the problem is well defined and constrained, the outcome of the solver is the optimal solution to the problem. This means that in the case of Revive, a more optimal transaction set for rebalancing does not exist. We, therefore, include Requirement 2a as we strive to design a protocol that comes close to the optimality of the generated transaction set as produced by Revive.

Finally, Requirement 2b embodies a concurrency improvement over Revive. In Revive, participants are requested to freeze the channels that are involved in the protocol and in our model, this happens before initiating the `negotiateObjective` protocol. Once frozen, the channels only get unfrozen after the transaction set has been generated and signed by all participants. While frozen, no other transactions can take place over these channels. The channels can therefore not be utilized for the duration of the protocol to facilitate transaction routing. As the goal of rebalancing is to improve the network's balance and thereby improve transaction routing and the probability of transaction success, we deem it important that the execution of our rebalancing protocol does not negatively impact the potential routes available for transaction routing.

4.4 Overview

As a privacy-friendly alternative to Revive, we present a peer-to-peer on-demand rebalancing protocol. Our rebalancing protocol allows each participant to assume an anonymous identity before finding suitable rebalancing cycles from and to itself during multiple rounds. Once rebalancing cycles are detected, their start and end edges are only known to the participant that detected the cycle, also known as a *cycle owner*. Our rebalancing protocol then allows each participant to settle their transactions locally with their neighbours while providing a guarantee that the total balance of each participant stays the same.

We designed our rebalancing protocol as a peer-to-peer algorithm as this would force us to think about the minimum of information required to produce rebalancing transactions. The design is inspired by the distributed cycle detection algorithm of Rocha and Thatte [43], modified to suit our needs. To solve the privacy problem with the distribution of a generated transaction set as happens in Revive, we determined that a localized transaction settling mechanism would be required so that participants are only involved in the validation and execution of transactions that include them in their path.

In the following sections, an overview is provided of our participant discovery protocol and our rebalancing protocol, the latter hereafter called the transaction generation protocol in order to more accurately describe its purpose. Example invocations of both protocols can be found in Figure 4.2 (participant discovery) and Figure 4.4 (transaction generation).

4.4.1 Participant discovery

The participant discovery protocol takes as its input a node u in G that wants to find other nodes who wish to join in the execution of a distributed algorithm Λ_d with settings S. It outputs a set of participating edges E_p and a set of participants P, the latter of which is represented by anonymous identities A_u. An *anonymous identity* A_u is a random string of a fixed predetermined length and generated by a node u to be used as its identity in the execution of Λ_d. Using anonymous identities allows for participation anonymity during the execution of Λ_d.

In essence, the discovery algorithm creates a subgraph G_p of G with only the participants as vertexes and the participating edges as the graph's edges.

The participant discovery consists of three phases, the WAIT/INVITE, ACCEPT and FINAL phase. Every node starts in the WAIT/INVITE phase until it receives an INVITE from a neighbouring node. An INVITE contains the settings S for the execution of Λ_d as well as a hop count h_c variable that limits how far the INVITE can travel throughout G and an $I_{\max}$ variable that limits how many INVITEs may be sent by a node. The first invites are sent by the initiator, which

is the node that started the discovery protocol. If an INVITE is received by a node, it can decide to accept or deny it. Accepting an INVITE implies that the node agrees with the settings S for Λ_d and wishes to participate with edge e_i in its execution. Here, edge e_i is the edge over which the INVITE is received. A node might therefore ACCEPT some invites but DENY others if it does not want to participate with certain edges in the execution of Λ_d. If it accepts, it forwards the INVITE on all its edges it wishes to partake with during the execution of Λ_d and moves on to the ACCEPT phase.

When a node moves into the ACCEPT phase, it creates a set P_l which is a local view on P and initially only contains the anonymous identity of the node itself. If the node receives an ACCEPT message containing a similar set P_m, it adds the received P_m to its own P_l. Once a node received a reply (ACCEPT or DENY) to all its invites, it sends an ACCEPT containing P_l to the node that invited it initially and moves on to the FINISH phase. Once the initiator received a reply to all its invites, it also adds all the received P_m to its own P_l. P_l then becomes P and the initiator sends P using a FINISH message to all neighbours[3] who accepted the invites. Once this is done, the initiator starts the execution of Λ_d. If a node receives a FINISH message, it also forwards it to all neighbours who accepted *its* invites and starts the execution of Λ_d. Once all nodes in the FINISH phase received the FINISH message, we consider the discovery protocol terminated and Λ_d to be running on every participant.

4.4.2 Transaction generation

In the previous section, the participant discovery protocol ended by providing every participant with a set of participants P, a set of participating edges E_p and executing Λ_d on each participating node. In this section, we assume Λ_d to be the transaction generation protocol, which takes as its inputs a set of participants P and a set of participating edges E_p, and allows the participants to rebalance their channels in multiple rounds according to their rebalancing objectives.

Before the start of the first round, all participants sort P in a predefined way such that every participant obtains the same list of participants in the same order. The first participant in that list is then chosen as the leader for the first round, the second participant is the leader of the second round, etc. until round $\rho|P|$, after which the protocol terminates. ρ is a setting of the transaction generation protocol and defines the percentage of participants that should become a leader.

At the start of the first round, all participants run `negotiateObjective` with their participating neighbours to obtain a $\vec{\Delta}_{u,v}$ for each participating edge. In essence, this creates a rebalancing graph R from G_p with the same vertexes but with directed edges $q_{u,v}$ instead of the original undirected $e_{u,v}$. The $\vec{\Delta}_{u,v}$ is updated at the end of each round as rebalancing transactions are executed.

Each participant in a round goes through four phases[4]: WAIT/REQUEST, SUCCESS, COMMIT and EXEC.

REQUEST phase

The leader starts by sending a REQUEST on its outgoing edges. A REQUEST contains a set of randomly generated strings T_r, which are known as *cycle detection tags*. Each participant holds a set T_s which is the set of all cycle detection tags that the participant has seen.

The first time a participant u receives a REQUEST, it generates a unique cycle detection tag t_r^q for each $q_o \in Q_o^u$ and adds the T_r of the REQUEST to T_s such that $T_s \leftarrow T_s \cup T_r$. It then sends a REQUEST over each q_o, containing $T_r^{q_o} = T_s + t_r^{q_o}$ and moves to the SUCCESS phase.

SUCCESS phase

If the same participant u later receives an additional REQUEST containing a $t_r^{q_o} \in T_r$ it recognizes, they generate a cycle tag $t_c^{q_i}$ for the edge q_i they received the T_r on. Once generated, u sends a

[3] The actual protocol sends this along a tree to decrease the message complexity

[4] As the protocol is peer-to-peer, participants may be in different phases compared to each other. The protocol is designed to handle this gracefully.

SUCCESS message on q_i containing the cycle tag $t_c^{q_i}$ and $\vec{\Delta}_m = w(q_i)$. Here, $\vec{\Delta}_m$ represents the minimum demand of a rebalancing cycle c, i.e. the demand of the edge $q_{u,v}$ in a cycle c with the lowest $\vec{\Delta}_{u,v}$.

If participant u receives an additional REQUEST with no $t_r^{q_o} \in T_r$ it recognizes, it sends an UPDATE on every outgoing edge q_o which contains $T_r^{q_o} = T_r - T_s$. Only after the UPDATE will the participant add T_r to T_s as done for the first REQUEST. The UPDATE mechanism prevents deadlocks from occurring during the cycle detection part of the protocol.

Once a participant u receives a SUCCESS[5] on all $q_o \in Q_o^u$, it merges all $(t_c, \vec{\Delta}_m)$ pairs it received into two lists T_c^u and $T_c^{\neg u}$, representing the cycles that u owns and the cycles u does not own. Receiving all pairs for the cycles that u owns allows u to know the final $\vec{\Delta}_m$ for each owned cycle, which is used later in the COMMIT phase. It can happen that u receives multiple $(t_c, \vec{\Delta}_m)$ pairs with the same t_c on different q_o's, which means that there are multiple cycles owned by u which share the same incoming edge q_i. In that case, u must choose the $(t_c, \vec{\Delta}_m)$ pair with the largest $\vec{\Delta}_m$ and discard the others.

Then, for each $q_i \in Q_i^u$, u runs a subroutine called `splitEqually`$(w(q_i), T_c^{\neg u})$ which modifies the pairs in $T_c^{\neg u}$ such that $\sum_{(t_c, \vec{\Delta}_m) \in T_c^{\neg u}} \vec{\Delta}_m \leq w(q_i)$. This step is important to guarantee that all possible cycles going over q_i cannot exceed the $w(q_i)$ of the channel, as the channel owners do not want to rebalance *more* than agreed during the `negotiateObjective` execution. Once modified, $T_c^{\neg u}$ is sent in a SUCCESS message over q_i and u moves on to the COMMIT phase.

COMMIT phase

Once the leader has received and processed all SUCCESS messages like other participants, the leader also knows the final $\vec{\Delta}_m$ for each owned cycle. This allows the leader to set up a conditional transaction using $\texttt{pay}_{\text{cond}}$ that is guaranteed to succeed if the protocol has been followed honestly. This is because each edge that the owned cycles of the leader go through has reserved a part of $w(q)$ specifically for each of the owned cycles of the leader. To commit to each conditional cyclic transaction, the leader l generates a φ_{setup} for each t_c it owns and stores them. For each outgoing edge $q_o \in Q_o^l$, the leader then finds the cycles c that it owns and share the same q_o. The leader then creates a list $T_{c_{q_o}}^l$, containing $(t_c, \vec{\Delta}_m, \varphi_{\text{setup}})$ triples where all cycles share the same q_o. $T_{c_{q_o}}^l$ is constructed from T_c^l and the stored φ_{setup}.

Once a $T_{c_{q_o}}^l$ is constructed for each q_o, the leader sends a COMMIT message containg $T_{c_{q_o}}^l$ over each q_o and sets up a conditional transaction using $\texttt{pay}_{\text{cond}}$ on q_o for each $t_c \in T_{c_{q_o}}^l$. Once the leader received COMMITS on all $q_i \in Q_i^l$, it moves on to the EXEC phase.

A participant u that receives a COMMIT message does nothing until it has received COMMIT messages on all incoming edges $q_i \in Q_i^u$ that are not part of a cycle it owns, i.e. all $q_i \neq q_i^c$ where c are the cycles that u owns. Once all COMMIT messages are received, the participant acts similar to the leader, processing all its owned cycles to generate a $T_{c_{q_o}}^u$. However, for each received COMMIT message, it adds the received $T_{c_{q_o}}$ to $T_{c_{q_o}}^u$ such that $T_{c_{q_o}}^u \leftarrow T_{c_{q_o}} \cup T_{c_{q_o}}^u$. It then acts similar again to the leader, sending a COMMIT message on each q_o and sets up conditional transactions. Contrary to the leader however, u waits until it has received COMMITs on all incoming edges q_i^c for all cycles c that it owns before moving to the EXEC phase.

EXEC phase

A participant u that enters the EXEC phase immediately executes all conditional transactions on the cycles that they own using $\texttt{pay}_{\text{exec}}$. This is possible as the participant has either waited until it received all COMMITs from its own cycles or it is the leader (who always only receives COMMITs from its own cycles). Once a conditional transaction on an edge $q_i \in Q_i^u$ is executed, u sends an EXEC message over edge q_i which holds the φ_{exec} that belongs to the conditional transaction and the tag t_c that indicates the cycle for which the transaction is executed. If u receives an EXEC message, u can use φ_{exec} to execute the corresponding incoming transaction belonging to t_c. This process mimics the HTLC mechanism found in the Lightning Network. Once the leader has executed a transaction for all cycles that it owns, it immediately broadcasts (peer-to-peer) a

[5] A FAIL is also possible, excluding the edge q_o from any further interaction this round.

NEXT_ROUND message to each participant and continues to the next round. If a participant u receives a NEXT_ROUND message and has executed a transaction for all cycles that it owns or go through it, it also moves to the next round.

4.5 Participant discovery protocol

As stated in Subsection 4.4.1, the participant discovery protocol takes as its input a node u in G which has the intent of finding other nodes who wish to join in the execution of a distributed algorithm Λ_d with settings S. It outputs a set of participants P and a set of participating edges E_p. Each participant u obtains a copy of P and a local version of E_p^u containing only edges that belong to it. More formally, $P, E_p = \texttt{partDisc}(u, S, \Lambda_d)$ and $P \equiv \{A_u | u \in \text{participants of } \Lambda_d\}$, where A_u is an anonymous identity.

An anonymous identity is a randomly generated string generated by u. We call it 'anonymous' as u does not reveal to anyone the relation between its real identity and A_u. In every execution of the protocol, each node generates a fresh A_u for usage during the protocol. We argue in Section 4.8 why using anonymous identities helps in providing participation anonymity (Definition 3).

Table 4.1: Definition of messages as used in the participant discovery protocol

Type	Format	Contents
$*$ (all messages)	$*(\nu)$	ν is the execution id of the algorithm
INVITE	$INV(\nu, h_c, I_{\max}, S)$	h_c is an integer representing the hop count, i.e. how far the INVITE may travel. $I_{\max}$ is an integer representing the maximum number of invites a node u may send. S represents a generic datastructure such as a JSON that holds the proposed settings for the execution of the distributed algorithm Λ_d
ACCEPT	$ACC(\nu, \{A_u, A_v, \ldots\}, isChild)$	$\{A_u, A_v, \ldots\}$ is a list of participants of the distributed algorithm Λ_d and *isChild* is a boolean that indicates if the receiver should adopt the sender as its child in the tree T
DENY	$DENY(\nu)$	–
FINISH	$FIN(\nu, \{A_u, A_v, \ldots\})$	Similar contents to ACCEPT but without *isChild*

The messages used in the discovery protocol are defined in Table 4.1. To allow for concurrent execution in the same graph G, all messages contain an execution id ν to allow multiple instances of the discovery protocol to determine which execution a message belongs to. If a node receives a message m from a different execution than its own, it replies with a DENY message to the sender.

4.5.1 Parameters

The results and performance of the discovery protocol can be tuned using two parameters, $I_{\max}$ and h_c. The parameters influence the quantity and distance of travel of an INVITE, respectively. This allows the initiator to put an upper bound on the size of P. The size of P is important as, in the case for which Λ_d is the transaction generation protocol, a larger size means that there is a bigger chance of rebalancing cycles being present. However, a large size of P might also mean that cycles are longer, which makes the transaction generation algorithm run slower. For a general Λ_d, a larger P also causes more opportunities for failures to occur and longer runtimes.

We argue that a straightforward mechanism – such as allowing only a maximum number of participants – to limit the size of P is not practical as this would require the initiator to choose a subset from the nodes that are willing to participate. As will become clear later, in our current protocol the initiator has no simple way of connecting the anonymous identities A_u of participants to their real identities. This in turn prevents the initiator from knowing the topology of the subgraph G_p that is created by combining P and E_p. If the initiator randomly removes a potential participant from P, there is a risk that G_p contains more than one component if the removed participant has an edge that acted as a bridge between two components. As an alternative that does not require participant removal, we, therefore, choose to define parameters h_c and $I_{\max}$. These allow the initiator to set an upper bound on the size of P.

From these parameters, h_c is an integer representing the hop count. For every node an INVITE message passes through, the h_c is decreased by one. Once the h_c reaches zero, a node is prohibited from forwarding the INVITE message. This is inspired by the Time-To-Live (TTL) mechanism as found in the IP packet header [16]. In the IP standard, the TTL is used as a means to prevent packets from endlessly being forwarded in accidental routing cycles. In the participant discovery protocol, we use it to limit the spread of an INVITE to a maximum number of hops away from the initiator. The parameter $I_{\max}$ is an integer representing the maximum number of invites a node may send. We can then use Equation 4.3 to calculate the upper bound on the size of P as this equals the number of INVITEs send when they are all accepted and received by unique nodes. This results in an upper bound on P of $\mathcal{O}({I_{\max}}^{h_c})$.

$$\sum_{i=1}^{h_c} {I_{\max}}^{i} = \frac{I_{\max} - {I_{\max}}^{h_c+1}}{1 - I_{\max}} = \mathcal{O}({I_{\max}}^{h_c}) \tag{4.3}$$

4.5.2 Functions

The participant discovery protocol consists of a `start`, a `handleResponses` function and four message handlers for INVITE, ACCEPT, DENY and FINISH messages. This section discusses how the protocol operates and our considerations during its detailed design. A compact view of the pseudocode of the algorithm is provided in Appendix A. A visualization of an execution of the algorithm is provided in Figure 4.2.

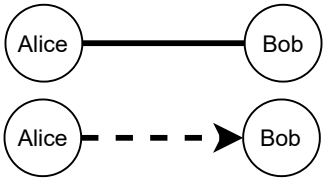
Algorithm 1 Pseudocode for the participant discovery protocol for a node u (`start`)

```
1:  awake, started, processedResponses, invitesSend ← false
2:  ν, A_u, e_p, S_d, alg ← null
3:  P, E_c, E_a, I_m ← ∅
4:
5:  procedure START(h_c, I_max, S, Λ_d)
6:      awake ← true
7:      started ← true
8:      invitesSend ← true
9:      ν ← randomly generated number
10:     A_u ← randomly generated number
11:     P ← P ∪ A
12:     for all e ∈ E_u do
13:         send (invite;ν,h_c, I_max, S, Λ_d) on edge e
```

Initialisation and start

Algorithm 1 specifies the initialization of the protocol's variables and the `start` function. The `start` function is invoked by the protocol's initiator. The initiator generates an execution id ν and its own anonymous identity A_u, after which it sends an INVITE on all its edges E_u. The invocation of this function corresponds to step 4.2b in Figure 4.2.

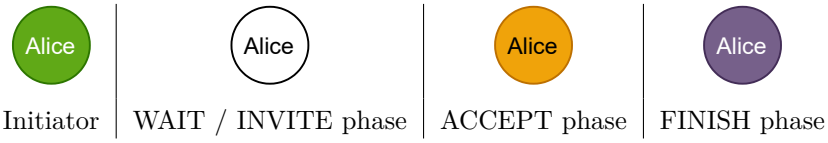
Table 4.2: Definition of edges for Figure 4.2



Alice and Bob are connected over an undirected edge, i.e. $e_{\text{Alice,Bob}}$

Alice sends a message to Bob over an edge $e_{\text{Alice,Bob}}$

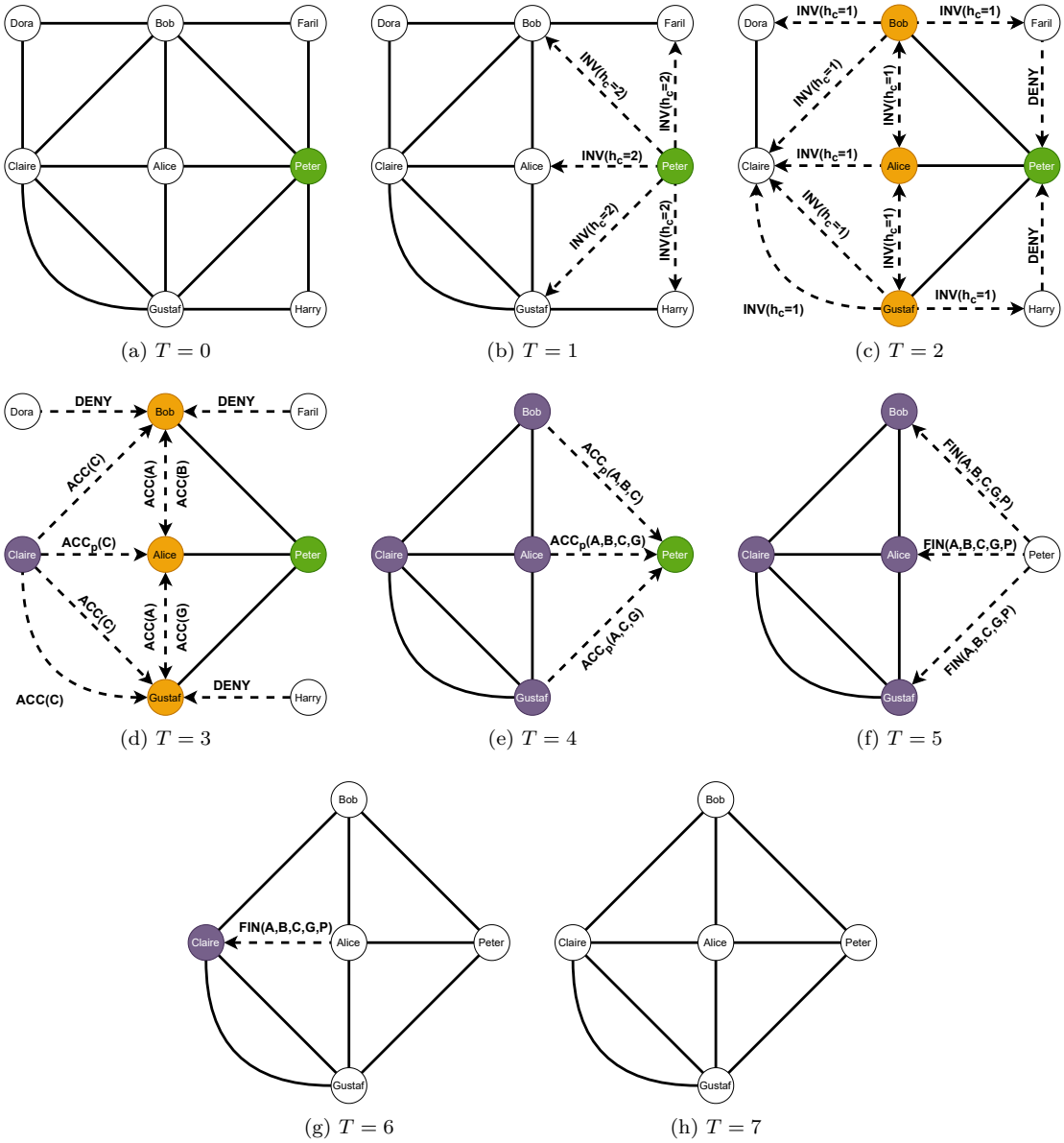
Table 4.3: Definition of colours for Figure 4.2



Initiator | WAIT / INVITE phase | ACCEPT phase | FINISH phase

Table 4.4: Definition of messages for Figure 4.2. Note that these messages are based on their formal definition as provided in Table 4.1 although for conciseness, ν, $I_{\max}$ and formal notation are ignored

Message	Type	Contents
$INV(h_c = x)$	INVITE	Hop count h_c indicating that the INVITE message may only visit x hops
$ACC(A, B, \ldots)$	ACCEPT	List of random strings representing participants such as A and B
$ACC_p(A, B, \ldots)$	ACCEPT	Same as ACC but also indicating that the receiver should adopt the sender as its child in the tree T
$DENY$	DENY	–
$FIN(A, B, \ldots)$	FINISH	Same as ACC



(a) $T = 0$ (b) $T = 1$ (c) $T = 2$

(d) $T = 3$ (e) $T = 4$ (f) $T = 5$

(g) $T = 6$ (h) $T = 7$

Figure 4.2: An invocation of the participant discovery protocol by Peter on an example PCN G_e. Table 4.2, Table 4.3 and Table 4.4 define the edges, colours and messages found in this example

Algorithm 2 Pseudocode for the participant discovery protocol for a node u (`invite`)

14: **upon receipt of** (invite;$id, h_c, I_{\max}, S, \Lambda_d$) **on edge** j **do**
15: **if** not willing to participate with j **then**
16: **send** (deny;id) **on edge** j and **return**
17: **if** $\nu \neq$ null $\wedge\, \nu \neq id$ **then** ▷ Ignore other executions of this algorithm
18: **send** (deny;id) **on edge** j and **return**
19: **if** $\nu =$ null **then** ▷ Node gets claimed by the starting node
20: $awake \leftarrow$ true
21: $\nu \leftarrow id$
22: $S_d \leftarrow S$
23: $alg \leftarrow \Lambda_d$
24: $A \leftarrow$ randomly generated number
25: $P \leftarrow P \cup A$
26: $e_p \leftarrow j$
27:
28: **if** $\neg invitesSend \wedge h_c - 1 > 0 \wedge |E_u| > 1$ **then**
29: $invitesSend \leftarrow$ true
30: $e_p \leftarrow j$
31:
32: **for all** $e \in E_u : e \neq e_p \wedge e$ has not been denied **do**
33: **send** (invite;$\nu, h_c - 1, I_{\max}, S, \Lambda_d$) **on edge** e
34: $I_m \leftarrow I_m \cup e$
35:
36: **if** $|I_m| > I_{\max}$ **then**
37: **break**
38:
39: **return**
40:
41: $E_a \leftarrow E_a \cup j$
42: **send** (accept;$\nu, P, e_p = j$) **on edge** j and **return**
43:

INVITE message handler

Algorithm 2 specifies how a node u should handle an INVITE message. It can be seen in action in step 4.2c and 4.2d in Figure 4.2.

Lines 14-18 allow u to deny participating for any reason and protect it from partaking in any other execution if it already has a ν. If u has not yet been claimed by an execution, i.e. if its ν is `null`, it enters the protected block at Line 19. The process of being claimed for an execution involves u assuming the received *id* as its own ν, storing the received settings S and algorithm Λ_d and also generating its own A_u. u also stores the edge on which the message has been sent (j) in the variable e_p to know where to send its ACCEPT message during the ACCEPT phase.

u is only allowed to forward INVITEs if: i) it has not already done so before, ii) the hop count is larger than zero and iii) it has more than one edge. If u *is* allowed to forward INVITEs, u enters Line 28. In case this is not the first INVITE u receives but it is the first to meet the necessary conditions for forwarding, u has to update its parent edge e_p. Updating the parent edge makes sure that u has not already sent an ACCEPT message as a reply to the INVITE, which would prevent the propagation of the potential participants u receives as a reply to *its own* invites.

u then sends an INVITE on all its edges E_u which are not its parent edge e_p and are not already 'denied' by u itself. The first condition is important as the original INVITE came through e_p and the next expected message sent over e_p is an ACCEPT that will be sent later in the `handleResponses` handler. The second condition ensures that if u already decided not to participate with e, u is not allowed to send an INVITE over e. After the sending of each invite, u keeps track of whom it invited in I_m and makes sure it does not go over $I_{\max}$.

Finally, if none of the other protected blocks are entered, u reaches Line 41. This happens if u already forwarded INVITEs or if it gets a valid INVITE with a hop count of zero or if it has only one edge. In all those cases, u can immediately ACCEPT and add j to its list of accepted edges E_a. The boolean *isChild* in the ACCEPT message (Table 4.1) is conditionally true or false depending on the predicate $e_p = j$. This ensures that if u is replying with ACCEPT on e_p, it always lets its (then) parent know that it should be adopted as its child. It can happen that if u first receives an INVITE with $h_c \leq 1$ and then an INVITE from another node with $h_c > 1$, u eventually sends[6] an ACCEPT to both nodes with *isChild* = true. This later causes both receivers of the ACCEPT message to think u is its child meaning that u, as we will see later on, will receive the FINISH message twice. This does not pose a problem as u will terminate the discovery protocol before processing the second FINISH message.

ACCEPT and DENY message handler

Algorithm 3 specifies how a node u should handle an ACCEPT and a DENY message. The `deny` handler can be seen in action in step 4.2c and 4.2d in Figure 4.2 while the `accept` handler is only active in step 4.2d and 4.2e.

Similar to the `invite` message handler, both ACCEPT and DENY have protections against their incorrect invocation in Lines 45 and 59. A small addition to the guards prevents u from processing any ACCEPT or DENY from edges it has not sent an INVITE to.

In the `accept` handler and after passing the guards, u reaches Lines 49-51. In these lines, u updates its own local version of P with the one received in the ACCEPT message. u also adds j to its set of accepted edges E_a and removes j from the edges it invited I_m, given it now has a proper response to its INVITE. u then checks if the ACCEPT message has the *isChild* boolean set and if so, adds j to its set of children E_c. Once all done, u invokes `handleResponses` to check if it has received enough ACCEPTs and DENYs to continue to the FINISH phase.

The `deny` handler does less than the `accept` handler, only removing j from the edges it invited I_m and also invokes `handleResponses` for the same reason as the `accept` handler.

Subroutine handleResponses

Algorithm 4 specifies the subroutine `handleResponses` that is called in both the `accept` and `deny` message handlers. It can be seen in action in step 4.2e in Figure 4.2.

[6] Once in `invite` and once in `handleResponses`

Algorithm 3 Pseudocode for the participant discovery protocol for a node u (`accept` and `deny`)

```
44: upon receipt of (accept;id, R, isChild) on edge j do
45:     if ¬awake then return
46:     else if ν ≠ id ∨ j ∉ I_m then
47:         send (deny;id) on edge j and return
48:
49:     P ← P ∪ R
50:     E_a ← E_a ∪ j
51:     I_m ← I_m − j
52:
53:     if isChild then
54:         E_c ← E_c ∪ j
55:
56:     handleResponses()
57:
58: upon receipt of (deny;id) on edge j do
59:     if ¬awake ∨ j ∉ I_m then return
60:     I_m ← I_m − j
61:     handleResponses()
62:
```

Algorithm 4 Pseudocode for the participant discovery protocol for a node u (`handleResponses`)

```
63: procedure HANDLERESPONSES
64:     if |I_m| = 0 ∧ ¬processedResponses then
65:         processedResponses ← true
66:
67:         if started then
68:             if E_a ≠ ∅ then
69:                 for all e ∈ E_c do
70:                     send (finish;ν, P) on edge e
71:                 execute alg(ν, A_u, P, E_a, S_d)
72:             terminate
73:         else
74:             E_a ← E_a ∪ e_p
75:             send (accept;ν, P, true) on edge e_p
```

The body of `handleResponses` is only executed *once* if u received a response to all its INVITEs, in which case I_m is zero. Once u entered the protected block at Line 64, it is presented with two paths which it enters depending on u being the initiator (Line 67).

In case u is not the initiator (Line 73), it sends an ACCEPT containing its local P and $isChild = true$ on e_p and moves to the FINISH phase. This step is important because the closer each node is to the initiator, the more complete their copy of P is because of the merging that happens in the `accept` message handler. This is similar to the divide-and-conquer design paradigm where sub-problems become closer in similarity to the original problem as the amount of divides decreases. The sending of the ACCEPT message containing u's local copy of P to its parent allows its parent to obtain a more complete copy of P than u has. This repeats all the way to the initiator who obtains a complete copy of P. This is visualized in step 4.2e in Figure 4.2 where Alice has knowledge of the whole set of participants $\{A, B, C, G\}$ while Claire only knows $\{C\}$. The next step of the protocol, concerning the propagation of the FINISH message, is designed to spread the complete copy of P through G such that all participants know the complete copy.

The other path is invoked if u is the initiator and has accepted edges E_a. u then sends its own complete copy of P in a FINISH message to all its children in E_c and executes Λ_d with ν, A_u, P, S and E_a. Once the execution of Λ_d is complete, u terminates. Only sending messages to

$e \in E_c$ allows the FINISH messages to follow a tree-like path through G which results in a lower message complexity than simply broadcasting the FINISH messages. This has the drawback that a FINISH message might take longer to arrive than compared to when all nodes broadcast the FINISH message to all $e \in E_a$. The choice for this mechanism is therefore one between message and time complexity.

FINISH message handler

Algorithm 5 Pseudocode for the participant discovery protocol for a node u (`finish`)

```
76: upon receipt of (finish;id, R) on edge j do
77:     if ¬awake then return
78:     else if ν ≠ id then
79:         send (deny;id) on edge j and return
80:
81:     P ← R
82:     for all e ∈ E_c do                          ▷ Propagate participant list along tree
83:         send (finish;ν, P) on edge e
84:     execute alg(ν, A_u, P, E_a, S_d)
85:     terminate
86:
```

Algorithm 5 specifies how a node u should handle a FINISH message. The `finish` handler can be seen in action in step 4.2f and 4.2g in Figure 4.2.

The `finish` message handler has the same guards as the `accept` handler to protect against its incorrect execution. It is only executed if a participant u who is not the initiator receives a FINISH message. Once the message is received, u overwrites its own copy of P with the received set of participants R (Line 81). It then acts similar to the initiator in `handleResponses` and sends a FINISH message containing P to all $e \in E_c$ after which it executes Λ_d with ν, A_u, P, S and E_a. Once Λ_d has finished executing, u terminates.

4.6 Transaction generation protocol

In the final part of the participant discovery protocol, the protocol executes its given distributed algorithm Λ_d with settings S and inputs ν, A_u, P and E_p. The functionality of this protocol is described in Section 4.5. In this section, we set Λ_d to be equivalent to the transaction generation protocol, such that after the execution of the participant discovery protocol the transaction generation protocol is executed.

As stated in Subsection 4.4.2, the transaction generation protocol takes as main inputs the set of participants P and participating edges E_p. Other important inputs it gets from the discovery protocol are the execution id ν, its settings S and the anonymous identity of a node A_u. Both E_p and A_u are unique to each participant and are a result of the local computation by u of the discovery protocol.

The protocol is a peer-to-peer protocol (similar to the participant discovery protocol) and is round based. It does not output anything, in contrast with what one might think reading its name. Instead, each round the protocol finds and executes rebalancing cycles C given a rebalancing graph R that is created from each $\vec{\Delta}_{u,v}$ of the participants. The largest challenge in the generation of the cycles for the generation protocol is that for all cycles $C_{q_{u,v}}$ that include a directed rebalancing edge $q_{u,v}$, $\sum_{(t_c,\vec{\Delta}_m)\in C_{q_{u,v}}} \vec{\Delta}_m \leq w(q_{u,v})$. Less formally, we require that the sum of cycles that include a rebalancing edge $q_{u,v}$ may not exceed the rebalancing objective $w(q_{u,v})$ of that edge.

4.6.1 Protocol specific definitions

The messages that are communicated during the execution of the protocol are defined in Table 4.5.

Table 4.5: Definition of messages as used in the transaction generation protocol

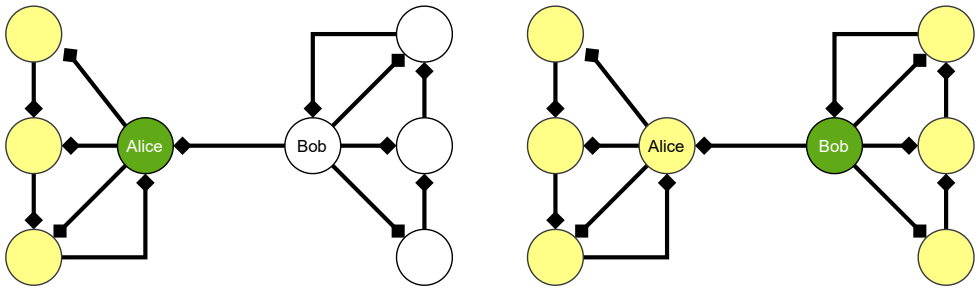
Type	Format	Contents
$*$ (all messages)	$*(\nu, A_l)$	ν is the execution id and A_l is the anonymous identity of the round leader as generated during the participant discovery
REQUEST	$R(\nu, A_l, \{t_r^{u_1}, t_r^{v_2}, \ldots\})$	List of t_r, e.g. $t_r^{u_1}$ is the first t_r the participant u generated and $t_r^{v_2}$ is the second t_r the participant v generated
UPDATE	$U(\nu, A_l, \{t_r^{u_1}, t_r^{v_2}, \ldots\})$	Same contents as REQUEST
SUCCESS	$S(\nu, A_l, \{(t_c, \vec{\Delta}_m)^{u_1}, (t_c, \vec{\Delta}_m)^{v_2}, \ldots\})$	List of $(t_c, \vec{\Delta}_m)$ pairs, e.g. $(t_c, \vec{\Delta}_m)^{u_1}$ is a pair representing the first cycle owned by participant u that has tag $t_c^{u_1}$ and minimum demand $\vec{\Delta}_m^{u_1}$
FAIL	$F(\nu, A_l, w_{\text{failure}})$	w_{failure} is an integer uniquely representing a reason for the failure
COMMIT	$C(\nu, A_l, \{(t_c, \vec{\Delta}_{\text{EX}}, \varphi_{\text{setup}})^{u_1}, (t_c, \vec{\Delta}_{\text{EX}}, \varphi_{\text{setup}})^{v_2}, \ldots\})$	List of $(t_c, \vec{\Delta}_{\text{EX}}, \varphi_{\text{setup}})$ triples, e.g. $(t_c, \vec{\Delta}_{\text{EX}}, \varphi_{\text{setup}})^{u_1}$ is a triple representing the first cycle owned by participant u that has tag $t_c^{u_1}$, a demand to be executed $\vec{\Delta}_{\text{EX}}^{u_1}$ and conditional transaction setup information $\varphi_{\text{setup}}^{u_1}$
EXEC	$E(\nu, A_l, t_c, \varphi_{\text{exec}})$	Cycle tag t_c and conditional transaction execution information φ_{exec}. t_c and φ_{exec} must correspond to a $(t_c, \vec{\Delta}_{\text{EX}}, \varphi_{\text{setup}})$ triple as seen in a COMMIT message
NEXT_ROUND	$NR(\nu, A_l)$	–

In the transaction generation protocol, we switch between representing a rebalancing cycle as c or as $(t_c, \vec{\Delta}_m)$, where t_c is a *cycle tag* identifying the cycle and $\vec{\Delta}_m$ represents the minimum demand of c, i.e. the demand of the edge $q_{u,v}$ in a cycle c with the lowest $\vec{\Delta}_{u,v}$.

4.6.2 Parameters

As stated in the functional discussion of the protocol and its evaluation, the cycles found in each round are not necessarily the optimal solution to the rebalancing problem when compared to Revive. Cycle discovery is dependent on the direction of an edge q in R and the order in which messages arrive during a round. An example is given in Figure 4.3 where, if there are two components with one directed bridge between them, the location of the leader can exclude or include a whole component during a round. We, therefore, designed our protocol to carry out multiple rounds of rebalancing in order to approach an optimal solution.

The number of rounds the protocol runs for each invocation is determined by *maxRounds* which is stored in S. *maxRounds* can be set directly or can be defined as $maxRounds = |P| \cdot \rho$, where ρ represents the percentage of P that is allowed to become a leader. The reasoning behind the second option is that we expect that the number of rounds required for a small $|P|$ to approach the optimal solution increases linearly with $|P|$. We argue that it, therefore, makes sense to always scale the number of rounds to the number of participants. We will evaluate both options in Chapter 5.



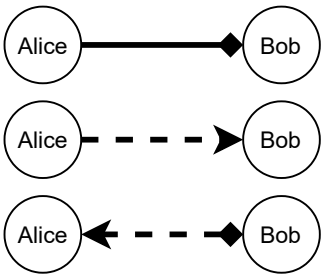
(a) (b)

Figure 4.3: Example of a rebalancing graph with two components connected with a bridge between Alice and Bob. In a), if Alice is the leader, only her component will take part in the rebalancing because a REQUEST is only send on an outgoing edge. In b), if Bob is the leader, both components will take part in the rebalancing as Bob has an outgoing edge to Alice. To make sure every node gets the chance to rebalance, the protocol runs multiple rounds with different leaders.

4.6.3 Functions

The transaction generation protocol is more complex than the participant discovery protocol and consists of `wakeUp`, `startRound`, `nextRound`, `checkForCycles`, `checkForCyclesAndNewTags`, `replyToRequests`, `commitLeader`, `commit`, `splitEqually`, `checkIfExecutionSafe` functions and message handlers for the REQUEST, UPDATE, SUCCESS, FAIL, COMMIT, EXEC and NEXT_ROUND messages. This section will discuss how the protocol operates and the considerations during its detailed design. A compact view of the pseudocode of the algorithm is provided in Appendix B. A visualization of an execution of the algorithm is provided in Figure 4.4.

Table 4.6: Definition of edges for Figure 4.4



Alice has a directed edge to Bob, i.e. $q_{\text{Alice,Bob}}$

Alice sends a message to Bob in the direction of the edge $q_{\text{Alice,Bob}}$

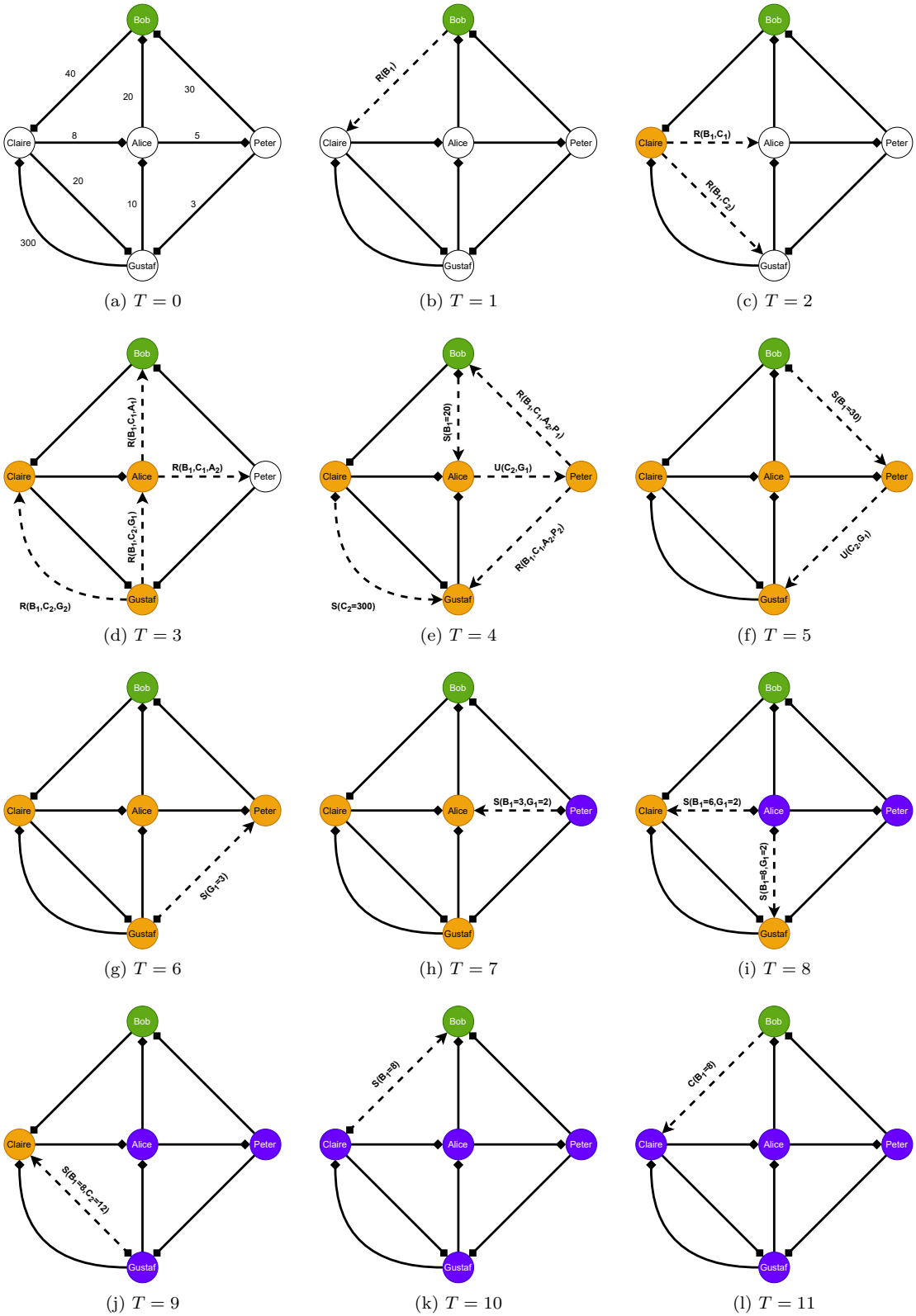
Bob sends a message to Alice in the opposite direction of the edge $q_{\text{Alice,Bob}}$

Table 4.7: Definition of colours for Figure 4.4

Leader | WAIT / REQUEST phase | SUCCESS phase | COMMIT phase | EXEC phase

Table 4.8: Definition of messages for Figure 4.4. Note that these messages are based on their formal definition as provided in Table 4.5 although for conciseness, ν, A_l and formal notation are ignored

Message	Type	Contents
$R(B_1, A_2, \ldots)$	REQUEST	List of t_r, e.g. B_1 is the first t_r Bob generated and A_2 is the second t_r Alice generated
$U(B_1, A_2, \ldots)$	UPDATE	Same contents as request
$S(B_1 = 3, \ldots)$	SUCCESS	List of $(t_c, \vec{\Delta}_m)$ pairs, e.g. $B_1 = 3$ is a $(t_c, \vec{\Delta}_m)$ pair with $t_c = B_1$ being the first cycle owned by participant B and $\vec{\Delta}_m = 2$ stating that the minimum demand of this cycle is 2
$C(B_1 = 3, \ldots)$	COMMIT	List of $(t_c, \vec{\Delta}_{\text{EX}})$ pairs. $B_1 = 3$ is a $(t_c, \vec{\Delta}_{\text{EX}})$ pair with $t_c = B_1$ being the first cycle owned by participant B and $\vec{\Delta}_{\text{EX}} = 2$ stating that the demand to be executed on this cycle is 2. We omit the conditional transaction setup information φ_{setup} from the formal definition of COMMIT for brevity.
$E(B_1 = 3)$	EXEC	Formally only contains a cycle tag t_c and conditional transaction execution information φ_{exec}. However, for brevity we include the $\vec{\Delta}_{\text{EX}}$ of the cycle the cycle tag t_c belongs to and ignore φ_{exec}.



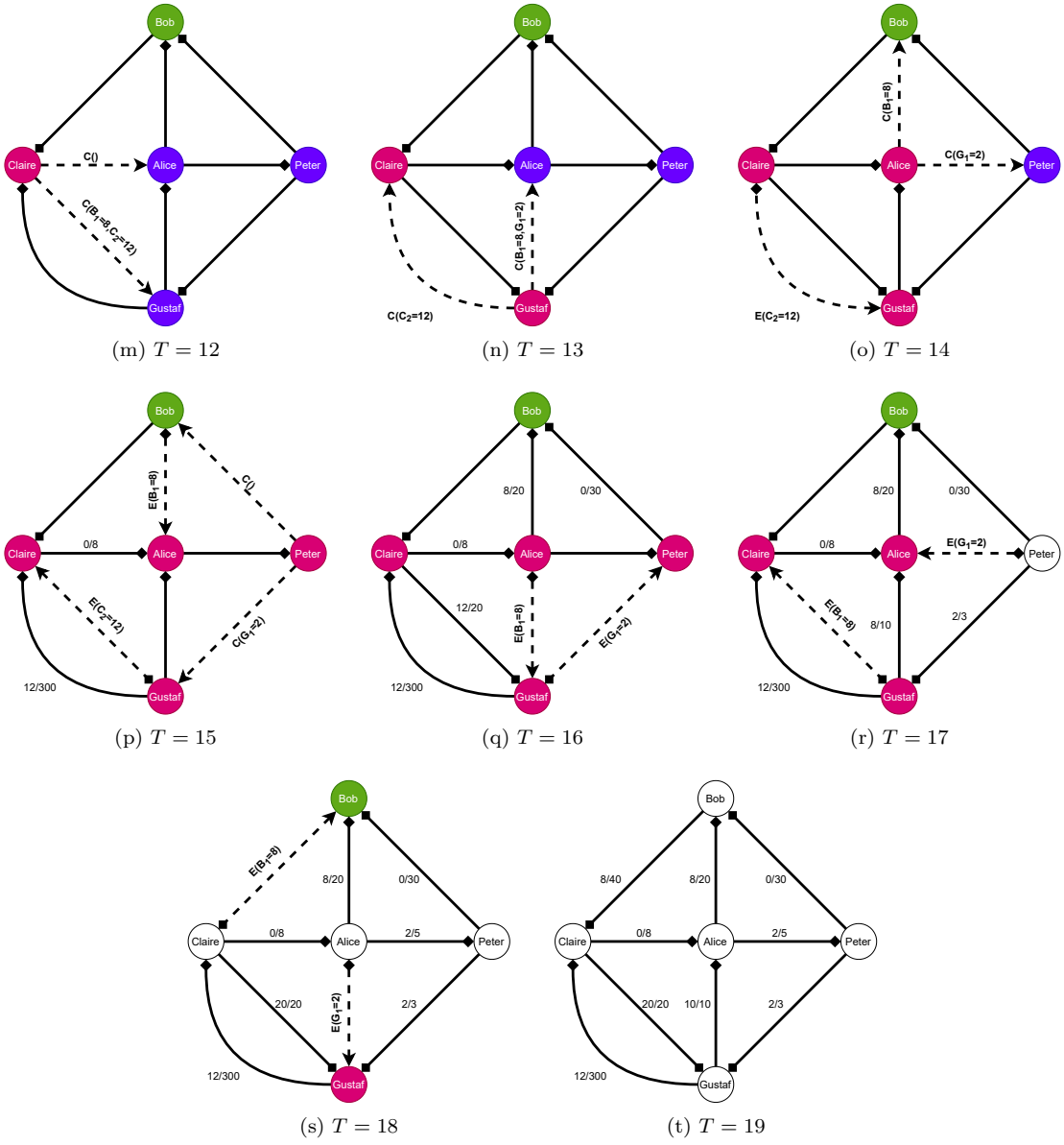
(a) $T = 0$ (b) $T = 1$ (c) $T = 2$

(d) $T = 3$ (e) $T = 4$ (f) $T = 5$

(g) $T = 6$ (h) $T = 7$ (i) $T = 8$

(j) $T = 9$ (k) $T = 10$ (l) $T = 11$

Figure 4.4: An example of one round of our transaction generation protocol on a rebalancing graph R_e with Bob as the leader (part 1/2)



(m) $T = 12$ (n) $T = 13$ (o) $T = 14$

(p) $T = 15$ (q) $T = 16$ (r) $T = 17$

(s) $T = 18$ (t) $T = 19$

Figure 4.4: An example of one round of our transaction generation protocol on a rebalancing graph R_e with Bob as the leader (part 2/2)

Initialization

Algorithm 6 Pseudocode for the implementation of the transaction generation protocol for a node u (`wakeUp`)

```
1:  procedure WAKEUP
2:      if awake then return
3:
4:      awake ← true
5:      L_P ← sort P using fixed sorting algorithm
6:      A_l ← L_P[i_r]
7:
8:      for all e ∈ E_a do
9:          Lock edge e
10:         q ← negotiateObjective(e)
11:         if w(q) < 0 then
12:             Q_out ← Q_out ∪ {q}
13:         else
14:             Q_in ← Q_in ∪ {q}
15:
16:     if A_l = A_u then
17:         startRound()
```

Algorithm 6 specifies what happens during the initial execution of the protocol after it is invoked by the discovery protocol. The `wakeUp` procedure showcases how we select the leader for each round and how we obtain rebalancing edges $q \in Q$.

In Line 5, the set of participants P is sorted by the node u to obtain a sorted list of participants L_P. Any sorting algorithm can be used here as long as all nodes use the same sorting algorithm and the algorithm has the *stable* property, i.e. sorting the same set k times always produces the same ordered list. Sorting P allows each node to derive the current leader from the current round, as once L_P is known to every node, each leader corresponds to an index in L_P (Line 6).

The next step is for the node to carry out `negotiateObjective` on its accepted edges E_a. We lock the edge before this step (Line 9) which prevents any normal[7] transactions from taking place on edge e. This is required as the outcome of `negotiateObjective` may vary depending on the current balance of the edge e. If the adversary decides to forego locking an edge, there is a chance that during a later stage of the protocol $\texttt{pay}_{\text{cond}}$ will fail due to a lack of coins in the channel to support the transaction. The edges E_a are only unlocked at the end of the last round of the protocol during the final invocation of `nextRound`. Invoking `negotiateObjective` has the result of producing two sets of rebalancing edges Q_{out} and Q_{in} for use during later procedures of the protocol.

Finally, in Line 16, the node checks if it is the leader of the first round in which case it starts the first round.

Starting a round

Algorithm 7 specifies what happens during the start of each round. `startRound` is only invoked by the leader of the round. It can be seen in action in step 4.4b in Figure 4.4.

Starting a round as the leader involves sending a REQUEST message on each $q \in Q_{\text{out}}$ that contains a cycle detection tag t_r unique to each edge q (Line 25). Because a t_r is uniquely associated with an edge q, if it is received again on an edge q_e, the node knows which edge q it initially send it on using $D_{t_r,q}$. This makes it possible to define a starting and ending edge of a cycle to the node u.

The `startRound` procedure contains one exception in Line 19 that skips sending the REQUEST messages. This case is required if the leader node has no outgoing edges, in which case it cannot

[7] i.e. every transaction not invoked by this protocol

Algorithm 7 Pseudocode for the implementation of the transaction generation protocol for a node u (`startRound`)

18: **procedure** STARTROUND
19: **if** $|Q_{\text{out}}| = 0$ **then**
20: `nextRound`()
21: **return**
22:
23: *roundState* $\leftarrow$ REQ
24: **for** $q \in Q_{\text{out}}$ **do**
25: $t_r \leftarrow$ randomly generated identifier
26: $D_{t_r,q}$.`put`(t_r, q)
27: $n_{\text{sendRequests}} \leftarrow n_{\text{sendRequests}} + 1$
28:
29: **send** (request;$\nu, A_l, \{t_r\}$) **on edge** q

send a REQUEST message. The simple solution for this problem is to trigger the leader to continue to the next round in which a new leader is elected that hopefully does have outgoing edges.

Cycle detection

Algorithm 8 Pseudocode for the implementation of the transaction generation protocol for a node u (`checkForCycles`)

30: **procedure** CHECKFORCYCLES(m = (request;T_r) **on edge** j)
31: $I \leftarrow \{t_r | (t_r, q) \in D_{t_r,q}\} \cap T_r$
32: **if** $|I| > 0$ **then**
33: $t_r^{\text{mtch}} \leftarrow$ randomly picked t_r out of I
34: $L_{\text{rREQUEST}} \leftarrow L_{\text{rREQUEST}} - m$
35: $t_c \leftarrow$ randomly generated identifier
36: $\vec{\Delta}_m \leftarrow w(j)$
37: $D^u_{t_c}$.`put` $\left(t_c, \left[j, D_{t_r,q}.\texttt{get}(t_r^{\text{mtch}}), \vec{\Delta}_m, \text{false}\right]\right)$
38:
39: **send** (success;$\nu, A_l, \{(t_c, \vec{\Delta}_m)\}$) **on edge** j
40: **return** true
41:
42: **return** false

Once a REQUEST is received by a node, it gets processed in a similar manner to how the leader started a round in Algorithm 7. We therefore do not discuss the REQUEST message handler in this chapter but provide it in Algorithm 19 in Appendix B. It can be seen in action in step 4.4c, 4.4d and 4.4e in Figure 4.4. However, we do wish to discuss the `checkForCycles` procedure used by the REQUEST and UPDATE message handler as it shows the workings of the cycle detection done in the protocol. We provide its pseudocode in Algorithm 8.

`checkForCycles` must be invoked with a REQUEST message containing a set T_r which contains one or more cycle detection tags t_r. It then calculates the intersection between T_r and the node's own tags stored in $D_{t_r,q}$. If the intersection contains at least one tag, a cycle has been detected to and from this node. If a cycle is detected, the node enters the block at Line 32 and generates a cycle tag t_c which identifies the detected cycle (Line 35). The node u then stores information about the cycle in its map of owned cycles $D^u_{t_c}$. This information includes:

- $q_e^{u,c}$: The edge where the cycle ends, defined as the edge q on which the REQUEST or UPDATE message arrived (j) containing the detected t_r.

- $q_s^{u,c}$: The edge where the cycle starts, initially defined as the edge q that is associated with the t_r using $D_{t_r,q}$. Can change later during `replyToRequests`.
- $\vec{\Delta}_m$: The minimum demand of the cycle, initially defined as $w(q_e^{u,c})$. Unless $q_e^{u,c}$ has the lowest weight of the cycle and there are no overlapping cycles, $\vec{\Delta}_m$ is lowered later on during `replyToRequests`.
- *completed*: A boolean indicating if the cycle is complete, initially set to false. We define a cycle as complete if the node who owns the cycle (i.e. has the t_c in $D_{t_c}^u$) has received a SUCCESS message containing t_c. *completed* is modified in `replyToRequests` and checked during `commit`.

Once these actions are completed, the node sends a SUCCESS message on $q_e^{u,c} = j$ to inform it about the cycle. This SUCCESS message only contains the cycle tag t_c and its corresponding $\vec{\Delta}_m$.

An example where `checkForCycles` detects a cycle is in step 4.4d and 4.4e in Figure 4.4. In step 4.4d, Alice sends a REQUEST to Bob containing the cycle detection tag B_1, among others. As B_1 is in the $D_{t_r,q}$ of Bob, Bob detects a cycle and replies with a SUCCESS to Alice in step 4.4e. This SUCCESS contains the cycle tag B_1 and the rebalancing objective $w(q_e^{u,c})$ which is 20 in this example.

Deadlock resolution

Algorithm 9 Pseudocode for the implementation of the transaction generation protocol for a node u (handle UPDATE)

```
43: upon receipt of m = (update;id, A_i, T_r) on edge j do
44:     check-action: Pr_ν(id) - disallow, ¬awake - disallow
45:     check-action: Pr_{r,future}(A_i) - defer, Pr_{r,early}(A_i) - disallow
46:
47:     if roundState ≠ REQ ∨ already send SUCCESS on j this round then
48:         return
49:
50:     if ¬checkForCyclesAndNewTags(m) then return
51:
52: procedure checkForCyclesAndNewTags(m = (request;T_r) on edge j)
53:     if checkForCycles(m) then return false
54:
55:     T_r^new ← T_r − T_r^recv
56:     if |T_r^new| > 0 then
57:         T_r^recv ← T_r^recv ∪ T_r
58:
59:         for all q ∈ Q_out that did not reply with SUCCESS or FAIL this round do
60:             send (update;ν, A_l, T_r^new) on edge q
61:
62:     return true
```

Algorithm 9 specifies the actions a node u takes on the receipt of an UPDATE message and specifies `checkForCyclesAndNewTags`. Besides being invoked in the UPDATE message handler, `checkForCyclesAndNewTags` is also called during the invocation of the REQUEST message handler. It can be seen in action in step 4.4e and 4.4f in Figure 4.4.

UPDATE messages are used in the protocol to resolve deadlocks. Such a deadlock can be observed in Figure 4.4 where in step 4.4e, Gustaf receives a REQUEST message containing no tag it recognizes. This is because in step 4.4e, when Gustaf sent a REQUEST to Alice containing – among others – its own cycle detection tag G_1, Alice already send a REQUEST to Peter containing different tags which were previously received from Claire. Peter then directly forwards the tags

from Alice to Gustaf which results in Gustaf receiving a REQUEST containing no detection tags it recognizes. This causes a cycle in our wait-for graph consisting of Gustaf → Alice → Peter → Gustaf as they are all waiting to receive a SUCCESS or FAIL on their REQUEST. Such a reply is only given when: i) a node has received a SUCCESS or FAIL on their request (Algorithm 10) or ii) a cycle is detected (Algorithm 8). As i) is a circular definition, a mechanism is needed to trigger ii). We achieve this by making a node propagate tags that are received after sending a REQUEST. This is done using an UPDATE message. In Figure 4.4 in step 4.4e, Alice sends an UPDATE message to Peter to follow up on the REQUEST message she has already send Peter. Alice was triggered to send an UPDATE message as she received a REQUEST from Gustaf with new tags after already sending a REQUEST to Peter. Peter forwards the UPDATE message to Gustaf in step 4.4f. Once Gustaf receives the UPDATE message containing G_1, it can resolve the deadlock because it detected a cycle, fulfilling condition ii).

In Algorithm 9 Lines 55-57, the T_r in the REQUEST message is filtered such that only tags unknown to node u are left in T_r^{new}. If such tags are left, u sends them in an UPDATE message on all edges which have not yet replied with a SUCCESS or FAIL. If a SUCCESS message is in transit to u but u has not yet received it, it might happen that u sends an UPDATE message to a node that has already replied. We therefore include Line 47 that besides checking if the node is in a correct state, also ignores an UPDATE if the node already replied with a SUCCESS.

Handling SUCCESS and FAIL

Algorithm 10 Pseudocode for the implementation of the transaction generation protocol for a node u (`replyToRequests`)

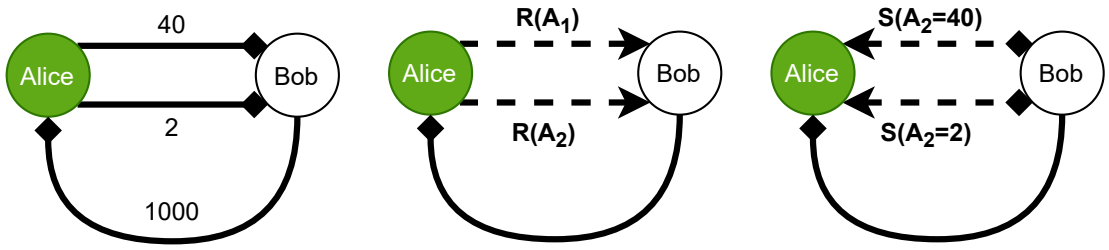
63: **procedure** REPLYTOREQUESTS
64: **if** $n_{\text{sendRequests}} \neq 0 \vee \mathit{roundState} \neq \text{REQ}$ **then return**
65:
66: $\mathit{roundState} \leftarrow \text{SUC}$
67: **for all** $(m = (\text{success}; id, A_i, T_c)$ on edge $j) \in L_{\text{rSUCCESS}}$ **do**
68: **for all** $(t_c, \vec{\Delta}_m) \in T_c$ **do**
69: **if** $t_c \in D^u_{t_c}$ **then** ▷ Case: t_c is a cycle owned by u
70: $L_{t_c} \leftarrow D^u_{t_c}.\texttt{get}(t_c)$
71:
72: **if** $\neg L_{t_c}[3] \vee \vec{\Delta}_m > L_{t_c}[2]$ **then**
73: $D^u_{t_c}.\texttt{put}\left(t_c, \left[L_{t_c}[0], j, \vec{\Delta}_m, \text{true}\right]\right)$
74: **else if** $t_c \notin D^{\neg u}_{t_c} \vee \vec{\Delta}_m > D^{\neg u}_{t_c}.\texttt{get}(t_c)[0]$ **then** ▷ Case: t_c is a cycle not owned by u
75: $D^{\neg u}_{t_c}.\texttt{put}\left(t_c, \left[\vec{\Delta}_m, j, \perp\right]\right)$
76:
77: **if** $A_l = A_u$ **then** ▷ Node started round
78: `commitLeader`()
79: **else** ▷ Forward t_c not owned by u
80: $L_{D^{\neg u}_{t_c}} \leftarrow \left[[t_c, \vec{\Delta}_m, q_o^{u,c}, q_i^{u,c}] \mid (t_c, [\vec{\Delta}_m, q_o^{u,c}, q_i^{u,c}]) \in D^{\neg u}_{t_c}\right]$
81: **for all** $(m = (\text{request}; id, A_i, T_r)$ on edge $j) \in L_{\text{rREQUEST}}$ **do**
82: $N \leftarrow \texttt{splitEqually}(w(j), \left[\vec{\Delta}_m \mid [t_c, \vec{\Delta}_m, q_o^{u,c}, q_i^{u,c}] \in L_{D^{\neg u}_{t_c}}\right])$
83: $T_c^{\neg u} \leftarrow \emptyset$
84: **for** $i \in [0, |N|)$ **do**
85: $T_c^{\neg u} \leftarrow T_c^{\neg u} \cup \left\{(L_{D^{\neg u}_{t_c}}[i][0], N[i])\right\}$
86:
87: **send** $(\text{success}; \nu, A_l, T_c^{\neg u})$ **on edge** j

Similarly to the limited discussion of the REQUEST handler, we choose to also skip the discussion of the SUCCESS and FAIL handler as those are small procedures. Nevertheless, we do

provide their pseudocode in Algorithm 20 in Appendix B. In this section, we focus our attention on the `replyToRequests` procedure defined in Algorithm 10, which is used by both the SUCCESS and the FAIL handler. It can be seen in action in step 4.4h, 4.4i, 4.4j and 4.4k in Figure 4.4.

`replyToRequests` is a procedure that is only intended to run once a node received a reply to all its REQUESTs, whether those are SUCCESS or FAIL replies. To enforce this, a check is done in Line 64. The goal of `replyToRequests` is to produce a SUCCESS message containing a compiled set $T_c^{\neg u}$ of cycle tags that u does not own or, if u is the leader, invoke `commitLeader`. To create $T_c^{\neg u}$, u iterates over all received SUCCESS messages in Line 67 and over all the $(t_c, \vec{\Delta}_m)$ pairs contained within. The node then checks if it owns the t_c by comparing it with the t_c stored in $D^u_{t_c}$. If this is the case, it updates the information in $D^u_{t_c}$ about t_c if either (Line 72): i) the cycle is not yet complete or ii) the $\vec{\Delta}_m$ is larger than the $\vec{\Delta}_m$ stored in $D^u_{t_c}$ for t_c.

As stated before, we consider a cycle c complete if its owner receives a SUCCESS message with the corresponding t_c. A cycle can only be completed *once*, so condition i) is also true only once per t_c. However, more than one SUCCESS message with the same t_c might be received indicating that there are multiple completed cycles C_{t_c} with the same ending edge $q_e^{u,c}$ and t_c, all identified by their different starting edges $q_s^{u,c}$. Such a situation is visualised in Figure 4.5. In such a case, condition ii) forces the node u to pick the starting edge with the largest $\vec{\Delta}_m$ as this cycle best meets the rebalancing objectives. The greedy approach, where u uses all potential cycles, can cause problems when two or more cycles use the same edge q (i.e. overlap on q) and $\vec{\Delta}_m^{c_1} + \vec{\Delta}_m^{c_2} > w(q)$. As u has no knowledge of the topology of R past its neighbours[8], u does not know which q overlap. u must therefore apply caution and only pick one cycle out of C_{t_c}, which is our approach.



(a) Starting graph (b) $T = 1$ (c) $T = 4$

Figure 4.5: A situation in the transaction generation protocol where Alice receives the same tag on two different outgoing edges and must pick the edge with the largest $\vec{\Delta}_m$

A similar approach is taken with cycles not owned by node u in Line 74, although node u does not have to keep track of their completeness given it is not the owner of the cycles. Similarly to the information stored in $D^u_{t_c}$, u stores the $\vec{\Delta}_m$, outgoing edge $q_o^{u,c}$ and incoming edge $q_i^{u,c}$ of the non-owned cycles in $D^{\neg u}_{t_c}$. During the `replyToRequests` invocation, $q_i^{u,c}$ is not yet known and will possibly be set later during the COMMIT message handler.

In the final half of `replyToRequests`, node u takes one of two paths depending on if it is the leader. In case u is the leader (Line 77), it can skip sending SUCCESS messages as the leader owns all cycles it is a part of and has, therefore, no need to inform requesting nodes about cycles it does not own. We show this property to be true by proving Theorem 1.

Theorem 1. *If the round leader l is part of a cycle c, c is always owned by l s.t. $t_c \in D^l_{t_c}$*

In order to prove Theorem 1, we first need to prove Lemma 1.

Lemma 1. *All REQUEST messages send during a round contain a $t_r \in T_r$ that is generated by the leader l s.t. $|T_r \cap D^l_{t_r,q}| \geq 1$*

Proof. We proof Lemma 1 by using a proof by contradiction where we assume that there exists a REQUEST message containing T_r^u send by a node $u \neq l$ where $|T_r^u \cap D^l_{t_r,q}| = 0$ and $|T_r^u| = k$.

[8] u might have some knowledge of the topology of R past its neighbours, but this is not always the case. See Section 4.8 for a detailed discussion.

According to the forwarding rules defined in the REQUEST message handler, each node that receives a REQUEST and that has not forwarded one already during the round, has to add a unique t_r to the received T_r for each $q \in Q_{\text{out}}$ it has to forward the message on. The only node who does not need to receive a REQUEST message to send REQUEST messages on Q_{out} is the leader l, as this is not done in the REQUEST message handler but during the execution of `startRound`.

If we then assume that there exists a REQUEST sent by a node $u \neq l$ with $|T_r^u \cap D_{t_r,q}^l| = 0$ and $|T_r^u| = k$, then there must be a node v that send a REQUEST message to u with $|T_r^v| = k-1$. This is because $u \neq l$ implies that u must first receive a REQUEST message from v before being able to send one. It must also be the case that $|T_r^v| = k-1$ because u added a unique t_r to T_r^v before forwarding it as T_r^u, making T_r^v one smaller than T_r^u.

It must then also be that $v \neq l$, as $T_r^v \subset T_r^u$ and $|T_r^u \cap D_{t_r,q}^l| = 0$. These two statements require $v \neq l$ as when $v = l$, v would have added a $t_r \in T_r^v$ to $D_{t_r,q}^l$ which would result in $|T_r^u \cap D_{t_r,q}^l| > 0$. We can then use the same reasoning as for u, where if $v \neq l$, it must have received a REQUEST message from another node j with a T_r one smaller than what v send. This processes continues until a node i receives from a node z a T_r^z where $|T_r^z| = 1$.

Node $z \neq l$ as if this was the case, similar to v, it would have added a $t_r \in T_r^z$ to $D_{t_r,q}^l$ which would result in $|T_r^u \cap D_{t_r,q}^l| > 0$. However, if $z \neq l$, z must have received a REQUEST message from a node y where $|T_r^y| = 0$. This is a contradiction as the forwarding rules defined in the REQUEST message handler require each node to add its own unique t_r to T_r which implies that $|T_r^y|$ must be larger than 0.

It must therefore be true that $z = l$ as the leader l is the only participant capable of sending a REQUEST where $|T_r| = 1$ without needing to have received a REQUEST message itself. This implies that the leader l must be the first node to send a REQUEST each round and as this REQUEST contains a $t_r \in T_r^l$ which is also in $D_{t_r,q}^l$, it follows that every other REQUEST in the round has $|T_r \cap D_{t_r,q}^l| \geq 1$. □

We then prove Theorem 1 using Lemma 1.

Proof. We proof Theorem 1 by using a proof by contradiction where we assume that if l is part of a cycle c, $t_c \notin D_{t_c}^l$ i.e. c is not owned by l. If $t_c \notin D_{t_c}^l$, this implies that the leader's `checkForCycles` has not generated t_c and added it to $D_{t_c}^l$. This only happens when it finds no intersections between a T_r from a received REQUEST and $D_{t_r,q}^l$. However, according to Lemma 1, all T_r received must contain a t_r for which it holds that $t_r \in D_{t_r,q}^l$. This is a contradiction, so it must therefore hold that if the round leader l is part of a cycle c, it must be the owner of the cycle s.t. $t_c \in D_{t_c}^l$. □

If u is not the leader, u compiles a set $T_c^{\neg u}$ of $(t_c, \vec{\Delta}_m, q_o^{u,c}, q_i^{u,c})$ of cycles it does not own for each edge q it has received a REQUEST on. u uses the information it just stored in $D_{t_c}^{\neg u}$ to do this. Before creating the $(t_c, \vec{\Delta}_m)$ pairs, u first runs a list of $\vec{\Delta}_m$ of all the tags through the `splitEqually` procedure for each $q \in Q_{\text{in}}$ that it received a REQUEST from (Line 82). Once this is done, u sends the $T_c^{\neg u}$ in a SUCCESS message on q (Line 87).

The need for the `splitEqually` step stems from the fact that u has no knowledge which of the t_c will be in T_{c_q}, which is the set containing $(t_c, \vec{\Delta}_{\text{EX}}, \varphi_{\text{setup}})$ tuples which is received in a COMMIT message over q in the next phase of the protocol. Node u only knows that $T_{c_q} \subseteq T_c^{\neg u}$. u also has the requirement that a transaction over q for each of the tuples in T_{c_q} must succeed[9], as no one except their channel partner knows $w(q)$ which prevents other participants from limiting their $\vec{\Delta}_{\text{EX}}$ to prevent $\sum_{\vec{\Delta}_{\text{EX}} \in T_{c_q}} > w(q)$. We therefore make it the responsibility of each participant that all transactions can simultaneously execute on their incoming edges. This in turn requires u to reserve parts of $w(q)$ such that, in the worst case where $T_{c_q} = T_c^{\neg u}$, all $(t_c, \vec{\Delta}_{\text{EX}}, \varphi_{\text{setup}}) \in T_{c_q}$ can successfully execute on $w(q)$. More formally, u must ensure that $\sum_{\vec{\Delta}_{\text{EX}} \in T_{c_q}} \leq w(q)$ even though it has no direct control over $\vec{\Delta}_{\text{EX}}$ as they are set by the cycle owner. As u can only change the $\vec{\Delta}_m$ in $T_c^{\neg u}$, u must therefore ensure that $\sum_{\vec{\Delta}_m \in T_c^{\neg u}} \leq w(q)$ which in turn ensures that $\sum_{\vec{\Delta}_{\text{EX}} \in T_{c_q}} \leq w(q)$. This is done using the `splitEqually` procedure.

[9] In a normal execution of the protocol

The `splitEqually` procedure takes as its inputs an integer representing the rebalancing objective t and a list of demands L_d, and outputs a list of demands that each represent a 'fair' split of t. We provide a sample implementation of it in Algorithm 22 in Appendix B. When called by u in `replyToRequests`, $t = w(q)$ and $L_d = [\vec{\Delta}_m : (t_c, \vec{\Delta}_m, q_o^{u,c}, q_i^{u,c}) \in T_c^{\neg u}]$. By saying 'fair', we mean that each demand $\vec{\Delta} \in L_d$ should get an equal share of t, or more if all the other demands are already satisfied (this to avoid not utilizing all of t). See Figure 4.6 for a visualization of this concept. More formally, `splitEqually` solves a non-linear optimization problem:

Objective: Minimize: $\sum_{\gamma_i \in \Gamma} (-\gamma_i) + \frac{1}{2} (\max(\Gamma) - \min(\Gamma))$
Subject to:
(1) $\forall i \in [0, |L_d|) : \gamma_i \leq \vec{\Delta}_i$
(2) $\sum_{\gamma_i \in \Gamma} \gamma_i \leq t$

In this problem, $\vec{\Delta}_i$ represents the i^{th} $\vec{\Delta} \in L_d$ and Γ a set of variables γ where $|\Gamma| = |L_d|$. This problem has a resemblance to the bin packing problem[10] but with t items of size 1, a fixed number of bins $|L_d|$ and each bin having a fixed capacity $\vec{\Delta}$. We constrain the problem with two conditions. Condition (1) prevents each bin from being filled above its capacity $\vec{\Delta}_i$. We argue that this condition is necessary as there is no reason for a cycle to use more than its already known bottleneck $\vec{\Delta}_i$. Condition (2) ensures that sum of the bin's contents can never exceed t, which in u's case is equivalent to $\sum_{\vec{\Delta}_m \in T_c^{\neg u}} \leq w(q)$.

The first term of the minimization problem maximises $\sum_{\gamma_i \in \Gamma} \gamma_i$. The second term is less intuitive, as it steers the optimal solution to prefer solutions with equally balanced channels that are otherwise equivalent when only looking at the first minimization term. The second term uses the distance between the maximum and minimum variable in Γ to penalize solutions that have a large discrepancy between the two. One can think of two potential optimal solutions when ignoring the second term. In the first, there are two bins of equal capacity and in the second, one bin is empty and one is fully filled. With the second term added, only the first solution is optimal.

We consider the second term to be an important addition as this increases the probability that q will receive a rebalancing transaction. To see why, it is important to remember that u does not know which of the $t_c \in T_c^{\neg u}$ appear in T_{c_q}. If t is spread over more bins compared to putting it all in one bin, there is a higher chance that if a t_c appears in T_{c_q}, it will be for a bin that is at least partially filled. If only the first term is present, u has a larger chance of getting a t_c for an empty bin. Another benefit of the second term is that it prevents favouring cycles with a large $\vec{\Delta}_m$. It is not a good idea to favour edges with a large $\vec{\Delta}_m$ because it is unknown to u how many edges are included in t_c. Compare a cycle c_1 with a $\vec{\Delta}_m$ of only 5 but passing through a 100 edges to a cycle c_2 with a $\vec{\Delta}_m$ of a 100 but passing through only 5 edges. Arguments exist for both options, which we consider a topic for future work. The second term in the optimization problems ascertains that u does not favour any specific option.

Committing cycles

Algorithm 11 specifies the general procedure for committing a node u's owned cycles to the rebalancing graph R. `commit` is called as a subroutine in `commitLeader` and the COMMIT message handler, both of which will be discussed in a later part of this section. The idea behind the `commit` procedure is that it operates on a map D_{q_o}, that maps $Q_{\text{out}} \to T_{c_{q_o}}$, i.e. the outgoing edges of u to a set of $(t_c, \vec{\Delta}_{\text{EX}}, \varphi_{\text{setup}})$ pairs. The `commit` procedure only concerns itself with the committing of owned cycles as this is done both by the leader and every other participant. There is no need for the leader to commit non-owned cycles as there are no cycles that involve the leader where the leader is not the owner. We show this property to be true in Theorem 1. The committing of non-owned cycles is therefore only a part of the COMMIT message handler and not a part of `commit`.

To commit its own cycles, a node u iterates over $D_{t_c}^u$. In the previous section we have shown that during the `replyToRequests` invocation, $D_{t_c}^u$ has been updated to reflect the information contained in incoming SUCCESS messages. For each cycle contained in $D_{t_c}^u$, u first checks if it

[10] See [20] for an explanation of the problem

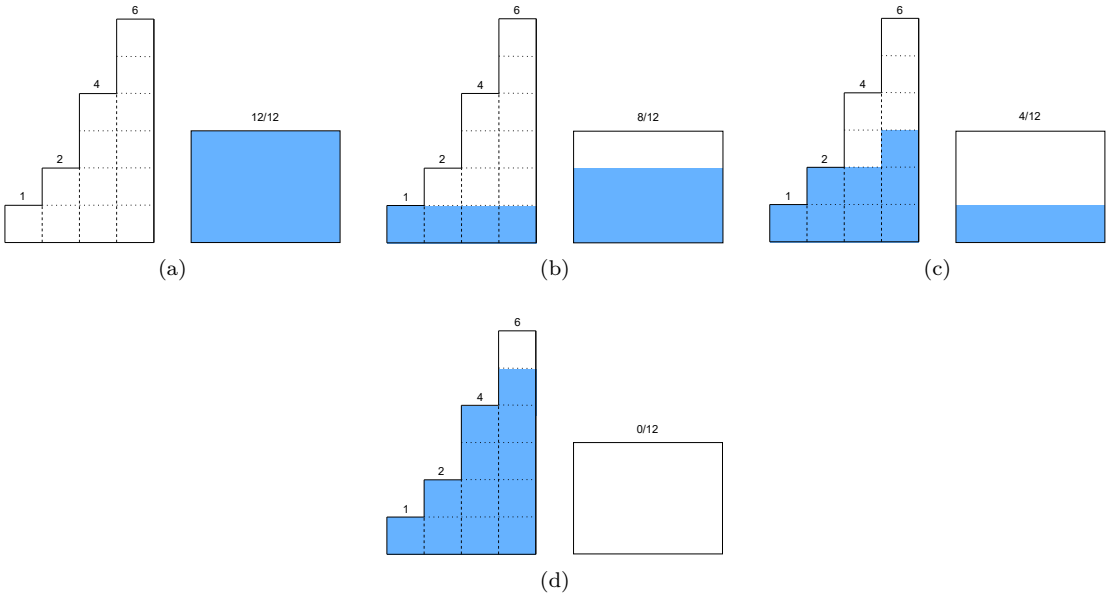


Figure 4.6: A visualization of the optimization problem `splitEqually` tries to solve. On the left are four buckets with capacities 1, 2, 4 and 6, representing the $\vec{\Delta} \in L_d$. On the right is a tank with 12 units of water, representing the rebalancing objective t. The goal is to fill up the buckets as much as possible while also keeping the level of non-filled buckets equal.

is *completed* (Line 90) and if not complete, discards the cycle. We defined what it means for a cycle to be *complete* earlier on during the discussion of the protocol. The reason for adding such a property to our protocol is that it resolves an important case in which two or more nodes think they own the same cycle. An example of such a case is presented in Figure 4.7. Experimental testing has shown that such situations are frequent and cause about 85 % of detected cycles to be discarded. We, therefore, consider it an interesting research direction for future work to find a way to resolve the problem in a different way than discarding the cycles, as this has the potential to increase the efficiency of our protocol.

If an owned cycle is complete and has a demand higher than zero (Line 93), node u creates a conditional transaction on $q_s^{u,c}$ by generating φ_{setup} and φ_{exec}, and invoking $\texttt{pay}_{\text{cond}}$ on $q_s^{u,c}$, where $q_s^{u,c}$ is the starting edge of the cycle. For later reference, u then stores φ_{exec} with the t_c in $D_{t_c,\varphi_{\text{exec}}}$ map and stores $(t_c, \vec{\Delta}_m, \varphi_{\text{setup}})$ in D_{q_o} for sending on $q_s^{u,c}$. The act of storing $\vec{\Delta}_m$ in D_{q_o} transforms it into the $\vec{\Delta}_{\text{EX}}$ that is part of the COMMIT message.

The final part of `commit` concerns the sending of the $(t_c, \vec{\Delta}_m, \varphi_{\text{setup}})$ pairs on each edge q that u received a SUCCESS from (Line 104). As D_{q_o} is a map of $Q_{\text{out}} \to T_{c_{q_o}}$, u simply has to retrieve the correct $T_{c_{q_o}}$ out of D_{q_o} before sending it. Note that the sending part of `commit` is structured such that a COMMIT is *always* send on q, irregardless of $T_{c_{q_o}} = \emptyset$. This is a necessary step as this allows the node receiving COMMITs to know if all nodes have answered its SUCCESS messages.

commitLeader Algorithm 11 specifies the procedure `commitLeader` that is called during the invocation of `replyToRequests`. `commitLeader` is a simple procedure as the leader only has to commit its own cycles (see Theorem 1 as to why) and therefore only calls `commit`. However, we also include a conditional statement that allows the leader to skip waiting for EXEC replies on its COMMITs in the case the leader does not own any cycles (Line 114). If the leader does not own any cycles s.t. $D_{t_c}^u = 0$, only COMMIT messages with an $\emptyset$ are sent which results in the leader also receiving only messages with an $\emptyset$. This waiting is inefficient and skipping the last phase of the protocol allows the leader to start sending NEXT_ROUND messages immediately, potentially improving the runtime of the protocol in certain cases.

Algorithm 11 Pseudocode for the implementation of the transaction generation protocol for a node u (`commit`)

88: **procedure** COMMIT(D_{q_o})
89: **for all** $(t_c, \left[q_e^{u,c}, q_s^{u,c}, \vec{\Delta}_m, \mathit{completed}\right]) \in D^u_{t_c}$ **do**
90: **if** $\neg \mathit{completed}$ **then**
91: **continue**
92:
93: **if** $\vec{\Delta}_m > 0$ **then**
94: $\varphi_{\text{setup}}, \varphi_{\text{exec}} \leftarrow$ generated by a subroutine
95: $D_{t_c,\varphi_{\text{exec}}}$.`put`$(t_c, \varphi_{\text{exec}})$
96: `pay`$_{\text{cond}}(u, \texttt{Opp}(q_s^{u,c}, u), \vec{\Delta}_m, \varphi_{\text{setup}})$
97:
98: $F \leftarrow D_{q_o}$.`get`$(q_s^{u,c})$ ▷ Update or create an entry for $q_s^{u,c}$
99: **if** $F = \perp$ **then**
100: $F \leftarrow \emptyset$
101: $F \leftarrow F \cup \left\{(t_c, \vec{\Delta}_m, \varphi_{\text{setup}})\right\}$
102: D_{q_o}.`put`$(q_s^{u,c}, F)$
103:
104: **for all** $q \in \{j \mid (m = (\text{success}; id, A_i, T_c) \text{ on edge } j) \in L_{\text{rSUCCESS}}\}$ **do**
105: $F \leftarrow D_{q_o}$.`get`(q)
106: **if** $F \neq \perp$ **then**
107: **send** $(\text{commit};\nu, A_l, F)$ **on edge** q
108: **else**
109: **send** $(\text{commit};\nu, A_l, \emptyset)$ **on edge** q

COMMIT message handler Algorithm 11 also specifies the COMMIT message handler. It can be seen in action in step 4.4m, 4.4n, 4.4o and 4.4p in Figure 4.4. Although at first sight a relatively large procedure, the COMMIT message handler can be logically split into three parts: actions to take when receiving a COMMIT from a non-owned cycle, actions to take when receiving a COMMIT from an owned cycle and actions to take when all COMMITs have been received. We discuss these parts in the following paragraphs.

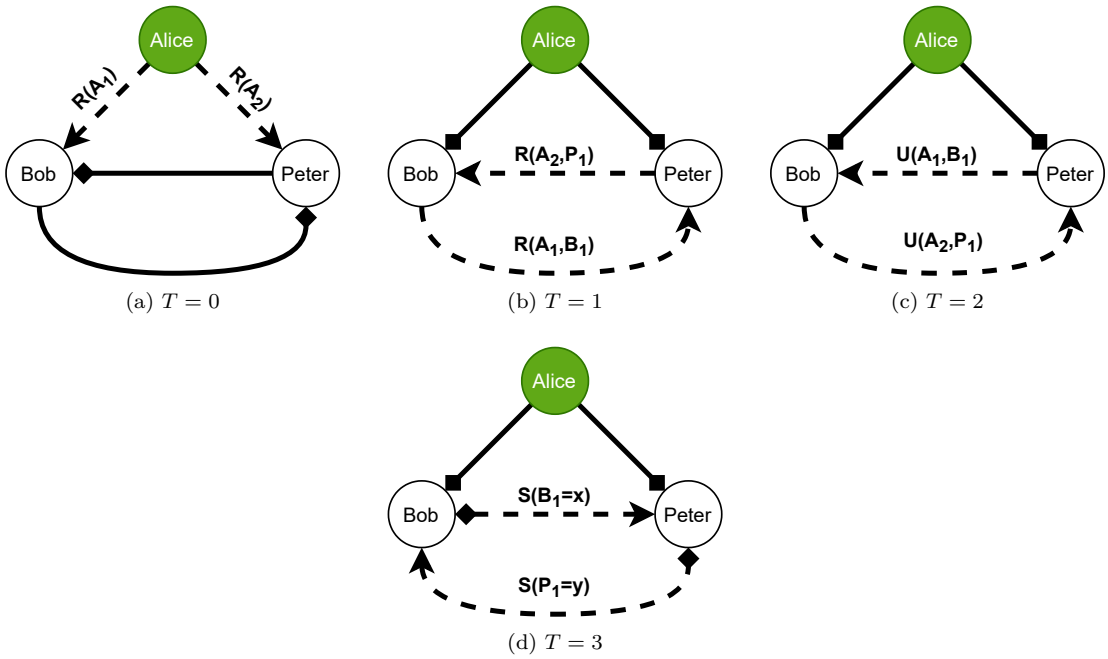
If a received commit is not owned by u (Line 124), u stores it in L_{rCOMMIT}. Once u has received an equal number of COMMITs and REQUESTs, u acts similar to `commit`, creating conditional transactions for cycles and storing these in D_{q_o}. However, the difference between `commit` and Lines 132-140 is that u now does this for all the non-owned cycles it received in the COMMIT messages. In order to know where to continue the cycle (which must be one of u's outgoing edges), u uses the $q_o^{u,c}$ it stored in $D^{\neg u}_{t_c}$. Once done with the non-owned cycles, u must call `commit` such that also its owned cycles will be committed and the COMMIT messages sent.

If u receives a COMMIT message for a cycle c it owns (Line 143), this implies that the COMMIT message it send itself on the cycle's start edge $q_s^{u,c}$ has made it all the way around the cycle to its end edge $q_e^{u,c}$. As u is both the sender and receiver of the conditional transaction of the cycle c, u can securely execute the conditional payment on $q_e^{u,c}$ once it receives a COMMIT message and conditional payment on $q_e^{u,c}$. u cannot immediately execute conditional transactions of non-owned cycles, as this may only happen during the EXEC phase. If u executes the conditional payment of a non-owned cycle immediately when receiving a COMMIT message, it runs the risks of losing its funds which would break the *balance security* property (Definition 1) and with it Requirement 1b.

Finally, if u has received as many COMMITs as $|L_{\text{rREQUEST}}| + |D^u_{t_c}|$ (Line 151), it can move its state to the EXEC state and invoke `checkExecutionSafe`. `checkExecutionSafe` is a small utility procedure of which we provide the implementation in Algorithm 24 in Appendix B. It checks if u has executed all non-owned cycles and if so, set *execSafe* to true. Setting *execSafe* to true indicates that u is fully done with executing its cycles which in turn influences the result of $Pr_{\text{nextRound}}$. If $Pr_{\text{nextRound}}$ evaluates to true, u is allowed to move to the next round. $Pr_{\text{nextRound}}$

Algorithm 12 Pseudocode for the implementation of the transaction generation protocol for a node u (`commitLeader` and COMMIT message handler)

110: **procedure** COMMITLEADER
111: $D_{q_o} \leftarrow \emptyset$ ▷ Map of $q_o \rightarrow \left\{(t_c, \vec{\Delta}_{\text{EX}}, \varphi_{\text{setup}})\right\}$
112: `commit`(D_{q_o})
113:
114: **if** $|D^u_{t_c}| = 0$ **then**
115: `nextRound`()
116:
117: **upon receipt of** $m = (\text{commit}; id, A_i, T_{c_{\text{setup}}})$ **on edge** j **do**
118: **check-action:** $Pr_\nu(id) \rightarrow$ **disallow**, $\neg awake \rightarrow$ **disallow**
119: **check-action:** $Pr_{r,\text{diff}}(A_i) \rightarrow$ **disallow**
120:
121: **if** *roundState* $\neq SUC$ **then return**
122:
123: *roundState* $\leftarrow COM$
124: **if** $j \notin \{q_e^{u,c} \mid (t_c, \left[q_e^{u,c}, q_s^{u,c}, \vec{\Delta}_m, \textit{completed}\right]) \in D^u_{t_c}\}$ **then**
125: $L_{\text{rCOMMIT}} \leftarrow L_{\text{rCOMMIT}} * m$
126:
127: **if** $|L_{\text{rCOMMIT}}| = |L_{\text{rREQUEST}}|$ **then**
128: $D_{q_o} \leftarrow \emptyset$ ▷ Map of $q \rightarrow \left\{(t_c, \vec{\Delta}_{\text{EX}}, \varphi_{\text{setup}})\right\}$
129: **for all** $T^i_{c_{\text{setup}}} \in \{T_{c_{\text{setup}}} \mid (m = (\text{commit}; id, A_i, T_{c_{\text{setup}}}) \text{ on edge } j) \in L_{\text{rCOMMIT}}\}$ **do**
130: **for all** $(t_c, \vec{\Delta}_{\text{EX}}, \varphi_{\text{setup}}) \in T^i_{c_{\text{setup}}}$ **do**
131: $[\vec{\Delta}, q_o^{u,c}, q_i^{u,c}] \leftarrow D^{\neg u}_{t_c}$.`get`$(t_c)$
132: $D^{\neg u}_{t_c}$.`put`$(t_c, [\vec{\Delta}_{\text{EX}}, q_o^{u,c}, j])$ ▷ Store the incoming edge of the cycle
133:
134: $\texttt{pay}_{\text{cond}}(u, \texttt{Opp}(q_o^{u,c}, u), \vec{\Delta}_{\text{EX}}, \varphi_{\text{setup}})$
135:
136: $F \leftarrow D_{q_o}$.`get`$(q_o^{u,c})$ ▷ Update or create an entry for q_o
137: **if** $F = \perp$ **then**
138: $F \leftarrow \emptyset$
139: $F \leftarrow F \cup \left\{(t_c, \vec{\Delta}_{\text{EX}}, \varphi_{\text{setup}})\right\}$
140: D_{q_o}.`put`$(q_o^{u,c}, F)$
141:
142: `commit`(D_{q_o})
143: **else**
144: $L_{\text{rCycleCOMMIT}} \leftarrow L_{\text{rCycleCOMMIT}} * m$
145:
146: **for all** $(t_c, \vec{\Delta}_{\text{EX}}, \varphi_{\text{setup}}) \in T_{c_{\text{setup}}}$ **do**
147: $\varphi_{\text{exec}} \leftarrow D_{t_c, \varphi_{\text{exec}}}$.`get`$(t_c)$
148: $\texttt{pay}_{\text{exec}}(u, \texttt{Opp}(j, u), \vec{\Delta}_{\text{EX}}, \varphi_{\text{exec}})$
149: **send** $(\text{exec}; \nu, A_l, t_c, \varphi_{\text{exec}})$ **on edge** j
150:
151: **if** $|L_{\text{rCOMMIT}}| + |L_{\text{rCycleCOMMIT}}| = |L_{\text{rREQUEST}}| + |D^u_{t_c}|$ **then**
152: *roundState* $\leftarrow$ EXEC
153: `checkIfExecutionSafe`()
154:



(a) $T = 0$ (b) $T = 1$ (c) $T = 2$

(d) $T = 3$

Figure 4.7: An example invocation of the transaction generation protocol showing the importance of the *complete* property. In step (c), as both Peter and Bob receive an UPDATE, they both think they own the cycle Bob - Peter - Bob so they both respond with a SUCCESS in step (d). However, because both Bob and Peter do not receive a SUCCESS message with their own cycle tag (B_1 for Bob and P_1 for Peter), the cycle is not complete and is discarded.

is also evaluated by `checkExecutionSafe`.

Executing conditional transactions

Algorithm 13 specifies the EXEC message handler, which is the last procedure we discuss in this chapter. Appendix B specifies all the procedures of the transaction generation protocol, including the ones only mentioned in this chapter. The EXEC message handler can be seen in action in step 4.4p, 4.4q, 4.4r and 4.4s in Figure 4.4.

A relatively small message handler, the EXEC message handler can only process EXEC messages once the state of u is EXEC and the t_c contained in the EXEC message matches one of the non-owned cycles u knows about (Line 159). u can then execute the conditional transaction on $q_i^{u,c}$ in a similar manner as to how it did for its own cycles in the COMMIT message handler. Once u has executed the transaction and send the EXEC message on $q_i^{u,c}$, u again runs `checkIfExecutionSafe`. This moves u to the next round if $Pr_{\text{nextRound}}$ evaluates to true.

4.7 Protocol termination

For both the participant discovery protocol and the transaction generation protocol, we guarantee their termination using two timeouts that are kept by each node u. These timeouts ensure that in any case where u *expects* a message m, u can always terminate even if it does not receive the expected message m. We define *expecting a message* in Definition 11.

Definition 11 (Expected message)**.** *A node u expects a message m if the receipt of m is required to advance it towards the next state S of the protocol.*

The first timeout we define is a global timeout $O_{\text{glob}}^{m_i}$, which is started when u expects a specific message m_i and triggers forced termination of u if it expires. It resets once u receives m_i. u holds a timeout $O_{\text{glob}}^{m_i}$ for every message m_i it expects to receive and never reuses the same timeout.

Algorithm 13 Pseudocode for the implementation of the transaction generation protocol for a node u (EXEC message handler)

155: **upon receipt of** $m = (\text{exec};id, A_i, t_c, \varphi_{\text{exec}})$ **on edge** j **do**
156: **check-action:** $Pr_{\nu}(id) \rightarrow$ **disallow**, $\neg awake \rightarrow$ **return**
157: **check-action:** $Pr_{r,\text{diff}}(A_i) \rightarrow$ **return** , $roundState = \text{COM} \rightarrow$ **defer**
158:
159: **if** $roundState = \text{EXEC} \wedge t_c \in D_{t_c}^{\neg u}$ **then**
160: $[\vec{\Delta}, q_o^{u,c}, q_i^{u,c}] \leftarrow D_{t_c}^{\neg u}.\texttt{get}(t_c)$
161: $\texttt{pay}_{\text{exec}}(u, \texttt{Opp}(q_i^{u,c}, u), \vec{\Delta}, \varphi_{\text{exec}})$
162: $D_{t_c}^{\neg u}.\texttt{remove}(t_c)$
163:
164: **send** $(\text{exec};\nu, A_l, t_c, \varphi_{\text{exec}})$ **on edge** $q_i^{u,c}$
165: `checkIfExecutionSafe`()
166:

The second timeout is a channel-specific timeout O_{chan}^{e}, of which there exists one for every channel e that is potentially participating[11] in the protocols. O_{chan}^{e} is started if a specific message m_i is expected to be received on e and is reset on the receipt of m_i on e. If O_{chan}^{e} expires, node u can consider the channel e to be unresponsive and take appropriate action. Appropriate action may include removing it from the list of nodes that it expects a response from, removing it from either Q_{out} or Q_{in} or terminating the protocol. If u deems a channel e to be unresponsive, the channel can never again be used during that particular execution of the protocol.

Having introduced the two timeouts, we will now discuss the need for including them. We intend for O_{chan}^{e} to be the main timeout that will be used often. It is common in the execution of distributed protocols to have some unresponsive neighbours, so a neighbour not replying over a channel is a common cause that should be handled gracefully. O_{chan}^{e} on its own does not guarantee termination as it only covers the case where u sends a message m_i on e and expects one or more messages $\{m_i, m_{i+1}, \ldots\}$ as a reply on e. If it is unknown on which edge e u should expect a message such as a REQUEST, another timeout is required. We, therefore, include $O_{\text{glob}}^{m_i}$ to act as a fallback to the O_{chan}^{e} timeouts in the cases where u expects m_i but cannot associate it to an edge e. In that case, at some point in time, $O_{\text{glob}}^{m_i}$ will expire which forces u to terminate. Using these two timeouts we can guarantee termination in any case where u expects a message, which is the case after the execution of any message handler in both our protocols, after `start` in the participant discovery protocol and after `startRound` and `nextRound` in the transaction generation protocol.

4.8 Security and privacy analysis

In this section, we discuss the security and privacy of our protocols. In Table 4.9 we provide an overview of the analysis that is provided in the following sections.

4.8.1 Balance security

One of the most important properties of multi-hop transactions in a PCN is *balance security*, which we define in Definition 1. Informally, balance security is the guarantee that an honest intermediary can lose its coins in a multi-hop transaction. As balance security only concerns multi-hop transactions, we limit our discussion in this section to the protocols where it is relevant, which is only our transaction generation protocol. In this section, we show that the transaction generation protocol guarantees balance security in the presence of an adversary as defined in Section 4.1.5 under the assumptions of Section 4.1.5. The most important assumption in Section 4.1.5 relating

[11] 'Potentially' refers to the fact that it is not yet determined in the participation discovery protocol if a channel will participate. If a channel is found not to be participating, its timeout can be removed.

Table 4.9: Security and privacy properties achieved with our protocols

Applicable to ... protocol	**Property**	**Achieved?**
Participation discovery	Participation anonymity	Depends on graph topology
Transaction generation	Balance security	Yes
	Balance conservation	Yes
	Channel balance privacy	Depends on graph topology
	Path privacy	Yes
	Value privacy	Yes
	Sender/receiver privacy	Depends on graph topology
	Relationship anonymity	No

to this section is that the $\texttt{pay}_{\text{condComp}}(s, r, x, p)$ function provides balance security, which we use during our proof to show that the transaction generation protocol guarantees balance security. ‘

Theorem 2. *If an honest participant u is part of the execution ν of the transaction generation protocol, it has balance security as defined in Definition 1.*

Proof. We show that u has balance security during the execution ν of the transaction generation protocol using a direct proof. Our proof is build on showing that the $\texttt{pay}_{\text{cond}}$ and $\texttt{pay}_{\text{exec}}$ invocations during the execution of the protocol are equivalent to a single $\texttt{pay}_{\text{condComp}}$. We assume that $\texttt{pay}_{\text{condComp}}$ provides balance security.

In the protocol as defined in Subsection 4.6.3, we only consider the locations where $\texttt{pay}_{\text{cond}}$ and $\texttt{pay}_{\text{exec}}$ are invoked when they involve non-owned cycles. This is because if u owns a cycle, u acts as both the sender and receiver of a multi-hop transaction. As balance security only concerns intermediaries of multi-hop transactions, we are not interested in the cases when $\texttt{pay}_{\text{cond}}$ and $\texttt{pay}_{\text{exec}}$ are used by u for its owned cycles.

There are two locations where $\texttt{pay}_{\text{cond}}$ and $\texttt{pay}_{\text{exec}}$ are invoked for non-owned cycles. $\texttt{pay}_{\text{cond}}$ is invoked in the COMMIT message handler in Line 134 in Algorithm 12 and $\texttt{pay}_{\text{exec}}$ is invoked in the EXEC message handler in Line 161 in Algorithm 13.

In Algorithm 12, $\texttt{pay}_{\text{cond}}$ is invoked to create a conditional transaction $\beta_{u,v}$ for a non-owned cycle c on the outgoing cycle edge $q_o^{u,c}$ from u to the node $v = \texttt{Opp}(q_o^{u,c}, u)$ with the value $\vec{\Delta}_{\text{EX}}$ and transaction information φ_{setup}. u invokes $\texttt{pay}_{\text{cond}}$ in response to a COMMIT message from a node $k = \texttt{Opp}(q_i^{u,c}, u)$ on the cycle's incoming edge $q_i^{u,c}$. This message must contain a non-owned cycle tag t_c and transaction information φ_{setup}. In addition to the COMMIT message, u must also receive a conditional transaction $\beta_{k,u}$ from k on $q_i^{u,c}$ with value $\vec{\Delta}_{\text{EX}}$ and based on the information provided in φ_{setup}. Once u has finished executing $\texttt{pay}_{\text{cond}}$, there exists a path of conditional transactions $p_f = k \to u \to v$. This procedure is equivalent to the invocation of $\texttt{pay}_{\text{condP}}(k, v, \vec{\Delta}_{\text{EX}}, p_f, \varphi_{\text{setup}})$ because we defined $\texttt{pay}_{\text{condP}}$ to be equivalent to a series of $\texttt{pay}_{\text{cond}}$ invocations over a path p with the same value x and φ_{setup}.

u invokes $\texttt{pay}_{\text{exec}}$ in Algorithm 13 when it receives an EXEC message from v on $q_o^{u,c}$. This EXEC message must contain a non-owned cycle tag t_c that matches the t_c it created $\beta_{u,v}$ for and must also contain φ_{exec}. In addition to the EXEC message, outside of the protocol, v executes $\beta_{u,v}$. When u invokes $\texttt{pay}_{\text{exec}}$, u executes transaction $\beta_{k,u}$. This again creates a path $p_r = v \to u \to k$ that is the inverse of p_f, which makes the execution procedure equivalent to $\texttt{pay}_{\text{execP}}(v, k, \vec{\Delta}_{\text{EX}}, p_r, \varphi_{\text{exec}})$.

The combined process of creating conditional transactions and executing them is therefore equivalent to the invocation of $\texttt{pay}_{\text{condComp}}(k, v, \vec{\Delta}_{\text{EX}}, p_f)$, as we defined $\texttt{pay}_{\text{condComp}}$ to be equivalent to a $\texttt{pay}_{\text{condP}}$ and $\texttt{pay}_{\text{execP}}$ with the same parameters. As we assumed that $\texttt{pay}_{\text{condComp}}$ provides balance security and as u is an intermediary in p_f, it must therefore hold that u has balance security during the execution of the transaction generation protocol. □

4.8.2 Balance conservation

Another important property that we try to achieve with our design is *balance conservation*, which we define in Definition 2. Informally, balance conservation means that a participant has the same total balance before and after the execution of the rebalancing algorithm. Balance conservation is a necessary property that is a unique design requirement for on-demand rebalancing algorithms. The authors of Revive [18], for example, have written the property as a constraint in their linear program, such that all solutions produced by the program have balance conservation. The on-demand rebalancing algorithm from Pickhardt and Nowostawski [40] works by having nodes find cycles to themselves using varying strategies. The authors allow a node to only execute a circular multi-hop transaction if such a cycle is found, thereby guaranteeing balance conservation.

Although balance conservation is also required in on-the-fly rebalancing algorithms, its importance in their design is less as they do not involve cyclic multi-hop transactions. Instead, on-the-fly algorithms such as [6, 11, 27] only influence the transaction path of an existing transaction such that the transaction rebalances one or more channels along the way. Because there are no cyclic multi-hop transactions involved in on-the-fly rebalancing algorithms, only the balance conservation of intermediaries is a necessary requirement. However, this is already achieved by the balance security property we discussed in the previous section.

We will show in this section that the transaction generation protocol achieves balance conservation for its participants in the presence of an adversary as defined in Section 4.1.5. We do not discuss the participant discovery protocol as it does not involve any changes in channel balances. We first proof Lemma 2 in order to later use it to prove Theorem 3.

Lemma 2. *If a node u is the owner of a cycle $c = \{u, i_1, i_2, \ldots, i_n, u\}$ and u sends an outgoing conditional transaction $\beta_s^{u,c} = \texttt{pay}_{cond}(u, i_1, \vec{\Delta}_{EX}, \varphi_{setup})$ on $q_s^{u,c}$, receives an incoming conditional transaction $\beta_e^{u,c} = \texttt{pay}_{cond}(i_n, u, \vec{\Delta}_{EX}, \varphi_{setup})$ on $q_e^{u,c}$ and executes $\beta_e^{u,c}$ using φ_{exec}, we can say that this is equivalent to $\texttt{pay}_{condComp}(u, u, \vec{\Delta}_{EX}, \{u, i_1, i_2, \ldots, i_n, u\})$ using the same φ_{setup} and φ_{exec}.*

Proof. We use Theorem 2 to show that Lemma 2 is true. In Theorem 2 we have shown that the process of setting up and executing conditional transactions along the path $p_f = \{k, u, v\}$ is equal to the invocation of $\texttt{pay}_{\text{condComp}}(k, v, \vec{\Delta}_{\text{EX}}, p_f)$. If we rewrite p_f to $p_f^u = \{u, i_1, u\}$, we have a similar path but now with i_1 as the intermediary and u as both the start and end of the path, creating a cycle. We can then extend p_f^u by adding more intermediaries such that $p_f^u = c = \{u, i_1, i_2, \ldots, i_n, u\}$, which is equivalent to $\texttt{pay}_{\text{condComp}}(u, u, \vec{\Delta}_{\text{EX}}, p_f^u)$. □

Theorem 3. *If an honest participant u is part of the execution ν of the transaction generation protocol, it has balance conservation as defined in Definition 2.*

Proof. We show Theorem 3 to be true using a direct proof. Our proof is built on showing that at any moment during the protocol, u can take an action that preserves balance conservation. There are two situations in which u might break the balance conservation property. The first situation occurs in the case of owned cycles and the second situation occurs in the case of non-owned cycles.

For every cycle c u owns, u has the option to generate $\varphi_{\text{setup}}^{u,c}$, $\varphi_{\text{exec}}^{u,c}$ and create an outgoing conditional transaction $\beta_s^{u,c} = \texttt{pay}_{\text{cond}}(u, k, \vec{\Delta}_{\text{EX}}^{u,c}, \varphi_{\text{setup}}^{u,c})$ where $k = \texttt{Opp}(q_s^{u,c}, u)$. u can then send $\beta_s^{u,c}$ alongside a COMMIT message on the cycle's starting edge $q_s^{u,c}$ and wait for an incoming conditional transaction $\beta_e^{u,c} = \texttt{pay}_{\text{cond}}(v, u, \vec{\Delta}_{\text{EX}}^{u,c}, \varphi_{\text{setup}}^{u,c})$ to arrive on the cycle's ending edge $q_e^{u,c}$, where $v = \texttt{Opp}(q_e^{u,c}, u)$. If u decides to send $\beta_s^{u,c}$ on $q_s^{u,c}$, it commits to a demand change on $q_s^{u,c}$ if $\beta_s^{u,c}$ executes. This does not break the balance conservation property as $\beta_s^{u,c}$ can only be executed using $\varphi_{\text{exec}}^{u,c}$, which is only known to u because u generated it. Barring u accidentally revealing $\varphi_{\text{exec}}^{u,c}$ due to activities outside the protocol itself and given that u is honest and follows the protocol, u only reveals $\varphi_{\text{exec}}^{u,c}$ to v if it receives $\beta_e^{u,c}$.

If $\beta_e^{u,c}$ arrives and u reveals $\varphi_{\text{exec}}^{u,c}$ to v in order to execute $\beta_e^{u,c}$, it holds that $\sum_{q \in Q_{\text{before}}^u} w(q) < \sum_{q \in Q_{\text{after}}^u} w(q)$ as u has definitively received coins on $q_e^{u,c}$ but the transaction $\beta_s^{u,c}$ on $q_s^{u,c}$ has not yet executed. Therefore u has not finalized sending coins on $q_s^{u,c}$. If the adversary is present in the cycle c, it might not reveal $\varphi_{\text{exec}}^{u,c}$ to its predecessor in the cycle c, thereby ensuring that $\beta_s^{u,c}$ never

executes. This causes u to have received coins without spending any, which breaks the balance conservation property. However, we do not consider this a problem because the combination of the balance security property and Lemma 2 requires that the coins must have come from the adversary. If such a situation arises, we consider the coins gained by u a penalty to the adversary for acting dishonestly. If the adversary is *not* present in c or chooses to follow the protocol, $\beta_s^{u,c}$ eventually executes and u spend as many coins as it received, resulting in $\sum_{q \in Q^u_{\text{before}}} w(q) = \sum_{q \in Q^u_{\text{after}}} w(q)$.

If u does not own a cycle c and it holds that $u \in c$, u receives a conditional transaction $\beta_i^{\neg u,c}$ on $q_i^{\neg u,c}$. According to the protocol, u is expected to forward this conditional transaction on $q_o^{\neg u,c}$. Receiving $\beta_i^{\neg u,c}$ does not break u's balance conservation as u can only execute $\beta_i^{\neg u,c}$ using a φ_{exec} it does not hold. As u is honest and the protocol does not provide other ways for u to obtain φ_{exec}, u's only possibility to obtain φ_{exec} is to create a $\beta_o^{\neg u,c}$ using the same amount and φ_{setup} as $\beta_i^{\neg u,c}$. u then sends $\beta_o^{\neg u,c}$ on $q_o^{\neg u,c}$. We have proven in Theorem 2 that if u is an intermediary, the process of receiving conditional transactions and forwarding them for non-owned cycles provides balance security. This therefore means that $\sum_{q \in Q^u_{\text{before}}} w(q) \leq \sum_{q \in Q^u_{\text{after}}} w(q)$ as u cannot lose any coins during this operation. However, u can also not *gain* coins as u first spends coins during $\beta_o^{\neg u,c}$, before it can gain coins by executing $\beta_i^{\neg u,c}$. Together with the balance security property it must therefore hold that $\sum_{q \in Q^u_{\text{before}}} w(q) = \sum_{q \in Q^u_{\text{after}}} w(q)$ and that the process of forwarding and executing conditional transactions for non-owned cycles provides balance conservation.

As in both situations where u owns or does not own a cycle, balance conservation is provided, we can conclude that an honest u always has balance conservation as defined in Definition 2 during the execution ν of the transaction generation protocol, except for u possibly gaining coins when u owns a cycle that contains an adversary actively withholding $\varphi^{u,c}_{\text{exec}}$. □

4.8.3 Denial of Service

The final security aspect we discuss in the relation to our protocols is a Denial of Service (DoS) attack by an adversary who has corrupted k participants. We consider both our protocols to be susceptible to a DoS attack with varying impacts depending on the stage of the protocols in which the adversary starts the attack.

In a DoS attack on the transaction generation protocol before the COMMIT stage and in a DoS attack on the participant discovery protocol, we assume that the adversary chooses to pick k participants to corrupt such that all paths between honest participants pass through a corrupted participant. If the adversary then stops forwarding the messages it receives on all its corrupted participants, all honest nodes are isolated and cannot receive any messages. If this situation holds indefinitely, each honest participant u has to wait until all timeouts O^e_{chan} for all $e \in E^u$ expire or wait until all $O^{m_i}_{\text{glob}}$ for all expected messages m_i expire. In both cases, u terminates. Another result occurs if we assume that the adversary knows the length of the timeouts O and that the goal of the adversary is not to cause the honest participants to terminate but instead to inflate the protocol's time complexity. In that case, the adversary can release the suppressed messages just before the timeouts expire, which causes the honest participants to reset their timeout and continue the protocol. Instead of the time complexity of the protocol being determined by the message delay function d_m and the number of messages M, the time complexity is then determined by a linear combination of O^e_{chan} and $O^{m_i}_{\text{glob}}$ multiplied by M.

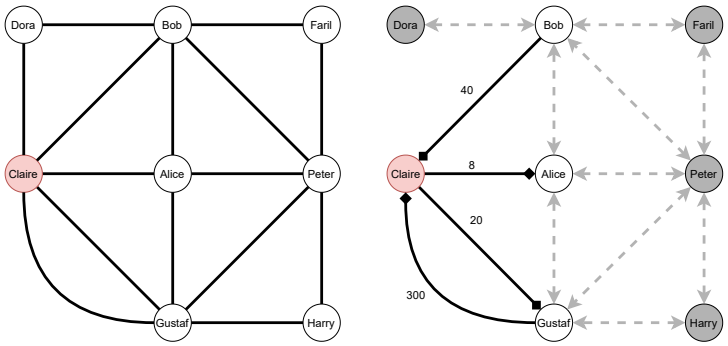
The impact of a DoS attack on the transaction generation protocol in and after the COMMIT stage is slightly different than discussed earlier. If a node u is in the COMMIT stage, it means that it has possibly created conditional multi-hop transactions for owned cycles and/or it is part of a multi-hop transaction for non-owned cycles. If now the adversary starts the DoS attack and continues it indefinitely, u has to wait for the same timeouts as discussed earlier. However, because u also has ongoing multi-hop transactions, it has to wait before they expire before it can retrieve the coins it staked as part of the conditional transaction. This introduces an additional element on top of the earlier stated time complexity.

4.8.4 Participation anonymity

We discuss participation anonymity as the first privacy property of our protocols as some of the properties discussed in the next sections can be reduced to participation anonymity. Participation

anonymity is a property that might hold for each individual node and is defined in Definition 3. In the definition, we state that an adversary can only break the participation anonymity of an honest participant node i if it can break i's node participation anonymity (Definition 4) and the edge participation anonymity (Definition 5) of all edges connected to i, $e \in E_i$. If the adversary fails to break the participation anonymity for all nodes $i \in V$, it cannot construct the full rebalancing graph R based on G outside the set of corrupted nodes and their edges. We assume that the adversary knows the full topology of G as in Lighting, the topology of G is known to all nodes and spread using a gossiping protocol [41].

As an example of participation anonymity, let us take Subfigure 4.8a to represent G. Subfigure 4.8b then represents a graph R based on G with node participation anonymity for all honest $i \in V$ and edge participation anonymity for E_h, which we define as all edges between honest nodes. Because of the assumption of participation anonymity, the adversary – represented by Claire – does not know the participation status of Faril, Dora, Peter or Harry. Claire also does not know whether there are participating edges between Bob, Alice, Gustaf or any of the other nodes. For convenience, we define the subgraph of vertices and edges that are known by the adversary to be participating as (V_p, E_p).



(a) (b)

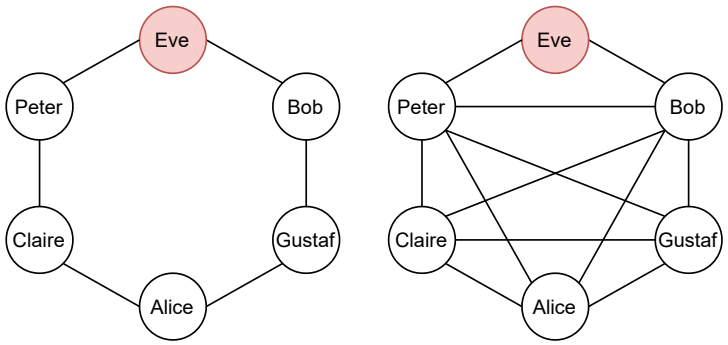
Figure 4.8: a) represents the PCN before the participation discovery protocol is started. b) represents the graph after the transaction generation protocol has started and woken up. Red nodes represent the adversary and grey edges and nodes represent the edges and nodes that the adversary does not know of concerning their participation, assuming participation anonymity.

Based on our analysis of Subfigure 4.8b, it is also clear that the participation anonymity is heavily dependent on the topology of G and the number of corrupted nodes $|C|$. If G is a star topology with Claire at the centre and all nodes have only one edge connecting them to Claire, Claire would immediately know which edges and nodes are participating. We, therefore, look at two more interesting cases in the shape of a graph with a single cycle and a graph with multiple cycles, shown in Figure 4.9.

Single cycle graph In Subfigure 4.9a, G is a single cycle graph and Eve is the adversary. If we assume that Eve starts the protocol, she has to send an INVITE to Bob and Peter containing (among other information) the hop count h_c, that she sets herself. Upon receiving the INVITE message, Bob and Peter either forward the INVITE if $h_c - 1 > 0$ or else, reply with an ACCEPT containing their own anonymous identity A or reply with a DENY. If Peter or Bob reply with an ACCEPT, Eve has broken their node participation anonymity. As Peter and Bob also send their ACCEPT over $e_{\text{Eve,Peter}}$ and $e_{\text{Eve,Bob}}$, Eve knows these edges are participating which breaks their edge participation anonymity and resulting in Peter and Bob losing their participation anonymity.

We can also show that Eve can partially or fully break the participation anonymity of Claire, Gustaf and Alice. If Eve wants to break their participation anonymity, she has to increase her h_c such that the INVITE is forwarded past Bob and Peter. After sending the INVITE, at some later point in time, Eve receives a list of participants P_p from Peter and P_b from Bob.

If then Claire and Gustaf deny, it must be that $|P_p| = 1$ and $|P_b| = 1$, which reduces the case



(a) Only one cycle (b) Many cycles

Figure 4.9: Two graphs G with different amounts of cycles. Eve represents the adversary.

to the case where $h_c = 1$. If Claire and/or Gustaf accept, it still is simple to deduce based on P_p and P_b which of them accepted and as the INVITE message can take only one path, Eve can determine if the edges $e_{\text{Peter,Claire}}$ and/or $e_{\text{Bob,Gustaf}}$ are participating. If Alice also accepts one of her edges and $|P_p| = 3$ and $|P_b| = 2$, Eve knows that $e_{\text{Claire,Alice}}$ must be participating. If $|P_p| = 2$ and $|P_b| = 3$, Eve knows that $e_{\text{Gustaf,Alice}}$ must be participating. If Alice accepts both of her edges then depending on message arrival, P_p and P_b form a (P_p, P_b) combination from the following set: $\{(1,6),(2,5),(3,4),(4,3),(5,2),(6,1)\}$. This tells Eve that both Alice's edges are participating. This therefore shows that using only the received P_p and P_b, Eve is able to break the participation anonymity of all nodes in Subfigure 4.9a.

Multiple cycle graph For a more complex graph, such as Subfigure 4.9b, it quickly becomes difficult for Eve to use the received sets of participants to find participating edges and nodes. In Subfigure 4.9b, all the honest nodes form a subgraph of G that is a complete graph. We first look at the probability of Eve breaking the node participation anonymity. When Eve receives P_b and P_p and $|P_b| + |P_p| \geq 7$, Eve knows that all nodes have accepted at least one edge. This breaks their node participation anonymity. If $|P_b| + |P_p| < 7$, Eve knows one or more nodes are not participating and has to guess from the set of $\{Claire, Alice, Gustaf\}$. If we consider edge participation anonymity we can quickly see this is more difficult than guessing nodes, as there are a total of 12 edges in Subfigure 4.9b. If one edge between two honest nodes does not participate, say $e_{\text{Claire,Gustaf}}$, this is not necessarily reflected in P_b or P_p as almost all nodes can still reach the other nodes over the participating edges.

If the adversary wishes to increase its chances, it should strive to increase the degree of its corrupted nodes by opening additional channels, preferably to nodes with a high degree. This allows the corrupted nodes to receive more sets of participants which allows the adversary to rule out certain paths the INVITE message may have taken. Another strategy involves the adversary corrupting additional nodes, which is more effective than increasing the degree of already corrupted nodes. Corrupting nodes allows the adversary to compare the sets of participants received by all the corrupted nodes, which provides more insight for the adversary compared to Eve receiving a complete set of participants. Both strategies can be run in parallel for maximum effect.

We have already seen that going from Subfigure 4.9a to Subfigure 4.9b increased the difficulty of breaking the participation anonymity of the honest nodes. If the honest participants wish to increase the difficulty even further, they should strive to create a topology of G with a large number of cycles and edges between honest nodes. If we denote the set of corrupted nodes to be C, the set of neighbours of the corrupted nodes $N(C)$, the set of edges between C and their neighbours E_{Corr} and $G_{\text{Corr}} = (N(C)+C, E_{\text{Corr}})$, then the graph of all nodes and edges outside the corrupted nodes and their neighbours can be denoted as $G' = G - G_{\text{Corr}}$. If $G' = (V', E')$, then we can state that if $V' \gg N(C) + C$ and $E' \gg E_{\text{Corr}}$, the probability decreases that the adversary can break the participation anonymity. As it is likely that honest nodes have no insight on which nodes are corrupted it is, therefore, beneficial to increase the number of participants and participating

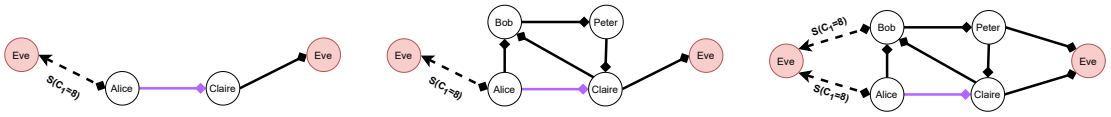
edges to increase the probability that $V' \gg N(C) + C$ and $E' \gg E_{\text{Corr}}$. For a network such as Lightning, which exhibits a heavy reliance on a limited number of highly connected 'hub' nodes [33, 45], we advise that at least multiple of such nodes are involved in the protocol to reduce the likelihood that all of them are corrupt.

Nevertheless, we must conclude that our protocol does not provide participation anonymity in all situations according to Definition 3, as there is a non-negligible probability that Eve can break the node participation anonymity or the edge participation anonymity. However, we emphasise that this is heavily dependent on the capabilities of the adversary. If the adversary has only corrupted one or two nodes with a low degree in a graph with hundreds of participants and edges, we consider it unlikely that the adversary is capable of breaking the participation anonymity of nodes that are far away from the adversary. This specifically applies in case the PCN is the Lightning Network, which at the moment of writing has $\approx 12\,000$ nodes present on the network [24]. We consider a full analysis of participation anonymity to be outside the scope of this thesis and an interesting topic for future work.

4.8.5 Channel balance privacy

In Definition 6, we define a channel $e_{u,v}$ to have channel balance privacy if there exists a negligible probability that the adversary can determine the balances $b(e_{u,v}, u)$ or $b(e_{u,v}, v)$. In our transaction generation protocol, the channel balances $b(e_{u,v}, u)$ and $b(e_{u,v}, v)$ are only accessed during the `negotiateObjective` procedure that is invoked in the `wakeUp` procedure. The `negotiateObjective` procedure outputs a weighted edge q with a weight $w(q) = \vec{\Delta}$ that is the rebalancing objective. In our design, we do not define the relationship that `negotiateObjective` creates between the channel balances $b(e_{u,v})$ and $\vec{\Delta}$. This relationship can be different for any pair of nodes. In this discussion, however, we assume all node pairs to use a linear relationship that is known to the adversary. We do this as we expect many negotiation schemes to be relatively simple such as one that produces equal balances ($b(e_{u,v}, u) = b(e_{u,v}, v)$) and assuming a linear relationship known to the adversary provides a weak assumption on the privacy provided by the `negotiateObjective` procedure.

As we consider the adversary to be able to deduce the channel balance from $\vec{\Delta}$, it has now become the question if our protocol prevents the adversary from mapping obtained $\vec{\Delta}$s to specific rebalancing edges q. In our protocol, if the adversary has corrupted a node u, there is only a possibility for nodes outside $N(u)$ to have channels that have channel balance privacy. This is because in Definition 6, channel balance privacy only applies to channels owned by two honest nodes. We, therefore, only look at cases where the adversary wishes to find out the channel balance of an edge between two honest nodes.



(a) The adversary knows that the $S(C_1 = 8)$ message can only have taken the path $\{q_{\text{Alice,Claire}}\}$

(b) The adversary must guess the path of the $S(C_1 = 8)$ message because there are multiple paths that the message can have taken and one of these paths contains a cycle, which increases the difficulty of guessing $w(q_{\text{Alice,Claire}})$

(c) If the adversary has more information, such as edges connected to nodes that are part of P_h, it can reduce the number of possible paths that the $S(C_1 = 8)$ message can have taken

Figure 4.10: Example showing how the graph topology between two honest nodes affects the ability of the adversary to determine the path that a $S(C_1 = 8)$ message has taken. The purple edge is the edge that the adversary wishes to obtain the $w(q)$ from.

If the adversary aims to map an obtained $\vec{\Delta}$ pair to an edge q, it must first determine the boundaries of the problem. For this discussion, we start with a problem defined in Subfigure 4.10a such that the adversary, also known as Eve, knows that the SUCCESS message with $T_c = \{C_1, 8\}$

must have originated from either Alice or Claire. We then later extend this case to show what happens if the graph topology becomes more complex.

The probability for the adversary to successfully map a received $(t_c, \vec{\Delta}_m)$ pair to an edge q between two honest nodes depends on the set of possible paths between the two honest nodes, which we define as P_h. This is because every honest node i that sends a SUCCESS message to j on $q_{j,i}$ changes the received $\vec{\Delta}_m$ using the `splitEqually` procedure to ensure that the $\sum_{(t_c,\vec{\Delta}_m)\in T_c} \vec{\Delta}_m \leq w(q_{j,i})$. If there is only one $p_h \in P_h$, which is the case in Subfigure 4.10a, and n represents the $\vec{\Delta}_m$ received by u in a SUCCESS message, we can define the information the adversary knows about $w(q)$ for $q \in p_h$ as follows:

1. $n < w(q_{u,i}) \rightarrow \exists q \in p_h;\ w(q) = n$
2. $n = w(q_{u,i}) \rightarrow \forall q \in p_h;\ w(q) \geq w(q_{u,i})$

In Subfigure 4.10a, we show the case of $p_h = \{q_{\text{Alice,Claire}}\}$. We assume in our example that the adversary knows the topology of the graph R such that there is no participation anonymity for any honest nodes. In the figure, Eve receives a SUCCESS message from Alice with $T_c = \{(C_1, 8)\}$ and wants to know $w(q_{\text{Alice,Claire}})$. To achieve this, Eve compares 8 to $w(q_{\text{Eve,Alice}})$ as she knows $w(q_{\text{Eve,Alice}})$ because Eve is one of the owners of the channel $e_{\text{Eve,Alice}}$. If case 1) occurs, i.e. $8 < w(q_{\text{Eve,Alice}})$, Eve knows that it must then be that $8 = w(q_{\text{Alice,Claire}})$ because $|p_h| = 1$, allowing Eve to break the channel balance privacy of channel $e_{\text{Alice,Claire}}$. If case 2) occurs, i.e. $8 = w(q_{\text{Eve,Alice}})$, Eve knows that it must be that $w(q_{\text{Alice,Claire}}) \geq 8$. Things change if we set p_h to be of arbitrary size g. In that case, if Eve receives $T_c = \{(C_1, 8)\}$ and gets case 1), Eve can no longer say with certainty which $q \in p_h$ the 8 belongs to and only knows it must be one of $q \in p_h$. Nothing changes for case 2) as with $p_h = g$ it still holds that all $q \in p_h$ must have a $w(q) \geq 8$.

In Subfigure 4.10b, we cannot make the assumption that there is only one $p_h \in P_h$ and we also have a cycle $p_{\text{cycle}} = \{q_{\text{Bob,Peter}}, q_{\text{Peter,Claire}}, q_{\text{Claire,Bob}}\}$. This invalidates case 1) because if $\exists p_h \in P_h$ for which it holds that $|p_h \cap p_{\text{cycle}}| \neq 0$, it must mean that there is an edge $q \in p_h \cap p_{\text{cycle}}$ on which both paths have to share the rebalancing objective $w(q)$. If then Eve receives a $T_c = (C_1, n)$ that travelled along p_h, it can happen that $\forall q \in p_h; n < w(q)$ as some of the $w(q)$ has already been used by a $(t_c, \vec{\Delta})$ travelling around p_{cycle}. In our example, this can happen if the $S(C_1, 8)$ message travels on the path $p_y = \{q_{\text{Peter,Claire}}, q_{\text{Bob,Peter}}, q_{\text{Alice,Bob}}, q_{\text{Eve,Alice}}\}$ for which it holds that $p_y \cap p_{\text{cycle}} = \{q_{\text{Bob,Peter}}, q_{\text{Peter,Claire}}\}$. We therefore have to change our earlier information definition to the following, where the adversary can only infer that:

$$n \leq w(q_{u,i}) \rightarrow \exists p_h \in P_h, \forall q \in p_h;\ w(q) \geq n \tag{4.4}$$

We can then define the probability that the adversary can determine $w(q)$ using n as follows:

$$\Pr[\text{Adversary obtains } w(q) \text{ using } n] = \frac{1}{|P_h|} \cdot \frac{1}{2^z - n} \tag{4.5}$$

In Equation 4.5, z equals the bit size of $w(q)$. In Lightning, 64-bit unsigned integers are used to define the channel balances ([25], BOLT #2). An implicit assumption done in Equation 4.5 is that all the rebalancing objectives in the graph are normally distributed, which implies that all channel balances in a PCN are normally distributed and this is not the case according to [49]. We, therefore, expect for the probability to be higher than implied in Equation 4.5 depending on the distribution of the channel balances.

Finally, in Subfigure 4.10c, Eve has access to more information than she did in Subfigure 4.10b. Using her extra edges, she can invalidate some of the possible paths in P_h, increasing her chances of guessing $w(q)$. For Eve, a good strategy is therefore being highly connected such that she can reduce the number of potential paths as much as possible, in a similar way as we discussed concerning participation anonymity.

Nevertheless, the probability of Eve determining the channel balance is not negligible and therefore, according to Definition 6, our protocol does not provide channel balance privacy. If we assume participation anonymity for all honest nodes, Eve has to construct P_h based on G instead of R. This increases the number of options for Eve to guess, even if G and R are equal in

topology as then Eve would still have to guess the direction of the edges in R. However, increasing the number of options still results in a non-negligible probability that Eve can break the channel balance privacy.

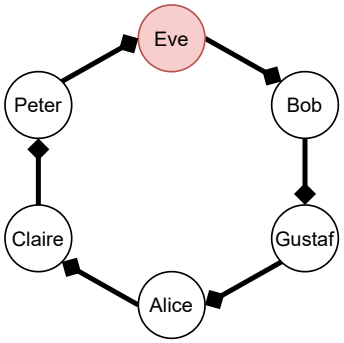


Figure 4.11: In this rebalancing graph there exist only five possible cycles with Eve as an intermediary

4.8.6 Path privacy

In Definition 7, we define $\texttt{pay}_{\text{condComp}}(s, r, x, p)$ to have path privacy if the probability for the adversary to determine p is less than or equal to $1/\left|\bigcap_{i \in C_p} P_i\right|$ and only if the sender is honest. Here, C_p is the set of corrupted nodes in p and P_i represents all possible paths going through node i. Less formally we say that the adversary can only break path privacy if *it knows more* about the path than what it can already infer from the topology of the graph and its corrupted nodes. If we take a look at Figure 4.11 for example, Eve can infer from the graph topology that there are only five possible transaction paths with her as an intermediary as all transactions in the transaction generation protocol are cyclic. Using our definition, this means that our protocol only provides path privacy if the probability that Eve guesses the path correctly is less than or equal to 1/5. If we add an extra directed edge from Bob to Claire, more cycles are possible decreasing the probability for Eve to guess the path correctly.

Seeing how path privacy is dependent on graph topology allows us to define a relationship between it and participation anonymity. If we assume participation anonymity, it means that the adversary has limited knowledge of the rebalancing graph R outside of the neighbours of its corrupted nodes. This would mean that, in Figure 4.11, Eve does not know the topology of R between Peter and Bob and it has to construct P_i based on G instead of R. This increases the number of options for Eve to guess, even if G and R are equal in topology as Eve would still have to guess the direction of the edges in R. However, as we have seen, our protocol does not always achieve participation anonymity.

Therefore, besides the additional privacy possibly provided by participation anonymity, we also provide path privacy with our design by having none of the parties[12] involved in a transaction know the full path. The cycle owner, who is the sender s and receiver r in our protocol, only knows on which edge a cycle transaction with tag t_c departs on and on which edge it arrives. Every intermediary u along the way only knows whom it received the transaction with tag t_c from and to which neighbour it needs to go. This limits the exposure of the path of the transaction to the bare minimum required for the successful execution of the transaction. Even if the adversary corrupts some of the intermediaries, it would gain little information about the transaction path. We, therefore, conclude that we achieve path privacy according to Definition 7 with and without participation anonymity.

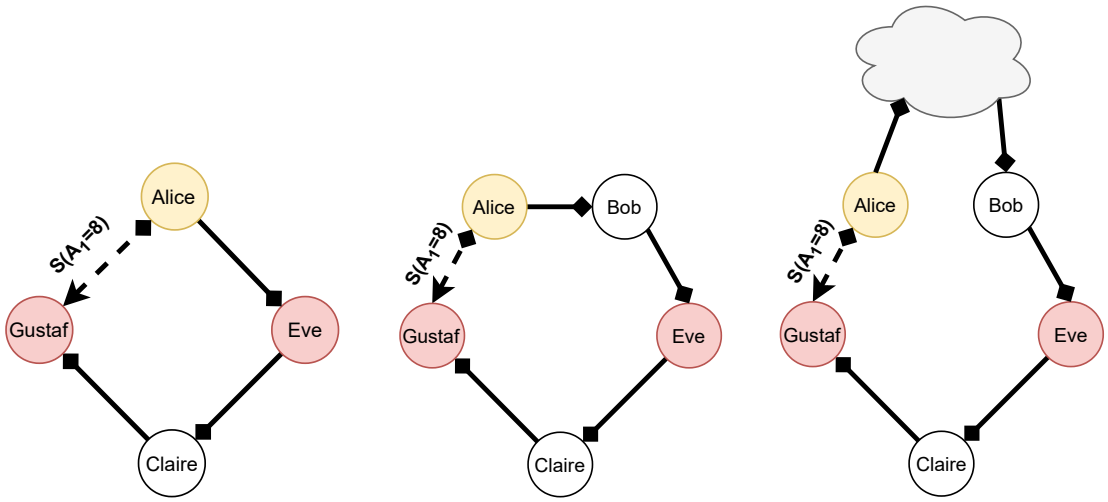
[12] i.e. sender, receiver and intermediaries

4.8.7 Value privacy

In Definition 8, we define $\texttt{pay}_{\text{condComp}}(s, r, x, p)$ to have value privacy if the adversary can determine x with negligible probability. Here, s is an honest sender, r an honest receiver, x the amount to be payed and p a path consisting of *only* honest intermediaries.

We can define a strategy for the adversary to increase its chances of breaking the value privacy if we assume that the adversary knows s, r and p. Knowing s, r and p, the adversary can do a $\texttt{pay}_{\text{condComp}}$ through a path $p' \subseteq p$ with amount x'. If the transaction fails, the adversary then knows that $x < x'$. The adversary can then try again and again, each time lowering x' until the transaction succeeds, which gives it an upper bound for x.

If we assume path privacy, then in our transaction generation protocol there is no way for the adversary to break the value privacy as knowledge about x in the form of $\vec{\Delta}_{\text{EX}}$ and $\vec{\Delta}_m$ is only transferred along p, which implies that the adversary can only obtain information about $\vec{\Delta}_{\text{EX}}$ and $\vec{\Delta}_m$ if it is part of the path p. As the adversary is not honest, this can never happen as according to our definition, value privacy only applies if p contains only honest intermediaries.



(a) The adversary can determine that Alice is the cycle owner as it receives a SUCCESS at Gustaf containing a tag that it has not seen at Eve

(b) The adversary now has a 50 % chance that Alice is the cycle owner as the tag must have been created by Alice or Bob

(c) If all honest nodes have participation anonymity, the adversary does not know outside of their neighbours who is participating. In that case the adversary does not know which nodes besides Alice or Bob might have created the tag.

Figure 4.12: Example showing the ability of the adversary to determine the cycle owner under different circumstances. Red represents the adversary, yellow represents the cycle owner and the grey cloud represents the nodes unknown to the adversary.

4.8.8 Sender/receiver privacy

In Definition 9, we define $\texttt{pay}_{\text{condComp}}(s, r, x, p)$ to have sender/receiver privacy if the probability for the adversary to determine s/r is larger than $1/|V - C|$ where C is the set of corrupted nodes. Contrary to the definition of value privacy, in sender/receiver privacy, the path p must hold *at least* one honest intermediary. To achieve sender/receiver privacy, $\texttt{pay}_{\text{condComp}}$ must first achieve path privacy as otherwise the adversary can directly infer s and r from p.

In our transaction generation protocol, the question of finding the sender/receiver from a $\texttt{pay}_{\text{condComp}}$ transaction is equivalent to finding the cycle owner u, given that all $\texttt{pay}_{\text{condComp}}$ operations done are cyclic and started by the cycle owner, which causes $s = r = u$. As we have seen in Table 4.5, none of the message types sent during the protocol contain contents that directly tell the adversary that u is the cycle owner. However, the presence of cycle tags t_c received by

the adversary that the adversary has never seen before can reveal the cycle owner. This problem is similar to the one we discussed for the channel balance privacy but now instead of determining $w(q)$ of an edge q between two honest nodes, the adversary tries to determine the creator of a SUCCESS message with a tag t_c.

In Subfigure 4.12a, the adversary can determine that Alice is the cycle owner as it receives a SUCCESS message at Gustaf with cycle tag A_1 that it has not seen at Eve, which implies that Alice must have created it as only cycle owners create cycle tags. In Subfigure 4.12b, the adversary cannot be sure that Alice created it as it is also possible that Bob created it and Alice forwarded it. In this case, the adversary has a 50 % chance that Alice is the cycle owner.

If we define the set of honest nodes between Gustaf and Eve as V^*, then $\Pr[u \in V^* \text{ is the cycle owner}] = 1/|V^*|$, which we illustrate in Subfigure 4.12c. Similar to our discussion about the channel balance privacy, it benefits the adversary to be highly connected as this decreases V^* which increases its probability of finding the cycle owner given a cycle tag t_c. Nevertheless, we normally expect $1/|V^*| > 1/|V - C|$ which, when strictly adhering to Definition 9, means we do not achieve sender/receiver privacy with our protocol. If we assume participation anonymity for all honest nodes, the adversary only knows G and not R. This possibly increases $|V^*|$ as the adversary must now consider all nodes between the two nodes and not only the ones that are participating.

4.8.9 Relationship anonymity

In Definition 10, we define a $\texttt{pay}_{\text{condComp}}$ operation to have relationship anonymity if, for two simultaneous $\texttt{pay}_{\text{condComp}}(s, r, x, p)$ operations between sender/receiver pairs (s_1, r_1) and (s_2, r_2) with the same amount x and the same path p holding i intermediaries of which at least one is honest, an adversary can determine if s_1 pays r_1 or s_2 pays r_2 with a probability less than $1/2$.

According to [29, 30], $\texttt{pay}_{\text{condComp}}$ operations in Lightning do not provide relationship anonymity. This is because Lightning sets the conditional transaction setup information φ_{setup} to be equal to $H(\alpha)$ for every $\texttt{pay}_{\text{cond}}$ operation that is part of $\texttt{pay}_{\text{condComp}}$. For the execution information φ_{exec}, Lightning does the same but then with $\varphi_{\text{exec}} = \alpha$. The authors show that this allows the adversary to easily distinguish the two $\texttt{pay}_{\text{condComp}}$ operations because each $\texttt{pay}_{\text{cond}}$ and $\texttt{pay}_{\text{exec}}$ operation can be tagged to one of the two $\texttt{pay}_{\text{condComp}}$ as all the $\texttt{pay}_{\text{cond}}$ and $\texttt{pay}_{\text{exec}}$ that are part of a $\texttt{pay}_{\text{condComp}}$ operation share the same φ_{setup} and φ_{exec}.

Even if we assume that the underlying PCN provides a $\texttt{pay}_{\text{condComp}}$ function that provides relationship anonymity, we can show that our protocol breaks it too. This is because each $\texttt{pay}_{\text{condComp}}$ is associated with a cycle tag t_c that is known to all nodes in p. Similarly to [29, 30] where the authors show that reusing the same φ_{setup} and φ_{exec} for a $\texttt{pay}_{\text{condComp}}$ operation breaks relationship anonymity, the adversary can use the t_c to distinguish the two simultaneous $\texttt{pay}_{\text{condComp}}$ operations. We can therefore conclude that with our current design no relationship anonymity is provided.

For future work, we consider an alteration on the design that uses the Anonymous Multi-Hop Locks for transaction setup and execution, as introduced in [29, 30]. The authors show that their scheme provides relationship anonymity. However, for the scheme to function, the authors require anonymous channels between every participant and also require the sender to know the full transaction path p. To satisfy both conditions, alterations to the design are required and more trust needs to be put into the cycle owners (who act as sender/receivers), who currently are not trusted. Initial ideas for such a design would involve the cycle owner providing a public key alongside the first SUCCESS message, which would be used by intermediate nodes to encrypt their own public keys that, when received by the cycle owner, allow the cycle owner to construct a path and encode it using an onion-like scheme. This would remove the need for t_c tags during the COMMIT and EXEC phase which, alongside using Anonymous Multi-Hop Locks, allows the protocol to provide relationship anonymity.

Chapter 5

Evaluation

In this chapter, we evaluate our protocols and compare our transaction generation protocol to Revive [18]. Our evaluation consists of a message and time complexity analysis of our protocols and a performance evaluation. In the performance evaluation, we compare our transaction generation protocol to Revive and run simulations on both our protocols to see how their parameters affect their results.

5.1 Message complexity

In this section, we analyse the message complexity of both our protocols. We use M to denote the set of messages sent during one execution of a protocol.

5.1.1 Participant discovery

To analyse the message complexity of the participation discovery protocol, we use the fact that our discovery protocol communicates outside and inside of a tree T for which it holds that $T \subseteq G$. T is constructed based on two rules: 1) the sender v of the first INVITE received by a node u is stored by u as its parent or 2) if u receives an INVITE from v and it has not forwarded INVITE's itself, it sets v as its parent. u informs v of its parental status by setting the *isChild* boolean to true in the ACCEPT message u sends to v. This creates a problem if $h_c = 1$, as in that case u must immediately reply to the INVITE with an ACCEPT(*isChild*=true) as it might be a leaf in T who do not receive any further messages beside a FINISH message. If, at a later point, u then receives another INVITE from j with $h_c > 1$, u forwards the INVITE and eventually sends an ACCEPT(*isChild*=true) to j. If u does not update its parent because of 2), u would send *another* ACCEPT to v, which is unexpected to v. However, because of 2), v and j both think u is their child which technically makes T a cyclic graph. Although easily resolved by u informing v of the change in ancestry, we chose not to include such a mechanism as the only consequence of the double ancestry is that u receives one extra FINISH message from v. In the following discussion, we ignore this extra message as it has no consequence on the final message complexity of the participant discovery protocol.

We denote the edges inside the tree as E_T and the edges outside the tree as $E_{\neg T}$. An edge $e_T \in E_T$ achieves the maximum number of messages if an INVITE message is first sent over e_T from a parent to one of their children, after which the child responds with an ACCEPT and in reply, the parent sends a FINISH message, totalling three messages. An edge $e_{\neg T} \in E_{\neg T}$ sees the maximum number of messages if both vertices send an INVITE to each other and then reply with an ACCEPT or DENY, totalling four messages.

We consider the worst case message complexity to occur when all nodes $v \in V$ wish to participate with all their edges. If T is the tree from the protocol's initiator[1] to all other nodes in V, it must be that $h_c > h(T)$ and the max number of invites $I_m >= d_{\max}$, where h_c is the hop count, $h()$ is a function that returns the height of T and $d_{\max}$ is the largest degree $\forall v \in V$. If both

[1] The node who started the protocol

conditions are achieved, all nodes $v \in V$ receive an INVITE and forward it to all other nodes on all their edges with the exception of their parent. This allows us to define the message complexity as in Equation 5.1, knowing that if all nodes in V participate and T is a tree, $E_T = |V| - 1$ and $E_{\neg T} = E - E_T$.

$$
\begin{aligned}
|M| &= 4|E_{\neg T}| + 3|E_T| && \text{(5.1a)}\\
&= 4(|E| - (|V| - 1)) + 3(|V| - 1) && \text{(5.1b)}\\
&= 4|E| - |V| + 1 && \text{(5.1c)}\\
&= \mathcal{O}(|E|) && \text{(5.1d)}
\end{aligned}
$$

We can remove $|V|$ in Equation 5.1d as in a connected graph, $|E| \geq |V| + 1$.

5.1.2 Transaction generation

We analyse the message complexity per round for one execution of the transaction generation protocol. This can be done because the number of rounds is well-defined by the protocol's parameters. The transaction generation protocol operates on a rebalancing graph $R = (V, Q)$, which is a graph where all vertices and edges are participating in the protocol and all edges are directed based on the rebalancing objective. It holds that $R \subseteq G$.

For our worst-case scenario, we consider an R where every directed edge $q \in Q$ has a $|w(q)| \gg 0$, such that after one round of rebalancing, $\forall q \in Q; w(q) > 0$. This is to ensure that all directed edges always participate in the rebalancing as the protocol has special exclusions for edges where $w(q) = 0$. Our scenario also requires that all nodes $u \in V$ have at least one incoming and one outgoing edge, i.e. $\forall v \in V; |Q_i^u| > 0 \wedge |Q_o^u| > 0$. This ensures that all nodes receive a REQUEST each round, which causes them to send more messages than when a node has only outgoing edges. If a node has only outgoing edges, it does nothing[2] until receiving a NEXT_ROUND message. If a node has only incoming edges and receives a REQUEST, it replies with a FAIL message as it knows it cannot be part of a cycle. Our final assumption to construct the worst-case scenario involves the direction of edges in Q. We assume that a direction is obtained that maximises the number of cycles in R, as this affects the number of UPDATE and EXEC messages sent during the protocol.

Theoretical derivation

Using the assumptions of the previous section, we can derive the number of REQUEST, SUCCESS and COMMIT messages during the protocol. As every node has an incoming edge on which it receives a REQUEST and an outgoing edge on which it can forward it, it holds that every $q \in Q$ sees one REQUEST message each round, totalling $N_{\text{REQUEST}} = |Q|$ messages.

A node u always sends a SUCCESS in response to a REQUEST as, under our assumptions, u always has an outgoing edge. The SUCCESS is either because u detected one of its owned cycle detection tags t_r in the received REQUEST or if it has received a SUCCESS on all its outgoing edges as a reply to all its forwarded REQUESTs. Similar to the REQUESTs, this means that every $q \in Q$ must see one SUCCESS message each round, totalling $N_{\text{SUCCESS}} = |Q|$ messages. The consequence of every edge seeing a SUCCESS message each round is that they all also see a COMMIT message, as the protocol requires that a SUCCESS is always replied to with a COMMIT. This means that $N_{\text{COMMIT}} = |Q|$ as well. It is also relatively easy to define $N_{\text{NEXT\_ROUND}}$ as this mechanism of the protocol works similar to an α-synchronizer (See [1]) meaning that it has $\mathcal{O}(|Q|)$ message complexity. Although a node does not send NEXT_ROUND messages to the node it received the first NEXT_ROUND from, $N_{\text{NEXT\_ROUND}} \leq 2|Q|$ is a close approximation.

The number of UPDATE messages is more difficult to define than the previous message types, as UPDATEs are designed to be sent and forwarded repeatedly to allow nodes to detect cycles if a deadlock occurs. The number of UPDATE messages can vary wildly for each round of the protocol based on the message order and which node is chosen as the leader. An upper bound

[2] If the node is not the leader

for the total number of UPDATE messages sent during one round is $N_{\text{UPDATE}} \leq |Q| \cdot (|Q| - 1)$, which represents each edge seeing an update message for each other edge in R. We can lower the bound slightly as the leader never sends an UPDATE message because of Theorem 1. However, excluding the outgoing edges of the leader does not have an influence on the eventual total message complexity.

The final messages that we need to discuss are EXEC messages, which a node sends once for every committed cycle it is a part of. As discussed in Subsection 4.6.2, the number of committed cycles found is dependent on the direction of the edges q in R and the order in which messages arrive. If we define C to be the set of committed cycles and if we assume that every node is part of every committed cycle, $N_{\text{EXEC}} \leq |V| \cdot |C|$. As the latter assumption does not always hold, we consider this to be an upper bound on the number of EXEC messages. As we have now obtained exactly or bounded message counts for all message types, we can define the message complexity of the transaction generation protocol as in Equation 5.2.

$$
\begin{aligned}
|M_{\text{r}}| &= N_{\text{REQUEST}} + N_{\text{SUCCESS}} + N_{\text{COMMIT}} + N_{\text{UPDATE}} + N_{\text{EXEC}} + N_{\text{NEXT\_ROUND}} && \text{(5.2a)} \\
&= |Q| + |Q| + |Q| + |Q| \cdot (|Q| - 1) + |V| \cdot |C| + 2|Q| && \text{(5.2b)} \\
&= \mathcal{O}(|Q|^2 + |V| \cdot |C|) && \text{(5.2c)}
\end{aligned}
$$

From Equation 5.2, we can deduce that the largest contributing factors to the message complexity of a single round are the UPDATE and EXEC messages. If a future design iteration of the protocol shows a need to reduce its message complexity, then the mechanisms behind the UPDATE and EXEC messages should be the first points of consideration. We also outline it as a topic for future work to find a relationship between C and Q, which might allow for an exact definition of N_{EXEC}.

As Equation 5.2 is only applicable to a single round, we can modify the message complexity such that it is applicable to one whole execution of the transaction generation protocol. The message complexity then becomes $\mathcal{O}\left(\mathit{maxRounds} \cdot (|Q|^2 + |V| \cdot |C|)\right)$ or, if we relate $\mathit{maxRounds}$ to $|V|$ through ρ such that $\mathit{maxRounds} = |V| \cdot \rho$, the message complexity becomes $\mathcal{O}\left(\rho \cdot (|V| \cdot |Q|^2 + |V|^2 \cdot |C|)\right)$.

In practice, we expect the number of messages to decrease for every round that the protocol advances. This is because if in every round some rebalancing objectives $\vec{\Delta}$ are met, it is logical to expect that in later rounds there are more edges q with a zero $\vec{\Delta}$ which causes q to be excluded from the participation of that round. This has the effect of splitting R into several components as the protocol progresses.

Experimental evaluation

To illustrate the dependency of certain message types on message delays and the order of message arrival, we ran 1000 simulations of the transaction generation protocol with different seeds and participants on $G_{\text{Lightning}}$ as defined in Subsection 5.3.2 and present the result in Table 5.1. We restricted the protocol to one round, but by changing the seed we changed both the leader of the round and the message delays, which also changes the ordering of messages. Table 5.1 shows that the number of UPDATE and NEXT_ROUND messages vary significantly between simulations while the REQUEST, SUCCESS, COMMIT and EXEC messages vary much less. This confirms our expectation that the number of UPDATE messages influences the total number of messages more than other message types. The results also show that many more NEXT_ROUND messages are sent than EXEC messages. This might indicate that $|V| \cdot |C| < 2|Q|$ but this requires a mathematical equation for the theoretical number of EXEC messages to confirm.

5.2 Time complexity

In our analysis of the time complexity of our protocols, we assume that the computation done by each node during the execution of one of our protocols is negligible. We consider this a valid assumption as our designs feature loops and data accesses that are at most $\mathcal{O}(|Q|)$. A much more

Table 5.1: Statistics on the number of messages for one round of the transaction generation protocol with a varying number of participants in $G_{\text{Lightning}}$ as defined in Subsection 5.3.2. Statistics were obtained using 1000 simulations with varying seeds.

Type	Maximum	Minimum	Mean	Standard deviation
REQUEST	94	1	12.84	17.00
UPDATE	233	1	25.30	38.99
SUCCESS	57	1	7.51	10.13
COMMIT	57	1	7.51	10.13
EXEC	50	2	9.08	9.64
NEXT_ROUND	182	5	60.97	50.08

influential factor in the time complexity of our protocols is the number of messages sent during each execution. Each message send arrives with a random communication delay. If we make the assumption that all messages are sequentially ordered in time, an upper bound for the time complexity of both protocols is $\mathcal{O}(|M|)$, where M is the set of messages during one execution of the protocol. Based on our discussion of the message complexity, this would imply that the time complexity of the protocol is always the same as the message complexity. However, as shown in this section, the parallelism achieved by our protocols allows the time complexity to be lower than this upper bound.

5.2.1 Participant discovery

In our discussion about the message complexity of the participant discovery protocol, we showed how we can model the sending of messages in our discovery protocol with a tree T. We use the same construction and assumptions as in our discussion about the protocol's time complexity. These assumptions are: all nodes $v \in V$ wish to participate with all their edges, $h_c > h(T)$ and $I_m >= d_{\max}$. A new assumption we make is that each message is equally delayed by one unit of time. For our worst-case scenario, one unit of time is equivalent to the maximum communication delay. We use a variable t to count how many units of time have passed.

At the start of the participation discovery protocol when $t = 0$, the initiator sends an INVITE to all its neighbours. At $t = 1$, the INVITEs arrive at the neighbours who then forward the INVITE to all their neighbours. The neighbours then receive it at $t = 2$, and this process repeats until all nodes have received an INVITE. As we know from our earlier discussion, such communication can be modelled as a tree T with the initiator as the root and the outermost nodes as the leaves. For each unit of time that passes, each level of the tree sends an INVITE to all their neighbours along E_T and $E_{\neg T}$. The last invite is therefore received at $t = h(T) + 1$ by a neighbour of the leaf u at the lowest level of the tree. The neighbour, together with all the other neighbours of u that received an INVITE, replies with an ACCEPT which u receives at $t = h(T) + 2$. This ACCEPT then propagates back along T to the initiator who receives the final ACCEPT at $t = 2h(T) + 2$. As the last step of the protocol, the initiator sends a FINISH message along T which is received by u at $t = 3h(T) + 2$, marking the termination of the complete protocol. The time complexity of the participation discovery protocol is, therefore, $\mathcal{O}(h(T))$. Note that the worst-case scenario only occurs when the initiator is chosen such that $h(T)$ is maximised.

5.2.2 Transaction generation

We analyse the time complexity of the transaction generation protocol by considering one round and analysing the maximum time taken for each step in that round. We use the same assumptions as in our discussion about the message complexity: $\forall q \in Q; |w(q)| \gg 0$ and $\forall v \in V; |Q_i^u| > 0 \wedge |Q_o^u| > 0$. We add the assumption that each message is equally delayed with one unit of time which, similarly as with the participant discovery, can be thought of as reflecting the maximum communication delay. We also use t with various subscripts to keep track of units passed.

Similarly, as with the discussion about the time complexity of the participant discovery, we can use a tree T_R which contains every node in R, to analyse how the messages spread throughout R. Each node, apart from the leader, receives their first REQUEST message on one incoming edge. This forms a tree with the round leader l as the root, who sends a REQUEST message at $t_{\text{REQ}} = 0$. At $t_{\text{REQ}} = h(T_R) + 1$, the REQUESTs sent by the nodes at the lowest level of T_R arrive at the nodes opposite of their outgoing edges.

Once the first REQUEST message arrives at a node u and until it receives a SUCCESS on all outgoing channels, u is allowed to send UPDATE messages on all its outgoing edges if it learns of any new cycle detection tags t_r. In our discussion of the message complexity, we were only able to define an upper bound of $N_{\text{UPDATE}} \leq |Q| \cdot (|Q| - 1)$ for the number of UPDATE messages each round. Barring an exact definition for the number of messages and the UPDATE mechanism, we also assume an upper bound of $t_{\text{UPD}} \leq N_{\text{UPDATE}}$ for when the last UPDATE message is received by a node.

Using T_R, we can also denote the time taken for the last SUCCESS message to arrive at the leader l. This SUCCESS message must be sent after the last UPDATE message as receiving the last SUCCESS message indicates all deadlocks are resolved, which is the goal of sending UPDATE messages. If we then assume that the last UPDATE message is received by a node at the lowest level of T_R, it takes $t_{\text{SUC}} = h(T_R)$ for l to receive the last SUCCESS message after the last UPDATE message has been received by a node.

For the last commit message to arrive we can take a similar approach as with the REQUEST messages, as they also follow the edges in T_R. From this we can deduce that $t_{\text{COM}} = t_{\text{REQ}} = h(T_R) + 1$. The time for the last EXEC message to arrive after the last COMMIT message was sent depends on the length L_C of the longest committed cycle in R. If we assume a worst-case scenario where the owner v of this longest cycle is on the lowest level of T_R, then when v receives the last COMMIT message it can start sending the EXEC message for this longest cycle. The time for that EXEC message to propagate is equal to the number of edges in this cycle i.e. L_C. The time for the last EXEC message to arrive is therefore $t_{\text{EXEC}} = L_C$.

Finally, the NEXT_ROUND messages follow a similar route as the REQUEST and COMMIT messages from the route leader to the other nodes in R along T_R. This means that the last NEXT_ROUND message also arrives after $t_{\text{NR}} = t_{\text{REQ}} = h(T_R) + 1$.

$$
\begin{aligned}
t_{\text{max,round}} &= t_{\text{REQ}} + t_{\text{UPD}} + t_{\text{SUC}} + t_{\text{COM}} + t_{\text{EXEC}} + t_{\text{NR}} && \text{(5.3a)} \\
&= h(T_R) + 1 + N_{\text{UPDATE}} + h(T_R) + h(T_R) + 1 + L_C + h(T_R) + 1 && \text{(5.3b)} \\
&= 4h(T_R) + N_{\text{UPDATE}} + L_C + 3 && \text{(5.3c)} \\
&= \mathcal{O}(h(T_R) + N_{\text{UPDATE}} + L_C) && \text{(5.3d)}
\end{aligned}
$$

Combining our findings, we obtain a time complexity as presented in Equation 5.3. In this time complexity, we expect N_{UPDATE} to be much larger than $h(T_R)$ or L_C, given that multiple UPDATE messages can be sent over the same edges in one round. In future work, a more detailed analysis of the time complexities involved in the UPDATE phase might allow one to achieve a theoretical upper bound for the UPDATE phase time complexity.

5.3 Performance analysis

In this section, we experimentally evaluate the performance of our protocols by running multiple simulations using our proof-of-concept. We first discuss the construction of our proof-of-concept and the topologies of the graphs we run our simulations on, after which we evaluate both our protocols using the proof-of-concept and the graph topologies.

5.3.1 Proof-of-concept

For the experimental evaluation of our protocols, we implemented a proof-of-concept in Kotlin[3] for which we provide the code on Github[4]. The proof-of-concept holds implementations for the

[3] https://kotlinlang.org/

[4] https://github.com/roemba/thesis-rebalancing

participant discovery protocol, the transaction generation protocol and Revive. As almost all these protocols operate on a PCN, we included in the proof-of-concept a PCN simulator that is capable of message passing between nodes and the payment functionality as described in Subsection 4.1.1.

The simulator can run on any graph G and is a *discrete event* simulator, which is a kind of simulator where, compared to a continuous simulator, next-event time progression occurs [12]. This property means that its internal clock jumps to the time of the next event, which it then executes. In our simulation, an event can be the arrival of a message or the start of a protocol. Jumping to the time of the next event has the benefit of speeding up the simulation, as the simulator does not have to idle for the time between two events. In this way, we can add a delay to message arrival events. These delays *will* appear in the clock of the simulator, but do not require the computer to simulate the delay.

When a node u sends a message m to v on channel $e_{u,v}$, we schedule the arrival of message m at v to happen at $t_s = \max(t_s, t_s^{e,u}) + \texttt{delay}(1, 200)$. Here, $t_s^{e,u}$ represents the last time a message from u arrived at v on edge $e_{u,v}$ and `delay` is a function that returns a random millisecond delay from an exponential distribution with a minimum of 1 ms and a 1 % chance of a delay of 200 ms. Using $t_s^{e,u}$ allows us to simulate channel congestion and guarantee message ordering over e in each direction. Using random latencies has the effect of scrambling the processing order of messages that each node receives during a run. In reality, latencies can vary significantly because of several factors such as the geographic distance between two nodes. However, in our simulation, we chose to exclude these factors as we are only interested in comparing our protocols runtime to Revive and not evaluating its runtime exactly.

As the simulator and some of the protocols require randomness, we seed all the random sources from a common seed that changes for each run. This allows us to average out the effect of random factors, such as the ordering of messages and leader selection, on the results of our protocols.

The implementation of our transaction generation protocol and Revive requires that each node knows which other nodes are participating in the protocol. It is for this reason that we designed the participant discovery protocol. In our implementation, both our transaction generation protocol and Revive use our participant discovery protocol. The latter is needed because the authors of Revive do not provide an implementation for such a protocol and assume the participants are already known. Another benefit of this approach is that it allows for a fair comparison of both transaction generation protocols. Both protocols also use the same `negotiateObjective` interface as we and the authors of Revive share the same requirement that the owners of a channel have a way of determining their rebalancing objective if requested.

5.3.2 Graph topologies

As we expect the results of our protocols to be influenced by the topology of a graph G, we prepared three different graphs for the simulator to run on. Each graph has its own topology and unique channel balances, which we obtain in a variety of different ways. Note that the exact value of the channel balances is not important for our experiments as we are not interested in the absolute number of rebalancing demands met or the absolute transaction success ratio but instead are interested in the differences *between* those numbers for varying parameters and scenarios.

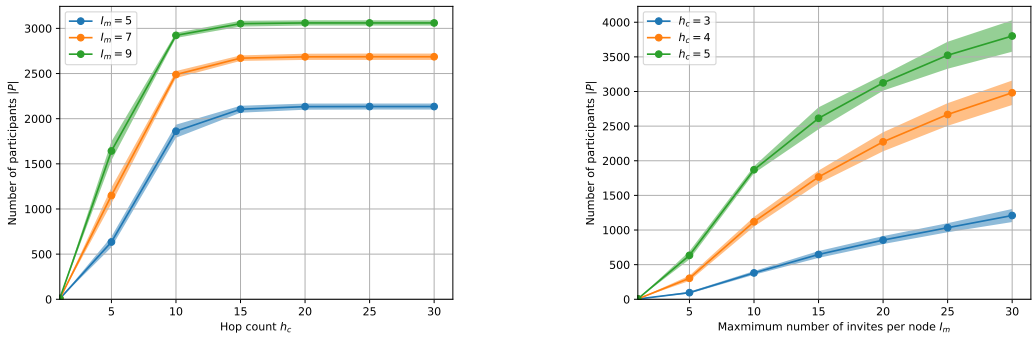
The first graph we simulate, G_{Design}, is the example used in the Design chapter in Figure 4.4. We include this graph as it contains many cycles and multiple edges between the same pair of nodes, and we expect readers of the Design chapter to be familiar with its layout. The rebalancing graph R_{Design}, based on G_{Design}, uses the same rebalancing demands as defined in Subfigure 4.4a.

The second graph G_{Complete} is a complete graph K_8. In a complete graph, many cycles are possible depending on the rebalancing objectives of each edge. We generated the channel balances *once* from a uniform distribution between $[0, 100)$.

Our final graph $G_{\text{Lightning}}$ consists of the *processed* topology of the Lightning network as scraped by explorer.acinq.co on the 5th of May 2021. The topology is provided as two JSON files of which one contains nodes and the other contains edges. The 'processed' refers to the fact that we removed all edges that could not be matched to any pair of nodes and that, after doing a component analysis, we kept the largest component consisting of 8991 nodes. As a result, the final $G_{\text{Lightning}}$ consists of 8991 nodes and 38 524 edges from the original scraped data that contained 9044 nodes and 38 697 edges. As in Lightning the channel balances are not publicly announced

and therefore not publicly available, we randomly generate the balances based on an exponential distribution to mimic the phenomenon that most channels in Lightning are of low capacity with few being of high capacity [10, 11].

5.3.3 Participant discovery



(a) (b)

Figure 5.1: The effect of different hop counts h_c and maximum number of invites per node I_m on the number of participants $|P|$ obtained by the participation discovery protocol. Results are from ten simulations on $G_{\text{Lightning}}$ with different seeds. The solid lines represent the mean of the simulations and the confidence bounds represent one standard deviation from the mean.

For the evaluation of our participant discovery protocol, we ran ten simulations with different seeds with varying h_c and I_m on $G_{\text{Lightning}}$ to study the number of participants $|P|$ achieved. We present the results in Figure 5.1. In all simulations, the same starting node was chosen to initiate one instance of our discovery protocol using the provided settings. This is necessary to ensure that the parameter under study is the only independent variable affecting $|P|$. If we were to change the starting node in every simulation, the degree of the starting node would also influence $|P|$, making the starting node an additional independent variable.

Looking at the obtained results in Subfigure 5.1a analysing the influence of the hop count h_c, we can clearly see that $|P|$ increases as h_c becomes larger. We can also see a steep increase in $|P|$ between the range of $5 \leq h_c < 15$ which levels off when $h_c \geq 15$. We suspect that when $h_c > 5$, the INVITE messages reach one or multiple hub nodes with many connections. Clearly after $h_c \geq 15$, no new hub nodes are reached as the curve levels off and $|P|$ stabilizes.

Subfigure 5.1b shows a different trend concerning the maximum number of invites per node I_m. As expected, we obtain a higher $|P|$ when I_m becomes larger. $|P|$ clearly increases almost linearly with I_m while flattening off for higher h_c when $I_m > 10$. We can also see that the larger I_m becomes, the larger the standard deviation. This phenomenon is more present when I_m is combined with a larger h_c such as 4 or 5, but we do not have a clear explanation as to why this is the case. Subfigure 5.1b also confirms that h_c has a larger effect on $|P|$ than I_m, which is in line with our analysis done in Subsection 4.5.1 where we obtained that $|P| = \mathcal{O}(I_m^{h_c})$.

5.3.4 Transaction generation

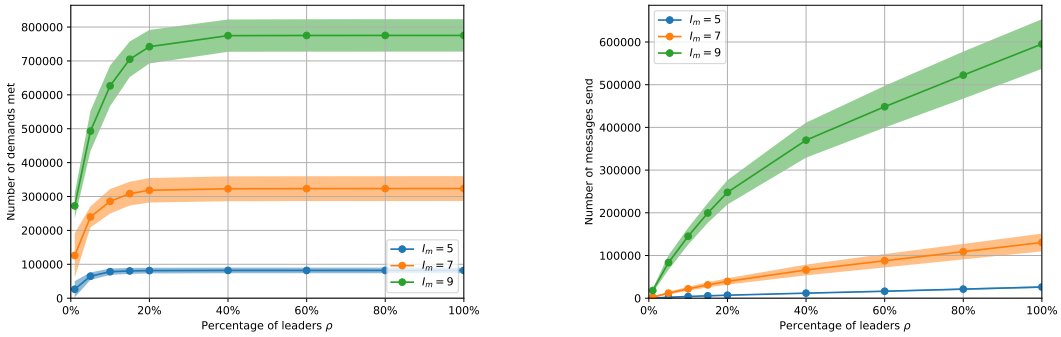
In this section, we evaluate our transaction generation protocol by experimenting with different values for ρ and comparing how it behaves compared to Revive in a static simulation and dynamic simulation.

With the exception of the dynamic simulation, we try to minimize the impact of the participation discovery protocol on the results of our comparisons. Based on our experiments with the participant discovery, we chose a $h_c = 3$ and an $I_m = 5$ for the ρ evaluation and a $h_c = 3$ and an $I_m = 10$ for the static simulation. According to Figure 5.1, all these values produce a $|P|$ with

only small deviations for varying seeds. This has the effect of minimizing the influence of $|P|$ on the obtained results of the transaction generation protocol and Revive.

Varying ρ

The only parameter of the transaction generation protocol is ρ, which we introduced in Subsection 4.6.2 to calculate the *maxRound* of the protocol based on $|P|$ using the formula $maxRound = |P| \cdot \rho$. Our goal with providing the formula is to define the number of rounds the transaction generation protocol should execute to obtain its best result possible. In this section, we investigate the relationship between ρ and *maxRound* to see how different ρ affect the result of the protocol for different $|P|$.



(a) Number of demands met

(b) Number of rebalancing messages send

Figure 5.2: The effect of different ρ on the number of demands met and the number of messages sent by the transaction generation protocol. Results are from ten simulations on $G_{\text{Lightning}}$ with different seeds. The simulations were run with $h_c = 3$ and varying I_m to obtain three different $|P|$. The solid lines represent the mean of the simulations and the confidence bounds represent one standard deviation from the mean.

For this evaluation, we use $G_{\text{Lightning}}$ where we keep the channel balances the same for varying seeds. This limits the influence of varying seeds to changing the message ordering and which nodes act as a leader. We ran ten simulations for each $I_m \in \{5, 7, 9\}$ while keeping $h_c = 3$. We vary I_m because we have no direct control over $|P|$, but can only influence it using I_m and h_c. We chose I_m to vary $|P|$ and not h_c as changing I_m allows for more granular control over $|P|$.

We measured the 'success' of the transaction generation protocol by comparing the demands *before* executing the protocol and *after*. The difference is the number of demands met by the protocol. We present the results of our simulations in Figure 5.2.

Looking at Subfigure 5.2a, it is clear that for all $|P|$ we tested, a $\rho \geq 40\,\%$ does not increase the number of demands met. This is logical considering that the more rounds the protocol executes, the higher the probability that it finds rebalancing cycles. After x rounds, no more rebalancing cycles are possible as some edges have $w(q) = 0$, which negates the benefits of executing additional rounds. We can clearly see this effect in Subfigure 5.2a. More interesting is that the start of this 'levelling off' appears at different ρ for different $|P|$. For $I_m = 5$, this point appears when $\rho = 20\,\%$ while for $I_m \in \{7, 9\}$, this point appears when $\rho = 40\,\%$[5]. This might indicate that the relationship between the optimal *maxRound* and $|P|$ is not linear but something closely resembling it, such as a slightly quadratic relationship.

When we combine Subfigure 5.2a with Subfigure 5.2b, a clear trade-off appears. Besides the expected increase in messages for higher $|P|$, it is clear that a larger ρ requires more messages to be sent. This intuitively makes sense as more rounds take place for larger ρ and each round requires additional messages. Another interesting aspect that appears comparing Subfigure 5.2a to Subfigure 5.2b involves the curve change in Subfigure 5.2b. In Subfigure 5.2b, the curve starts off

[5]For $I_m = 7$ this is hard to see on the graph because of the difference in scale for each I_m

non-linearly until the curve in Subfigure 5.2a levels. After the levelling, the curve in Subfigure 5.2b continues linearly. We believe this happens because rounds with rebalancing cycles present (the non-linear part of Subfigure 5.2a) behave differently than rounds without rebalancing cycles (the linear part of Subfigure 5.2a). If a round has rebalancing cycles, EXEC messages are sent for each cycle found. Once the number of rebalancing cycles starts to diminish, fewer EXEC messages are sent and edges where $w(q) = 0$ are excluded from participating in the round. This has the effect of reducing the number of deadlocks that can appear which in turn reduces the number of UPDATE messages. Once the protocol reaches a steady state with no new rebalancing cycles, the same number of messages is sent each round. From that point onwards, the number of messages sent during the protocol increases linearly for each extra round. This matches the results seen in Figure 5.2 but also our derived message complexity for each round, $\mathcal{O}(|Q|^2 + |V| \cdot |C|)$. As $|C|$ changes, the number of messages for each round changes too. Once $|C|$ is constant and zero, the same number of messages are sent each round.

Based on the results of both figures, we recommend that one should use $\rho = 20\,\%$ for small $|P|$, as this produces the best result for the minimum number of messages.

Static comparison

In this section, we evaluate the performance of our transaction generation protocol in a head-to-head comparison with Revive in a static setting with no transactions taking place. We evaluate both protocols on G_{Design}, G_{Complete} and $G_{\text{Lightning}}$. Both protocols run on the same set of participants and the same generated channel balances. Based on the results from previous experiments, we chose to use $h_c = 3$, $I_m = 10$ and $\rho = 20\,\%$ as the settings of the transaction generation protocol. We again ran ten simulations with varying seeds which shuffles the message ordering and the choice of leader. We present the results of these simulations in Figure 5.3.

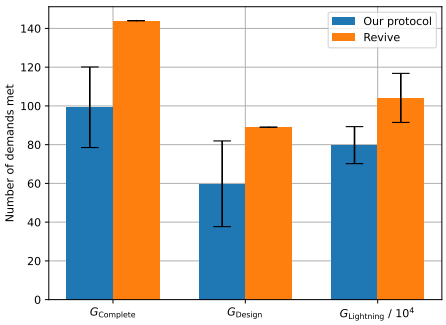
Looking at Figure 5.3, it is clear that our transaction generation protocol performs worse than Revive in the metrics of the experiment. For the same graph and number of participants, our protocol meets fewer demands, sends more messages and takes longer to execute than Revive. In designing the protocol, we heavily focused on privacy but not as much on performance. This focus on privacy is clearly visible in Figure 5.3. Nevertheless, in terms of the number of demands met, our protocol is always relatively close to Revive even though our protocol is very different in construction. It is also interesting to see that for small rebalancing graphs, the difference in messages and time between Revive and our protocol is small. This is positive as we expect a practical implementation of our protocol to be run on smaller $|P|$, as this lowers the risks of pre-emptive terminations due to failures in either the nodes or the communication layer.

Dynamic comparison

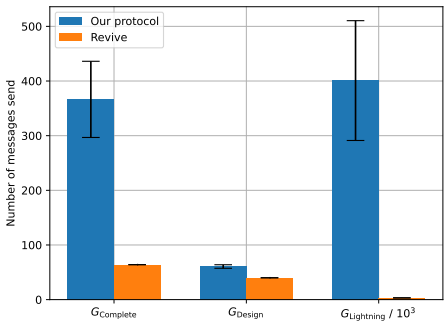
The final evaluation in this chapter is a dynamic simulation of both our transaction generation protocol and Revive. The dynamic simulation differs from the static simulation in that we are not interested in the number of demands met, the messages send or the time taken, but we are only interested in the ability of both protocols on the ability of the PCN to carry out transactions and the effect on the 'imbalance' of the PCN. For this reason, the dynamic simulation is only run on $G_{\text{Lightning}}$ where we simulate transactions taking place. We then trigger one of the two protocols to run if a node reaches a certain 'imbalance'.

We define the ability of the PCN G to carry out transactions as the *transaction success ratio*, which is the ratio of the total number of completed multi-hop transactions divided by the total number of multi-hop transactions. More accurately, as we do not retry failed transactions, this ratio represents the number of transactions that completed *on the first try* divided by the total number of multi-hop transactions. We do not retry transactions as this would make the success ratio dependent on the number of allowed retries, which is not a parameter we wish to study.

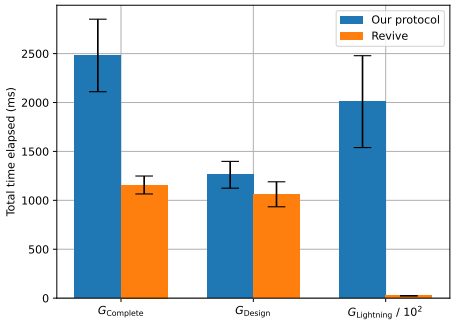
In a PCN, transactions do not complete and fail if there is 1) no path from sender to receiver or 2) one or more channels along the path do not have enough balance to cover the transaction amount. With our transaction generation protocol and Revive we introduce another failure reason, 3) if the channel is *locked* by (an instance of) a protocol. If a channel is *locked*, committed transactions using the channel can still execute but any new transactions are denied. If a rebal-



(a) Number of demands met



(b) Number of rebalancing messages send



(c) Total time elapsed

Figure 5.3: Comparison of our transaction generation protocol and Revive in a static simulation. The simulation was ran 10 times with $h_c = 3$, $I_m = 10$, $\rho = 0.5$ and different seeds on G_{Design}, G_{Complete} and $G_{\text{Lightning}}$. The bars represent the mean of the simulations and the error bars represent one standard deviation from the mean. The results of the simulations on $G_{\text{Lightning}}$ are scaled by powers of 10 to fit on the graph.

ancing protocol balances channels to reduce the cases where 2) happens, it must take care not to *lock* the involved channels for too long otherwise 3) becomes the primary reason for transaction failure. We included Requirement 2b to highlight this concurrency aspect of rebalancing protocol design, although concurrency is not a design feature of both our transaction generation protocol and Revive.

We define the *imbalance* of a node in the same way as [40], who use Equation 5.4 to calculate the Gini coefficient of one node u.

$$G_u = \frac{\sum_{i \in N(u)} \sum_{j \in N(u)} |\zeta_{u,i} - \zeta_{u,j}|}{2 \sum_{i \in N(u)} \sum_{j \in N(u)} \zeta_{u,j}} \tag{5.4}$$

In Equation 5.4, $\zeta_{u,v} = \frac{b(e_{u,v},u)}{c(e_{u,v})}$, where $b(e_{u,v}, u)$ is the channel balance of u in $e_{u,v}$ and $c(e_{u,v})$ the capacity of channel $e_{u,v}$. If $G_u = 0$, all the channels of u are equally balanced while if $G_u = 1$, all channels of u are as unequal as possible. The authors of [40] then define the imbalance of the whole G to be equal to the mean of all G_u for $u \in V$. Note that this imbalance metric measures the equality of balances, while equality might not be the objective of the owners of a channel. To use G_u as a trigger point for a node u to initiate one of the two rebalancing protocols, we must also constrain the implementation of `negotiateObjective`$(e_{u,v})$ to return a rebalancing demand that, if met, equalizes $e_{u,v}$ such that $b(e_{u,v}, u) = b(e_{u,v}, v)$.

The final part of the simulation concerns simulating the transactions in G. To achieve this, we start by randomly picking a sender and receiver of a transaction from an exponential distributions with scale $\lambda = 1$ and of which the output is scaled such that there is a 1 % chance that the node with the highest index is picked. Inspired by [47], this causes some senders and receivers to send and receive more transactions than other nodes. We then randomly generate a transaction amount from another exponential distribution with $\lambda = 15$, which mimics the presence of many small transaction amounts and some large transaction amounts. We then scale the drawn transaction amounts such that we receive a transaction success ratio of around 0.7 at the start of the simulation. Getting the transaction amount exactly representative of a real PCN is not important as we are only interested in the relative improvement of the transaction success ratio compared to a no rebalancing scenario. We obtain the path of the transaction using a non-distributed shortest-path algorithm. We chose to use shortest-path to limit the complexity of the simulation although in a future, more extensive analysis, the protocols should be compared with different routing algorithms for the transactions. Many purpose-made routing algorithms for PCNs such as SpeedyMurmurs [44] are made to tolerate imbalance in PCNs up to a certain point. We expect that using such routing algorithms can reduce the number of times a rebalancing protocol has to run, which in turn reduces the amount of locking required.

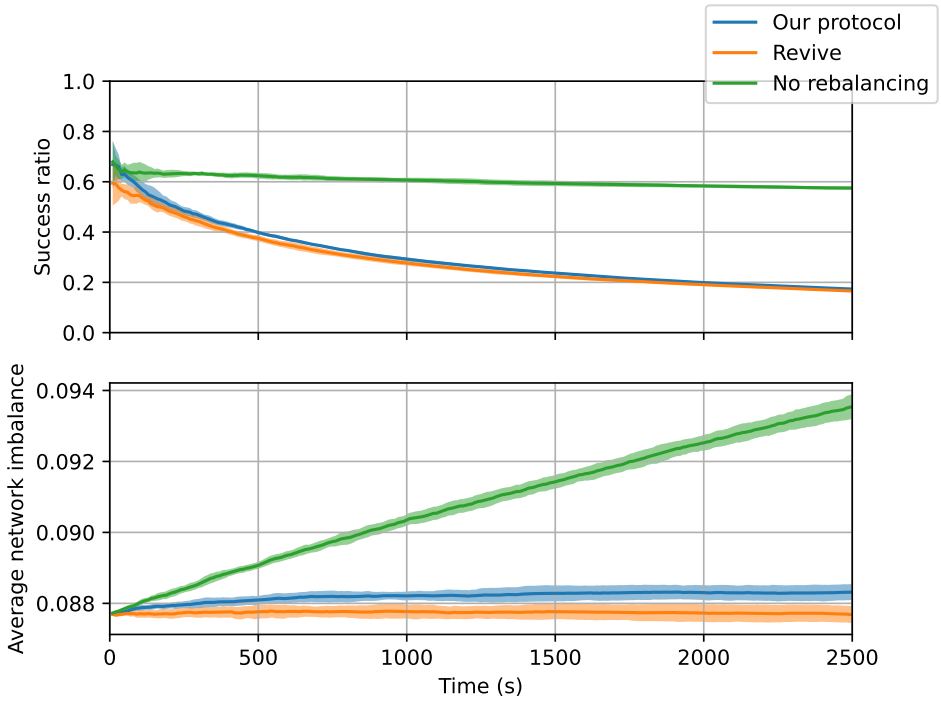
With this simulation setup, we executed the simulation 10 times for each protocol and with varying seeds. For the participant discovery used by both protocols, we set $h_c = 3$ and $I_m = 5$ which keeps $|P|$ below $5^3 = 125$. For the transaction generation protocol, we chose $\rho = 20\,\%$ as Subfigure 5.2a showed that, for small $|P|$, this produces the best result attainable with the minimum amount of messages. Each simulation consists of 5000 transactions which are started every 500 ms. We trigger a node u to start rebalancing when G_u is larger than or equal to the Gini coefficient trigger point ϕ s.t. $G_u \geq \phi$. We use $\phi = 0.2$ for our simulation and the results of this experiment can be found in Figure 5.4.

Looking at the results in Subfigure 5.4a, it is interesting to see that both our transaction generation protocol and Revive achieve a lower success ratio compared to a scenario where no rebalancing takes place. When comparing the protocol's effects on the average network imbalance $1/|V| \sum_{u \in V} G_u$, Revive performs better than our protocol in keeping the network imbalance stable. It is interesting to see that Revive performs only marginally better than our protocol, which causes us to believe that there is another underlying factor at play besides the benefit Revive has over our protocol in runtime and optimality of the solution.

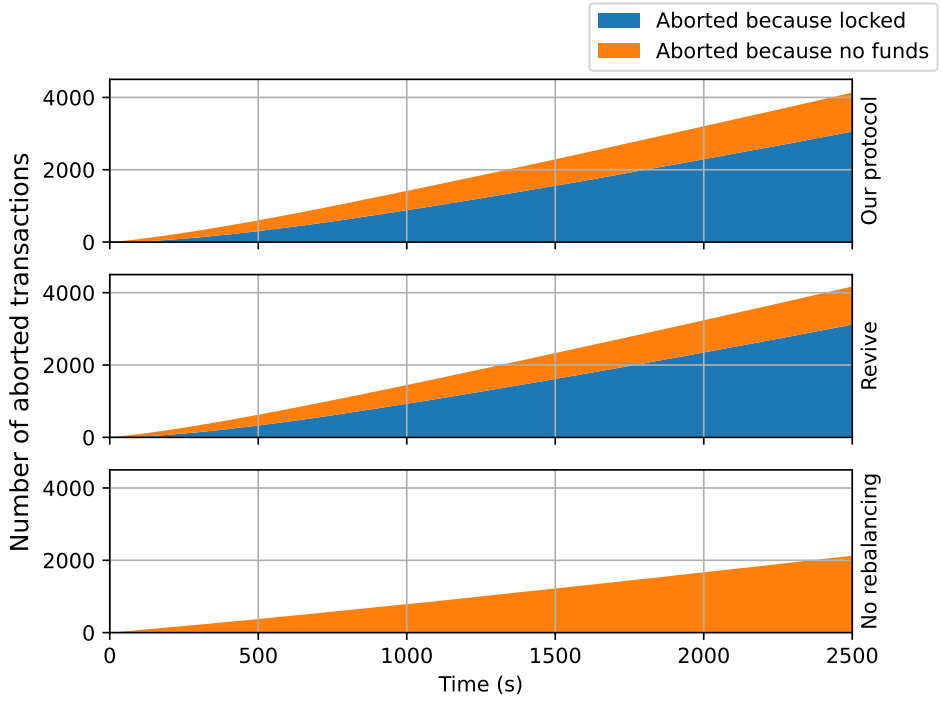
We believe that the primary influential factor is the locking of channels done by both protocols, and that it is both a factor in causing the low success ratio and making both protocols perform similarly. A requirement of the design of both protocols is that they need to lock the involved channels in order to execute the rebalancing. However, we believe that while these channels are locked, many potential transactions have to be aborted because the channels are not available for transaction routing. We can confirm this by looking at Subfigure 5.4b, which shows that the majority of aborted transactions during the simulations with rebalancing are caused by channels being locked.

Aborting transactions decreases the success ratio and also prevents these potential transactions from increasing the network imbalance. At the same time, the rebalancing protocols execute, which also decreases the network imbalance. The net effect is that both the success ratio and network imbalance decrease. This shows that non-concurrent rebalancing protocols have a 'fake' and a 'real' impact on the network imbalance, where the 'fake' impact is caused by the protocols locking channels during their execution and the 'real' impact is the rebalancing done by the protocols. An extreme example of 'fake' rebalancing is when a rebalancing protocol runs on the whole of G, which causes all channels to be locked, lowering the network imbalance but at the cost of aborting all transactions.

Other simulations To investigate the influence of our parameters on the dynamic simulation results, we also ran simulations with different h_c, I_m and ϕ. We present some of these simulations in Appendix C. From our design, we know that h_c and I_m influence $|P|$ and therefore the number of channels involved that are locked. In contrast, ϕ influences the frequency of the locking action as a high ϕ causes channels to be involved in more executions of rebalancing protocols when compared to a low ϕ. Our experiments show that ϕ has the most interesting impact on the results



(a) The success ratio and average network imbalance during the simulation. The solid lines represent the mean of the simulations and the confidence bounds represent one standard deviation from the mean.



(b) Number of aborted transactions during the simulation averaged over 10 simulations. Confidence bounds are not shown as the maximum standard deviation $\sigma_{\max} \approx 50$.

Figure 5.4: Comparison of our transaction generation protocol, Revive and no rebalancing in a dynamic simulation of a PCN. The simulation was run ten times with $h_c = 3$, $I_m = 3$, $\rho = 20\,\%$ and different seeds on $G_{\text{Lightning}}$.

of both protocols, as a high ϕ allows the average network imbalance to steadily increase with a marginally higher success ratio while a low ϕ steadily decreases the imbalance at the cost of a low success ratio. This suggests that the moment for a node to trigger a rebalancing is what determines if the network imbalance decreases or increases over time until reaching a steady-state.

As mentioned before, in future work we consider implementing a different transaction routing algorithm than shortest-path as such algorithms deal better with unavailable channels. This can be combined with running longer simulations and differing frequencies of transactions.

Chapter 6

Future Work and Conclusion

Payment channels and payment channel networks are promising technological developments that have the potential to minimise the impact of slow consensus mechanisms of cryptocurrency blockchains while preserving many of their security properties. This is achieved by reducing the number of transactions that need to be performed on the blockchain. Most payment transactions happen off-chain in a payment channel network, apart from dispute cases. Beyond disputes, the current design of payment channel networks also requires users to link their cryptocurrency wallets to their payment channels through opening and closing on-chain transactions. Such transactions have to take place not only when a user wishes to establish a new channel with another user, but also when the user runs out of funds in a channel. Running out of funds is a common occurrence in payment channel networks and is the reason why much research concerning payment channel networks has focused on creating transaction routing algorithms that take this aspect of payment channels into account. The main focus of this work was to investigate and design a new privacy-preserving rebalancing protocol, which is an alternative to the opening and closing of channels. Our main motivation lies in our analysis of the current state-of-the-art solution Revive [18], which did not provide any privacy for its users. From our observations, we formulated the following research question:

> How to construct a protocol that allows an arbitrary set of users in a payment channel network to securely rebalance their channels while achieving sender, receiver, value, channel balance and path privacy?

In this chapter, we look back at our designs and evaluation, and discuss how well we achieve our research question and objectives. We also discuss potential focus areas for future research after which we conclude this thesis.

6.1 Designs and findings

Based on the research question, we created a list of must-have and should-have requirements for our designs. Requirement 1d specified that our designs should include a privacy-preserving participant discovery protocol that allows participants to discover each other in order to execute protocols such as Revive or our transaction generation protocol. We therefore designed a participant discovery protocol that is based on a peer-to-peer architecture and produces a result where every participant obtains a list of anonymous identities of other participants.

In the theoretical performance evaluation of the participant discovery protocol, we showed that it is relatively efficient concerning time and message complexity. We confirmed its message complexity by running multiple simulations of the protocol on a snapshot of the Lightning Network. In the privacy analysis of the protocol, we found that the protocol provides participation anonymity depending on whether the subgraph G' of participating edges and nodes is very different from the topology of the overall PCN G. The more different G and G' are, the harder it is for the adversary to break the participation anonymity. This is related to the number of corrupted nodes because as these increase, participation anonymity becomes easier to break for the adversary.

Besides designing a participant discovery protocol, we also designed a transaction generation protocol to carry out the rebalancing, similar to Revive. We designed the protocol to cover many of the must-have requirements, such as generating a non-trivial transaction set (Requirement 1a), providing balance security (Requirement 1b) and providing privacy (Requirement 1c). A secondary goal is to have a design that works concurrently (Requirement 2b) and provides performance that is close or equal to the performance of Revive (Requirement 2a). Throughout the design process, we chose to focus on a non-concurrent protocol to limit the complexity of the design, thereby preventing the design from meeting Requirement 2b.

In the security and privacy analysis of the transaction generation protocol, we showed that our protocol is secure in a malicious setting and that it can guarantee most of the privacy properties set in our requirements. The ones that cannot be guaranteed are reducible to participation anonymity. We showed that achieving participation anonymity depends on the graph topology. Relationship anonymity is the only privacy property that cannot be achieved under any condition due to the design of the COMMIT and EXEC phase of the protocol. To summarize, our analysis showed that our design meets Requirement 1b in a malicious setting and the privacy properties of Requirement 1c if restrictions are put on the malicious setting.

In the theoretical performance analysis of the transaction generation protocol, we showed that the upper bound on the message complexity of the protocol mostly depends on the number of edges $|Q|$ in the rebalancing graph R. We also showed that the upper bound on the time complexity of the protocol is most dependent on the number of UPDATE messages sent during the execution of the protocol. We were only able to achieve an upper bound on the complexity as we were unable, during the complexity analysis, to obtain an exact time and message complexity for the UPDATE phase of the transaction generation protocol.

In the practical performance analysis, we analysed the impact of the protocol's parameters on the number of rebalancing demands that were met by the execution of our transaction generation protocol. We showed that if a single execution has more than $20\,\% \cdot |P|$[1] rounds, a state of diminishing returns is reached where the number of demands met marginally increases for a linear increase in the number of messages sent. We also showed that, in a one-to-one comparison with Revive, our transaction generation protocol performs slightly worse in meeting rebalancing demands and much worse in the number of messages sent and runtime. We therefore have not strictly achieved Requirement 2a but do come close to the intended goal. We consider an actual optimal rebalancing solution to be unobtainable in all scenarios because of the peer-to-peer design of our transaction generation protocol.

Finally, we showed that in a dynamic simulation where the protocol is run simultaneously with transactions, our transaction generation protocol performs similarly to Revive. We argue that this mainly depends on the non-concurrent nature of both protocols, as locking channels lowers the success ratio of transactions more than a situation without any rebalancing. We, therefore, consider that, besides the reduced performance, the non-concurrent nature of our protocol is its biggest drawback. For a future design iteration, we consider Requirement 2b a must-have requirement and not a should-have.

6.2 Future work

In this section, we highlight potential ideas for future research based on the insights we obtained during the work for this thesis. The ideas are organised around improving the privacy, performance and evaluation of our and other rebalancing protocols.

6.2.1 Improving privacy

The main goal of our research was to create a privacy-preserving alternative to existing rebalancing solutions such as Revive. In Section 4.8 we showed that our protocol guarantees some, but not all of the privacy properties we aimed for. In future work, our design could be improved to increase the number of privacy properties it can guarantee. A first step in such an endeavour would be to carry out a full analysis of participation anonymity and see if a method can be found to guarantee

[1] |P| is the number of participants of the protocol

it in a malicious setting without depending on the complexity of the graph topology. If such a method is found, its impact might be substantial given that it could be used for other distributed algorithms outside of rebalancing protocols and on other types of networks outside of PCNs.

A more focused improvement to the privacy properties would be to redesign the COMMIT and EXEC phase of the transaction generation protocol in combination with a different multi-hop transaction mechanism such that our protocol provides relationship anonymity. We sketched a potential architecture for such a mechanism in Section 4.8.

Finally, we recognize that our chosen method is not the only approach which can provide privacy to users of a rebalancing protocol. Alternative solutions such as using Secure Multi-Party Computation to carry out the linear program, as used in Revive, might provide a more private and highly performing solution to on-demand rebalancing. However, we see the complexity of such a solution as the biggest challenge in realizing it.

6.2.2 Improving performance

During the design of our transaction generation protocol, we touched upon a fundamental problem of designing a privacy-preserving peer-to-peer protocol for cycle detection. In our protocol, we try to prevent other nodes from knowing the path and owner of a cycle as this reveals information about the transactions that are executed during the EXEC phase, such as the senders/receivers and the path they take. Keeping these cycle properties private creates a problem when two nodes detect the same cycle, as by design there is no easy way for them to determine that there is another node trying to claim the same cycle. This problem exists because we based our cycle detection algorithm on an existing solution of [43]. In [43], if the same cycle is detected by two or more nodes, the conflict is resolved by having all nodes concede to the node with the lowest id. This is only possible if all the conflicting nodes know each other's identity, in contrast to our solution. It would therefore be interesting to investigate if there exists a way for nodes to resolve such a conflict without revealing the identities of either node. This has the potential to increase performance, as our current conflict resolution removes the multi-claimed cycle altogether which reduces the number of demands met each round.

During the evaluation, it also became clear how much impact the blocking nature of on-demand rebalancing protocols have on the transaction success ratio of a PCN. Our protocol and Revive are unable to improve the transaction success ratio due to the locking of channels during their execution. We, therefore, see it as an interesting potential research question if our protocol or Revive can be made concurrent or, if not possible, to design a new protocol that is both privacy-preserving *and* concurrent.

A more straightforward improvement to performance can also be made by redesigning the UPDATE and EXEC mechanisms of the transaction generation protocol as they currently have the largest impact on its message and time complexity.

6.2.3 Improving the evaluation

The evaluation of our protocols and Revive can be improved in many different ways. To increase the validity of our results, it would be beneficial to port both our protocol and Revive from Kotlin to the C programming language such that it can be run on the CLoTH simulator [7], which is a discrete event simulator specifically made to simulate the Lightning Network. Although we stand by the validity of our static evaluations of both protocols, the dynamic simulation of a PCN is much more complex due to the limited research and information that is available surrounding aspects such as the frequency, size and path of transactions. We, therefore, believe that the validity of the results of the dynamic simulation could be improved when using a tested and reviewed simulator for PCNs.

Another aspect that would improve the dynamic simulation is the implementation of different routing algorithms. CLoTH implements the reference Lightning client's routing algorithm, which is a modified Dijkstra's shortest path algorithm. One could also consider implementing routing algorithms such as SpeedyMurmurs [44] as those claim to achieve a higher transaction success ratio than Dijkstra's shortest path or the reference Lightning client implementation. One of the main goals of rebalancing is to improve the transaction success ratio of the PCN and as this is also

dependent on the routing algorithm, it would be interesting to see how the different combinations of routing algorithms and rebalancing protocols affect each other.

Besides comparing different routing algorithms to different on-demand rebalancing protocols, one could also consider other rebalancing trigger mechanisms besides the Gini coefficient network imbalance mechanism from [40]. One could consider triggering rebalancing after a certain deviation from the initial channel balance or a trigger based on historical data showing that the channel can only process less than x percent of the transactions it processed before. We consider it also possible that non-concurrent on-demand rebalancing, due to its high impact on transaction success, should be seen more as a manual cost-saving tool than an automatically triggered mechanism. This would mean that a rebalancing only runs when a user is unable to carry out a certain transaction on one of its channels and does not wish to close the channel and pay a fee to open a new one.

Finally, one can consider naively modifying parts of Revive to achieve one or more privacy properties. This would allow for a performance comparison between two rebalancing protocols that are closer in privacy-preserving behaviour, in contrast to our current comparison. A relatively simple option that increases Revive's privacy is to apply differential privacy [8] to the rebalancing demands received by the leader. Depending on how the privacy budget of the differential privacy mechanism is set, partial channel balance privacy can be achieved at the cost of performance. We expect that Revive's performance becomes more in line with our transaction generation protocol with such a privacy-preserving modification.

6.3 Conclusion

We present in this thesis an on-demand, privacy-preserving rebalancing protocol. In our work we focus on comparing our work to a non privacy-preserving on-demand rebalancing protocol called Revive [18]. We also present the first, to the best of our knowledge, participation discovery protocol that allows nodes in a graph to find other interested nodes to execute a distributed algorithm in a payment channel network.

Our rebalancing protocol design is inspired by the Rocha-Thatte distributed cycle detection algorithm [43]. We showed that it provides *balance security*, *balance conservation*, *path privacy* and *value privacy* in the presence of an adversary with the capability to corrupt nodes in the PCN. Furthermore, we showed that our protocols can provide *participation anonymity*, *channel balance privacy* and *sender/receiver privacy* if the adversary is limited in the number of corrupted nodes and if there are a large number of cycles and edges between honest nodes.

We implemented both designs and showed that the rebalancing obtained by our rebalancing protocol approaches the optimal solution of Revive, at the cost of an increased message and time complexity. We also showed that, although our rebalancing protocol is theoretically worse than Revive in terms of demands met, message and time complexity, both our protocol and Revive perform similarly in a dynamic PCN simulation where transactions take place simultaneously. We argue that this is most likely the result of the non-concurrent nature of both algorithms as they lock channels during their execution. This has a larger impact on the number of aborted transactions than a lack of balance in the channel.

Finally, we outlined directions for future research in this chapter. In our opinion, the largest two improvements to be made that would benefit our protocols would be to make their privacy guarantees more robust with regard to graph topology and to make the rebalancing protocol concurrent. We expect that a concurrent on-demand rebalancing protocol might be able to increase the transaction success ratio instead of decreasing it. A higher success ratio would provide a motivation for PCN users to start using rebalancing protocols which, in the end, reduces the frequency of on-chain transactions for closing and opening channels.

Nomenclature

Λ_d	Implementation of a distributed algorithm, page 30
$\mathcal{G}_{\text{Ledger}}$	Global ledger functionality, page 25
ν	Execution id of the execution of an algorithm, page 33
ϕ	Gini coefficient trigger point after which to start a rebalancing protocol, page 78
ρ	Percentage of participants that should become a leader, page 31
φ_{exec}	Information required to execute a conditional transaction, page 23
φ_{setup}	Information required to setup a conditional transaction, page 23
$\vec{\Delta}_m$	The minimum demand of a cycle, page 31
$\vec{\Delta}_{\text{EX}}$	The demand to be executed on a cycle, page 31
$\vec{\Delta}_{u,v}$	The demand or rebalancing objective of an edge $e_{u,v}$, page 24
A_u	Anonymous identity of node u, page 30
$b(e_{u,v}, v)$	Balance function, returns the balance of a undirected edge, page 23
c	Rebalancing cycle, page 24
$c(e)$	Capacity function, returns the capacity of a undirected edge, page 23
E	Set of all $e_{u,v}$ edges in G, page 23
E_p	Set of edges participating in the execution of Λ_d, page 30
E_u	Set of all edges e involving u in G, page 23
$e_{u,v}$	Undirected edge between two nodes u and v, represents a payment channel, page 23
G	Undirected multigraph, represents a PCN, page 23
h_c	Hop count, page 30
$I_{\max}$	Maximum number of INVITEs a node may send, page 30
$N(u)$	Set of all neighbours of u, page 23
P	Set of participants for the execution of Λ_d, page 30
p	Path in G, page 23
Q	Set of all $q_{u,v}$ edges in R, page 24
Q_i^u	Set of all incoming edges $q_{i,u}$ of node u, page 24
$q_i^{u,c}$	Incoming edge of a cycle c in a node u, page 24

Q_o^u	Set of all outgoing edges $q_{u,i}$ of node u, page 24
$q_o^{u,c}$	Outgoing edge of a cycle c in a node u, page 24
Q_u	Set of all edges q of node u, page 24
$q_{u,v}$	Directed weighted edge between two nodes u and v, represents a $\vec{\Delta}_{u,v}$, page 24
R	Directed weighted multigraph, also called a rebalancing graph, page 24
r	Receiver of a transaction, page 23
S	Settings for the execution of Λ_d, page 30
s	Sender of a transaction, page 23
T_c	Set of cycle tags t_c, page 31
T_c^u	Set of cycle tags t_c owned by participant u, page 31
$T_c^{\neg u}$	Set of cycle tags t_c not owned by participant u, page 31
T_r	Set of cycle detection tags t_r, page 31
t_r	Cycle detection tag, page 31
V	Set of nodes in G, page 23
$w(q)$	Weight function, returns weight of edge q, page 24
x	Transaction amount, page 23

Bibliography

[1] Baruch Awerbuch. "Complexity of network synchronization". In: *Journal of the ACM* 32.4 (1985-10), pp. 804–823. ISSN: 0004-5411, 1557-735X. DOI: 10.1145/4221.4227. URL: https://dl.acm.org/doi/10.1145/4221.4227 (visited on 2021-11-15).

[2] *Block Chain — Bitcoin*. URL: https://developer.bitcoin.org/devguide/block_chain.html (visited on 2021-09-01).

[3] Vitalik Buterin. *Ethereum Whitepaper*. ethereum.org. 2021-08-31. URL: https://ethereum.org/en/whitepaper/ (visited on 2021-09-15).

[4] Jan Camenisch, Susan Hohenberger, and Anna Lysyanskaya. "Compact E-Cash". In: *Advances in Cryptology – EUROCRYPT 2005*. Ed. by Ronald Cramer. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer, 2005, pp. 302–321. ISBN: 978-3-540-32055-5. DOI: 10.1007/11426639_18.

[5] R. Canetti. "Universally composable security: a new paradigm for cryptographic protocols". In: *Proceedings 42nd IEEE Symposium on Foundations of Computer Science*. Proceedings 42nd IEEE Symposium on Foundations of Computer Science. ISSN: 1552-5244. 2001-10, pp. 136–145. DOI: 10.1109/SFCS.2001.959888.

[6] M. Conoscenti, A. Vetrò, and J. C. De Martin. "Hubs, Rebalancing and Service Providers in the Lightning Network". In: *IEEE Access* 7 (2019). Conference Name: IEEE Access, pp. 132828–132840. ISSN: 2169-3536. DOI: 10.1109/ACCESS.2019.2941448.

[7] Marco Conoscenti et al. "The CLoTH Simulator for HTLC Payment Networks with Introductory Lightning Network Performance Results". In: *Information* 9.9 (2018-09). Number: 9 Publisher: Multidisciplinary Digital Publishing Institute, p. 223. DOI: 10.3390/info9090223. URL: https://www.mdpi.com/2078-2489/9/9/223 (visited on 2020-12-15).

[8] Cynthia Dwork and Aaron Roth. "The Algorithmic Foundations of Differential Privacy". In: *Foundations and Trends® in Theoretical Computer Science* 9.3 (2014-08-10). Publisher: Now Publishers, Inc., pp. 211–407. ISSN: 1551-305X, 1551-3068. DOI: 10.1561/0400000042. URL: https://www.nowpublishers.com/article/Details/TCS-042 (visited on 2022-01-19).

[9] S. Dziembowski et al. "Perun: Virtual Payment Hubs over Cryptocurrencies". In: *2019 IEEE Symposium on Security and Privacy (SP)*. 2019 IEEE Symposium on Security and Privacy (SP). ISSN: 2375-1207. 2019-05, pp. 106–123. DOI: 10.1109/SP.2019.00020.

[10] Lisa Eckey et al. *Splitting Payments Locally While Routing Interdimensionally*. 555. 2020. URL: https://eprint.iacr.org/2020/555 (visited on 2020-11-09).

[11] Yuup van Engelshoven. "Designing and Evaluating Rebalancing Algorithms for Payment Channel Networks". Master. Delft: Delft University of Technology, 2019-12-10. 96 pp. URL: https://repository.tudelft.nl/islandora/object/uuid%3A32bf01ba-b6c6-410e-b0e1-b15d95eb05d9 (visited on 2021-01-13).

[12] George S. Fishman. *Discrete-Event Simulation: Modeling, Programming, and Analysis*. Ed. by Peter Glynn and Stephen M. Robinson. Springer Series in Operations Research. Google-Books-ID: y5fuBwAAQBAJ. Springer Science & Business Media, 2013-03-09. 554 pp. ISBN: 978-1-4757-3552-9.

[13] Matthew Green and Ian Miers. "Bolt: Anonymous Payment Channels for Decentralized Currencies". In: *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*. CCS '17: 2017 ACM SIGSAC Conference on Computer and Communications Security. Dallas Texas USA: ACM, 2017-10-30, pp. 473–489. ISBN: 978-1-4503-4946-8. DOI: 10.1145/3133956.3134093. URL: https://dl.acm.org/doi/10.1145/3133956.3134093 (visited on 2020-12-15).

[14] Y. Guo, J. Tong, and C. Feng. "A Measurement Study of Bitcoin Lightning Network". In: *2019 IEEE International Conference on Blockchain (Blockchain)*. 2019 IEEE International Conference on Blockchain (Blockchain). 2019-07, pp. 202–211. DOI: 10.1109/Blockchain.2019.00034.

[15] Jordi Herrera-Joancomartí et al. "On the Difficulty of Hiding the Balance of Lightning Network Channels". In: *Proceedings of the 2019 ACM Asia Conference on Computer and Communications Security*. Asia CCS '19. New York, NY, USA: Association for Computing Machinery, 2019-07-02, pp. 602–612. ISBN: 978-1-4503-6752-3. DOI: 10.1145/3321705.3329812. URL: https://doi.org/10.1145/3321705.3329812 (visited on 2020-11-19).

[16] *Internet Protocol - RFC 791*. 791. RFC Editor, 1981-09, p. 51. URL: https://www.rfc-editor.org/rfc/rfc791.txt (visited on 2021-10-02).

[17] S. Jiang et al. "Policy assessments for the carbon emission flows and sustainability of Bitcoin blockchain operation in China". In: *Nature Communications* 12.1 (2021). DOI: 10.1038/s41467-021-22256-3.

[18] Rami Khalil and Arthur Gervais. "Revive: Rebalancing Off-Blockchain Payment Networks". In: *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*. CCS '17. New York, NY, USA: Association for Computing Machinery, 2017-10-30, pp. 439–453. ISBN: 978-1-4503-4946-8. DOI: 10.1145/3133956.3134033. URL: https://doi.org/10.1145/3133956.3134033 (visited on 2020-12-03).

[19] A. Kiayias and O. S. T. Litos. "A Composable Security Treatment of the Lightning Network". In: *2020 IEEE 33rd Computer Security Foundations Symposium (CSF)*. 2020 IEEE 33rd Computer Security Foundations Symposium (CSF). ISSN: 2374-8303. 2020-06, pp. 334–349. DOI: 10.1109/CSF49147.2020.00031.

[20] Bernhard Korte and Jens Vygen. "Bin-Packing". In: *Combinatorial Optimization: Theory and Algorithms*. 3rd ed. Algorithms and Combinatorics 21. Berlin, Heidelberg: Springer, 2006, pp. 426–441. ISBN: 978-3-540-29297-5. DOI: 10.1007/3-540-29297-7_18. URL: https://doi.org/10.1007/3-540-29297-7_18 (visited on 2021-11-09).

[21] Bernhard Korte and Jens Vygen. "Linear Programming". In: *Combinatorial Optimization: Theory and Algorithms*. 3rd ed. Algorithms and Combinatorics 21. Berlin, Heidelberg: Springer, 2006, pp. 49–64. ISBN: 978-3-540-29297-5. DOI: 10.1007/3-540-29297-7_3. URL: https://doi.org/10.1007/3-540-29297-7_3 (visited on 2021-11-09).

[22] P. Li, T. Miyazaki, and W. Zhou. "Secure Balance Planning of Off-blockchain Payment Channel Networks". In: *IEEE INFOCOM 2020 - IEEE Conference on Computer Communications*. IEEE INFOCOM 2020 - IEEE Conference on Computer Communications. ISSN: 2641-9874. 2020-07, pp. 1728–1737. DOI: 10.1109/INFOCOM41043.2020.9155375.

[23] Wenzheng Li, Mingsheng He, and Sang Haiquan. "An Overview of Blockchain Technology: Applications, Challenges and Future Trends". In: *2021 IEEE 11th International Conference on Electronics Information and Emergency Communication (ICEIEC)2021 IEEE 11th International Conference on Electronics Information and Emergency Communication (ICEIEC)*. 2021 IEEE 11th International Conference on Electronics Information and Emergency Communication (ICEIEC)2021 IEEE 11th International Conference on Electronics Information and Emergency Communication (ICEIEC). ISSN: 2377-844X. 2021-06, pp. 31–39. DOI: 10.1109/ICEIEC51955.2021.9463842.

[24] *Lightning Network Explorer*. URL: https://explorer.acinq.co/ (visited on 2021-02-24).

[25] *Lightning RFC*. original-date: 2016-11-14T19:21:45Z. 2020-11-12. URL: https://github.com/lightningnetwork/lightning-rfc (visited on 2020-11-12).

[26] *lightningnetwork/lnd*. original-date: 2016-01-16T08:19:33Z. 2021-09-03. URL: https://github.com/lightningnetwork/lnd (visited on 2021-09-03).

[27] S. Lin, J. Zhang, and W. Wu. "FSTR: Funds Skewness Aware Transaction Routing for Payment Channel Networks". In: *2020 50th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. 2020 50th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN). ISSN: 1530-0889. 2020-06, pp. 464–475. DOI: 10.1109/DSN48063.2020.00060.

[28] Zhichun Lu, Runchao Han, and Jiangshan Yu. *General Congestion Attack on HTLC-Based Payment Channel Networks*. 456. 2020. URL: https://eprint.iacr.org/2020/456 (visited on 2021-09-21).

[29] Giulio Malavolta et al. "Anonymous Multi-Hop Locks for Blockchain Scalability and Interoperability". In: *Proceedings 2019 Network and Distributed System Security Symposium*. Network and Distributed System Security Symposium. San Diego, CA: Internet Society, 2019. ISBN: 978-1-891562-55-6. DOI: 10.14722/ndss.2019.23330. URL: https://www.ndss-symposium.org/wp-content/uploads/2019/02/ndss2019_09-4_Malavolta_paper.pdf (visited on 2020-12-03).

[30] Giulio Malavolta et al. "Concurrency and Privacy with Payment-Channel Networks". In: *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*. CCS '17: 2017 ACM SIGSAC Conference on Computer and Communications Security. Dallas Texas USA: ACM, 2017-10-30, pp. 455–471. ISBN: 978-1-4503-4946-8. DOI: 10.1145/3133956.3134096. URL: https://dl.acm.org/doi/10.1145/3133956.3134096 (visited on 2020-12-15).

[31] Giulio Malavolta et al. "SilentWhispers: Enforcing Security and Privacy in Decentralized Credit Networks". In: *Proceedings 2017 Network and Distributed System Security Symposium*. Network and Distributed System Security Symposium. San Diego, CA: Internet Society, 2017. ISBN: 978-1-891562-46-4. DOI: 10.14722/ndss.2017.23448. URL: https://www.ndss-symposium.org/ndss2017/ndss-2017-programme/silentwhispers-enforcing-security-and-privacy-decentralized-credit-networks/ (visited on 2020-11-16).

[32] Bernard Marr. *A Short History Of Bitcoin And Crypto Currency Everyone Should Read*. Forbes. 2017-12-06. URL: https://www.forbes.com/sites/bernardmarr/2017/12/06/a-short-history-of-bitcoin-and-crypto-currency-everyone-should-read/ (visited on 2021-08-05).

[33] Stefano Martinazzi and Andrea Flori. "The evolving topology of the Lightning Network: Centralization, efficiency, robustness, synchronization, and anonymity". In: *PLOS ONE* 15.1 (2020-01-15). Publisher: Public Library of Science, e0225966. ISSN: 1932-6203. DOI: 10.1371/journal.pone.0225966. URL: https://journals.plos.org/plosone/article?id=10.1371/journal.pone.0225966 (visited on 2021-11-04).

[34] Patrick McGuire. *Such Weird: The Founders of Dogecoin See the Meme Currency's Tipping Point*. 2013-12-23. URL: https://www.vice.com/en/article/jp5x3d/dogecoins-founders-believe-in-the-power-of-meme-currencies (visited on 2021-09-15).

[35] Suat Mercan, Enes Erdin, and Kemal Akkaya. "Improving transaction success rate in cryptocurrency payment channel networks". In: *Computer Communications* 166 (2021-01-15), pp. 196–207. ISSN: 0140-3664. DOI: 10.1016/j.comcom.2020.12.009. URL: http://www.sciencedirect.com/science/article/pii/S0140366420320156 (visited on 2021-01-07).

[36] Ralph C. Merkle. "A Digital Signature Based on a Conventional Encryption Function". In: *Advances in Cryptology — CRYPTO '87*. Ed. by Carl Pomerance. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer, 1988, pp. 369–378. ISBN: 978-3-540-48184-3. DOI: 10.1007/3-540-48184-2_32.

[37] Andrew Miller et al. "Sprites and State Channels: Payment Networks that Go Faster Than Lightning". In: *Financial Cryptography and Data Security*. Ed. by Ian Goldberg and Tyler Moore. Lecture Notes in Computer Science. Cham: Springer International Publishing, 2019, pp. 508–526. ISBN: 978-3-030-32101-7. DOI: 10.1007/978-3-030-32101-7_30.

[38] Pedro Moreno-Sanchez et al. “Privacy Preserving Payments in Credit Networks: Enabling trust with privacy in online marketplaces”. In: *NDSS Symposium 2015*. NDSS: Network and Distributed System Security Symposium. Internet Society, 2015-02-08. URL: https://www.ndss-symposium.org/ndss2015/ndss-2015-programme/privacy-preserving-payments-credit-networks-enabling-trust-privacy-online-marketplaces/ (visited on 2021-09-09).

[39] Satoshi Nakamoto. “Bitcoin: A Peer-to-Peer Electronic Cash System”. In: (2008), p. 9.

[40] Rene Pickhardt and Mariusz Nowostawski. “Imbalance measure and proactive channel rebalancing algorithm for the Lightning Network”. In: *arXiv:1912.09555 [cs]* (2019-12-19). arXiv: 1912.09555. URL: http://arxiv.org/abs/1912.09555 (visited on 2021-01-05).

[41] Joseph Poon and Thaddeus Dryja. “The Bitcoin Lightning Network”. In: (2016-01-14). DRAFT Version 0.5.9.2, p. 59. URL: https://lightning.network/lightning-network-paper.pdf.

[42] *Raiden Explorer*. URL: https://explorer.raiden.network (visited on 2021-02-24).

[43] Rodrigo Rocha and Bhalchandra Thatte. “Distributed cycle detection in large-scale sparse graphs”. In: 2015-08-25. DOI: 10.13140/RG.2.1.1233.8640.

[44] Stefanie Roos et al. “SpeedyMurmurs: Efficient Decentralized Routing for Path-Based Transactions”. In: *arXiv:1709.05748 [cs]* (2017-12-13). arXiv: 1709.05748. URL: http://arxiv.org/abs/1709.05748 (visited on 2020-11-16).

[45] D. Šatcs. “Understanding the lightning network capability to route payments”. Publisher: University of Twente. info:eu-repo/semantics/bachelorThesis. 2020-07-03. URL: https://essay.utwente.nl/82015/ (visited on 2021-09-03).

[46] Meng Shen, Liehuang Zhu, and Ke Xu. “Blockchain and Data Sharing”. In: *Blockchain: Empowering Secure Data Sharing*. Ed. by Meng Shen, Liehuang Zhu, and Ke Xu. Singapore: Springer, 2020, pp. 15–27. ISBN: 9789811559396. DOI: 10.1007/978-981-15-5939-6_2. URL: https://doi.org/10.1007/978-981-15-5939-6_2 (visited on 2021-09-16).

[47] Vibhaalakshmi Sivaraman et al. “High Throughput Cryptocurrency Routing in Payment Channel Networks”. In: *arXiv:1809.05088 [cs]* (2020-03-23). arXiv: 1809.05088. URL: http://arxiv.org/abs/1809.05088 (visited on 2020-12-17).

[48] L. M. Subramanian, G. Eswaraiah, and R. Vishwanathan. “Rebalancing in Acyclic Payment Networks”. In: *2019 17th International Conference on Privacy, Security and Trust (PST)*. 2019 17th International Conference on Privacy, Security and Trust (PST). ISSN: 2643-4202. 2019-08, pp. 1–5. DOI: 10.1109/PST47121.2019.8949051.

[49] Sergei Tikhomirov, Pedro Moreno-Sanchez, and Matteo Maffei. “A Quantitative Analysis of Security, Anonymity and Scalability for the Lightning Network”. In: *2020 IEEE European Symposium on Security and Privacy Workshops (EuroS&PW)*. 2020 IEEE European Symposium on Security and Privacy Workshops (EuroS&PW). Genoa, Italy: IEEE, 2020-09, pp. 387–396. ISBN: 978-1-72818-597-2. DOI: 10.1109/EuroSPW51379.2020.00059. URL: https://ieeexplore.ieee.org/document/9229723/ (visited on 2021-09-21).

[50] Tomas Toft. “Solving Linear Programs Using Multiparty Computation”. In: *Financial Cryptography and Data Security*. Ed. by Roger Dingledine and Philippe Golle. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer, 2009, pp. 90–107. ISBN: 978-3-642-03549-4. DOI: 10.1007/978-3-642-03549-4_6.

[51] Itay Tsabary et al. “MAD-HTLC: Because HTLC is Crazy-Cheap to Attack”. In: *arXiv:2006.12031 [cs]* (2021-03-25). arXiv: 2006.12031. URL: http://arxiv.org/abs/2006.12031 (visited on 2021-09-21).

[52] *Visa Fact Sheet*. Fact sheet. 2019-07. URL: https://usa.visa.com/dam/VCOM/global/about-visa/documents/visa-fact-sheet-july-2019.pdf (visited on 2021-10-02).

[53] Paul Wackerow et al. *Proof-of-stake (PoS)*. ethereum.org. 2021-07-29. URL: https://ethereum.org (visited on 2021-09-01).

[54] R. Yu et al. “CoinExpress: A Fast Payment Routing Mechanism in Blockchain-Based Payment Channel Networks”. In: *2018 27th International Conference on Computer Communication and Networks (ICCCN)*. 2018 27th International Conference on Computer Communication and Networks (ICCCN). ISSN: 1095-2055. 2018-07, pp. 1–9. DOI: 10.1109/ICCCN.2018.8487351.

Appendix A

Pseudocode of the participant discovery protocol

Algorithm 14 Pseudocode for the implementation of the participant discovery protocol for a node u (part 1/2)

```
1:  awake, started, processedResponses, invitesSend ← false
2:  ν, A_u, e_p, S_d, alg ← null
3:  P, E_c, E_a, I_m ← ∅
4:
5:  procedure START(h_c, I_max, S, Λ_d)
6:      awake ← true
7:      started ← true
8:      invitesSend ← true
9:      ν ← randomly generated identifier
10:     A_u ← randomly generated identifier
11:     P ← P ∪ A
12:     for all e ∈ E_u do
13:         send (invite;ν,h_c, I_max, S, Λ_d) on edge e
14:
15: upon receipt of (invite;id, h_c, I_max, S, Λ_d) on edge j do
16:     if not willing to participate with j then
17:         send (deny;id) on edge j and return
18:     if ν ≠ null ∧ ν ≠ id then                ▷ Ignore other executions of this algorithm
19:         send (deny;id) on edge j and return
20:     if ν = null then                         ▷ Node gets claimed by the starting node
21:         awake ← true
22:         ν ← id
23:         S_d ← S
24:         alg ← Λ_d
25:         A ← randomly generated identifier
26:         P ← P ∪ A
27:         e_p ← j
28:
29:     if ¬invitesSend ∧ h_c − 1 > 0 ∧ |E_u| > 1 then
30:         invitesSend ← true
31:         e_p ← j
32:
33:         for all e ∈ E_u : e ≠ e_p ∧ e has not been denied do
34:             send (invite; ν, h_c − 1, I_max, S, Λ_d) on edge e
35:             I_m ← I_m ∪ e
36:
37:             if |I_m| > I_max then
38:                 break
39:
40:         return
41:
42:     E_a ← E_a ∪ j
43:     send (accept; ν, P, e_p = j) on edge j and return
44:
```

Algorithm 15 Pseudocode for the implementation of the participant discovery protocol for a node u (part 2/2)

```
45: upon receipt of (accept;id, R, isChild) on edge j do
46:     if ¬awake then return
47:     else if ν ≠ id ∨ j ∉ I_m then
48:         send (deny;id) on edge j and return
49:
50:     P ← P ∪ R
51:     E_a ← E_a ∪ j
52:     I_m ← I_m − j
53:
54:     if isChild then
55:         E_c ← E_c ∪ j
56:
57:     handleResponses()
58:
59: upon receipt of (deny;id) on edge j do
60:     if ¬awake ∨ j ∉ I_m then return
61:     I_m ← I_m − j
62:     handleResponses()
63:
64: procedure handleResponses
65:     if |I_m| = 0 ∧ ¬processedResponses then
66:         processedResponses ← true
67:
68:         if started then
69:             if E_a ≠ ∅ then
70:                 for all e ∈ E_c do
71:                     send (finish;ν, P) on edge e
72:                 execute alg(ν, A_u, P, E_a, S_d)
73:             terminate
74:         else
75:             E_a ← E_a ∪ e_p
76:             send (accept;ν, P, true) on edge e_p
77:
78: upon receipt of (finish;id, R) on edge j do
79:     if ¬awake then return
80:     else if ν ≠ id then
81:         send (deny;id) on edge j and return
82:
83:     P ← R
84:     for all e ∈ E_c do                    ▷ Propagate participant list along tree
85:         send (finish;ν, P) on edge e
86:     execute alg(ν, A_u, P, E_a, S_d)
87:     terminate
88:
```

Appendix B

Pseudocode of the transaction generation protocol

Algorithm 16 Pseudocode for the implementation of the transaction generation protocol for a node u (part 1/9)

▷ Variables that are received from the participant discovery protocol
$\nu \leftarrow$ Execution ID
$A_u \leftarrow$ Anonymous ID of u
$P \leftarrow$ Set of participants of the protocol
$E_a \leftarrow$ Set of participating edges of u
$S \leftarrow$ Settings of the protocol

▷ Variables that are reset during every invocation of the protocol
awake $\leftarrow$ false
$i_r \leftarrow 0$ ▷ Round index
$A_l \leftarrow \epsilon$ ▷ Anonymous id of the round leader
$L_P \leftarrow []$ ▷ Sorted list of participants
$Q_{\text{out}} \leftarrow$ Set of edges that have outgoing demand
$Q_{\text{in}} \leftarrow$ Set of edges that have incoming demand

▷ Variables that are reset for every round of the protocol
roundState $\leftarrow$WAIT ▷ Can be one of: WAIT, REQ, SUC, COM, EXEC
$n_{\text{sendRequests}} \leftarrow 0$
forwardedNextRoundMes, *execSafe* $\leftarrow$ false
$D_{t_r,q} \leftarrow \emptyset$ ▷ Map of $t_r \rightarrow q$
$D^{u}_{t_c} \leftarrow \emptyset$ ▷ Map of $t_c \rightarrow \left[q_e^{u,c}, q_s^{u,c}, \vec{\Delta}_m, \textit{completed}\right]$, where *completed* is a boolean
$D^{\neg u}_{t_c} \leftarrow \emptyset$ ▷ Map of $t_r \rightarrow \left[\vec{\Delta}, q_o^{u,c}, q_i^{u,c}\right]$
$D_{t_c,\varphi_{\text{exec}}} \leftarrow \emptyset$ ▷ Map of $t_c \rightarrow \varphi_{\text{exec}}$
$L_{\text{rREQUEST}}, L_{\text{rCOMMIT}}, L_{\text{rCycleCOMMIT}}, L_{\text{rSUCCESS}} \leftarrow []$
$T_r^{\text{recv.}} \leftarrow \emptyset$

▷ Predicates
$Pr_{\nu}(id) \leftarrow \nu = \epsilon \vee \nu \neq id$
$Pr_{r,\text{early}}(A_i) \leftarrow L_P.\texttt{indexOf}(A_i) < i_r$
$Pr_{r,\text{future}}(A_i) \leftarrow L_P.\texttt{indexOf}(A_i) > i_r$
$Pr_{r,\text{diff}}(A_i) \leftarrow A_i \neq A_l$
$Pr_{\text{nextRound}}() \leftarrow \textit{roundState} = \text{WAIT} \vee (\textit{execSafe} \wedge (\textit{forwardedNextRoundMes} \vee A_l = A_u))$

Algorithm 17 Pseudocode for the implementation of the transaction generation protocol for a node u (part 2/9)

```
33: procedure WAKEUP
34:     if awake then return
35:
36:     awake ← true
37:     L_P ← sort P using fixed sorting algorithm
38:     A_l ← L_P[i_r]
39:
40:     for all e ∈ E_a do
41:         Lock edge e
42:         q ← negotiateObjective(e)
43:         if w(q) < 0 then
44:             Q_out ← Q_out ∪ {q}
45:         else
46:             Q_in ← Q_in ∪ {q}
47:
48:     if A_l = A_u then
49:         startRound()
50:
51: procedure STARTROUND
52:     if |Q_out| = 0 then
53:         nextRound()
54:         return
55:
56:     roundState ← REQ
57:     for q ∈ Q_out do
58:         t_r ← randomly generated identifier
59:         D_{t_r,q}.put(t_r, q)
60:         n_sendRequests ← n_sendRequests + 1
61:
62:         send (request;ν, A_l, {t_r}) on edge q
63:
64: procedure NEXTROUND
65:     if A_l = A_u then
66:         for all e ∈ E_a do
67:             send (next_round;ν, A_l) on edge e
68:
69:     i_r ← i_r + 1
70:     A_l ← L_P[i_r]
71:     if i_r ≥ S['maxRound'] then
72:         Unlock participating edges e ∈ E_a
73:         terminate
74:
75:     Reset round variables
76:     if A_l = A_u then
77:         startRound()
```

Algorithm 18 Pseudocode for the implementation of the transaction generation protocol for a node u (part 3/9)

78: **procedure** CHECKFORCYCLES($m = (\text{request};T_r)$ **on edge** j)
79: $I \leftarrow \{t_r | (t_r, q) \in D_{t_r,q}\} \cap T_r$
80: **if** $|I| > 0$ **then**
81: $t_r^{\text{mtch}} \leftarrow$ randomly picked t_r out of I
82: $L_{\text{rREQUEST}} \leftarrow L_{\text{rREQUEST}} - m$
83: $t_c \leftarrow$ randomly generated identifier
84: $\vec{\Delta}_m \leftarrow w(j)$
85: $D^u_{t_c}.\texttt{put}\left(t_c, \left[j, D_{t_r,q}.\texttt{get}(t_r^{\text{mtch}}), \vec{\Delta}_m, \text{false}\right]\right)$
86:
87: **send** $(\text{success};\nu, A_l, \{(t_c, \vec{\Delta}_m)\})$ **on edge** j
88: **return** true
89:
90: **return** false
91:
92: **procedure** CHECKFORCYCLESANDNEWTAGS($m = (\text{request};T_r)$ **on edge** j)
93: **if** `checkForCycles(`m`)` **then return** false
94:
95: $T_r^{\text{new}} \leftarrow T_r - T_r^{\text{recv}}$
96: **if** $|T_r^{\text{new}}| > 0$ **then**
97: $T_r^{\text{recv}} \leftarrow T_r^{\text{recv}} \cup T_r$
98:
99: **for all** $q \in Q_{\text{out}}$ that did not reply with SUCCESS or FAIL this round **do**
100: **send** $(\text{update};\nu, A_l, T_r^{\text{new}})$ **on edge** q
101:
102: **return** true

Algorithm 19 Pseudocode for the implementation of the transaction generation protocol for a node u (part 4/9)

```
103: upon receipt of m = (request;id, A_i, T_r) on edge j do
104:     check-action: Not finished with participant discovery → defer, Pr_ν(id) → disallow
105:     check-action: Pr_{r,future}(A_i) → defer, Pr_{r,early}(A_i) → disallow
106:
107:     if |Q_out| = 0 then
108:         send (fail;ν, A_l, 0) on edge j return          ▷ 0 = no possibility for a SUCCESS
109:
110:     if roundState = WAIT then
111:         roundState = REQ
112:         T_r^recv ← T_r^recv ∪ T_r
113:
114:         for all q ∈ Q_out do
115:             t_r ← randomly generated identifier
116:             D_{t_r,q}.put(t_r, q)
117:             n_sendRequests ← n_sendRequests + 1
118:
119:             send (request;ν, A_l, T_r^recv ∪ {t_r}) on edge q
120:     else if roundState = REQ then
121:         if ¬checkForCyclesAndNewTags(m) then return
122:     else
123:         send (fail;ν, A_l, 0) on edge j and return
124:
125:     L_rREQUEST ← L_rREQUEST * m
126:
127:
128: upon receipt of m = (update;id, A_i, T_r) on edge j do
129:     check-action: Pr_ν(id) - disallow, ¬awake - disallow
130:     check-action: Pr_{r,future}(A_i) - defer, Pr_{r,early}(A_i) - disallow
131:
132:     if roundState ≠ REQ ∨ already send SUCCESS on j this round then
133:         return
134:
135:     if ¬checkForCyclesAndNewTags(m) then return
136:
```

Algorithm 20 Pseudocode for the implementation of the transaction generation protocol for a node u (part 5/9)

137: **upon receipt of** $m = (\text{success}; id, A_i, T_c)$ **on edge** j **do**
138: **check-action:** $Pr_\nu(id) \rightarrow$ **disallow**, $\neg awake \rightarrow$ **disallow**
139: **check-action:** $Pr_{r,\text{diff}}(A_i) \rightarrow$ **disallow**
140:
141: **if** *roundState* = REQ **then**
142: $L_{\text{rSUCCESS}} \leftarrow L_{\text{rSUCCESS}} * m$
143: $n_{\text{sendRequests}} \leftarrow n_{\text{sendRequests}} - 1$
144: `replyToRequests`()
145:
146:
147: **upon receipt of** $m = (\text{fail}; id, A_i, w_{\text{failure}})$ **on edge** j **do**
148: **if** *roundState* = REQ $\wedge\, w_{\text{failure}} = 0$ **then**
149: $n_{\text{sendRequests}} \leftarrow n_{\text{sendRequests}} - 1$
150: `replyToRequests`()
151:
152:
153: **procedure** REPLYTOREQUESTS
154: **if** $n_{\text{sendRequests}} \neq 0 \vee$ *roundState* $\neq$ REQ **then return**
155:
156: *roundState* $\leftarrow$ SUC
157: **for all** $(m = (\text{success}; id, A_i, T_c)$ on edge $j) \in L_{\text{rSUCCESS}}$ **do**
158: **for all** $(t_c, \vec{\Delta}_m) \in T_c$ **do**
159: **if** $t_c \in D^u_{t_c}$ **then**
160: $L_{t_c} \leftarrow D^u_{t_c}.\texttt{get}(t_c)$
161:
162: **if** $\neg L_{t_c}[3] \vee \vec{\Delta}_m > L_{t_c}[2]$ **then**
163: $D^u_{t_c}.\texttt{put}\left(t_c, \left[L_{t_c}[0], j, \vec{\Delta}_m, \text{true}\right]\right)$
164: **else if** $t_c \notin D^{\neg u}_{t_c} \vee \vec{\Delta}_m > D^{\neg u}_{t_c}.\texttt{get}(t_c)[0]$ **then**
165: $D^{\neg u}_{t_c}.\texttt{put}\left(t_c, \left[\vec{\Delta}_m, j, \epsilon\right]\right)$
166:
167: **if** $A_l = A_u$ **then** ▷ Node started round
168: `commitLeader`()
169: **else**
170: $L_{D^{\neg u}_{t_c}} \leftarrow \left[[t_c, \vec{\Delta}_m, q^{u,c}_o, q^{u,c}_i] \mid (t_c, [\vec{\Delta}_m, q^{u,c}_o, q^{u,c}_i]) \in D^{\neg u}_{t_c}\right]$
171: **for all** $(m = (\text{request}; id, A_i, T_r)$ on edge $j) \in L_{\text{rREQUEST}}$ **do**
172: $N \leftarrow \texttt{splitEqually}(w(j), \left[\vec{\Delta}_m \mid [t_c, \vec{\Delta}_m, q^{u,c}_o, q^{u,c}_i] \in L_{D^{\neg u}_{t_c}}\right])$
173: $T^{\neg u}_c \leftarrow \emptyset$
174: **for** $i \in [0, |N|)$ **do**
175: $T^{\neg u}_c \leftarrow T^{\neg u}_c \cup \left\{(L_{D^{\neg u}_{t_c}}[i][0], N[i])\right\}$
176:
177: **send** $(\text{success}; \nu, A_l, T^{\neg u}_c)$ **on edge** j

Algorithm 21 Pseudocode for the implementation of the transaction generation protocol for a node u (part 6/9)

178: **procedure** COMMITLEADER
179: $D_{q_o} \leftarrow \emptyset$ ▷ Map of $q_o \to \left\{(t_c, \vec{\Delta}_{\text{EX}}, \varphi_{\text{setup}})\right\}$
180: $\texttt{commit}(D_{q_o})$
181:
182: **if** $|D^u_{t_c}| = 0$ **then**
183: $\texttt{nextRound}()$
184:
185: **procedure** COMMIT(D_{q_o})
186: **for all** $(t_c, \left[q_e^{u,c}, q_s^{u,c}, \vec{\Delta}_m, \textit{completed}\right]) \in D^u_{t_c}$ **do**
187: **if** $\neg \textit{completed}$ **then**
188: **continue**
189:
190: **if** $\vec{\Delta}_m > 0$ **then**
191: $\varphi_{\text{setup}}, \varphi_{\text{exec}} \leftarrow$ generated by a subroutine
192: $D_{t_c,\varphi_{\text{exec}}}.\texttt{put}(t_c, \varphi_{\text{exec}})$
193: $\texttt{pay}_{\text{cond}}(u, \texttt{Opp}(q_s^{u,c}, u), \vec{\Delta}_m, \varphi_{\text{setup}})$
194:
195: $F \leftarrow D_{q_o}.\texttt{get}(q_s^{u,c})$ ▷ Update or create an entry for $q_s^{u,c}$
196: **if** $F = \epsilon$ **then**
197: $F \leftarrow \emptyset$
198: $F \leftarrow F \cup \left\{(t_c, \vec{\Delta}_m, \varphi_{\text{setup}})\right\}$
199: $D_{q_o}.\texttt{put}(q_s^{u,c}, F)$
200:
201: **for all** $q \in \{j \mid (m = (\text{success}; id, A_i, T_c) \text{ on edge } j) \in L_{\text{rSUCCESS}}\}$ **do**
202: $F \leftarrow D_{q_o}.\texttt{get}(q)$
203: **if** $F \neq \epsilon$ **then**
204: **send** $(\text{commit}; \nu, A_l, F)$ **on edge** q
205: **else**
206: **send** $(\text{commit}; \nu, A_l, \emptyset)$ **on edge** q

Algorithm 22 Pseudocode for the implementation of the transaction generation protocol for a node u (part 7/9)

```
207: procedure splitEqually(t, L_d)          ▷ t is the max amount to be divided, L_d is a list of demands
208:     if t ≥ ∑ L_d then
209:         R ← L_d
210:     else
211:         R ← [0]^|L_d|
212:         E ← L_d
213:         c, r ← t / |E|
214:         while c > 0 do
215:             for i ∈ [0, |E|) do
216:                 g ← min(c, E[i])
217:                 R[i] ← R[i] + g
218:                 E[i] ← E[i] − g
219:                 t ← t − g
220:             c, r ← t / |E|
221:
222:         while r > 0 do                   ▷ Divide the remainder among the buckets
223:             for i ∈ [0, |E|) do
224:                 g ← min(1, E[i])
225:                 r ← r − g
226:                 R[i] ← R[i] + g
227:                 E[i] ← E[i] − g
228:                 if r = 0 then break
229:     return R
```

Algorithm 23 Pseudocode for the implementation of the transaction generation protocol for a node u (part 8/9)

upon receipt of $m = (\text{commit}; id, A_i, T_{c_{\text{setup}}})$ **on edge** j **do**
  check-action: $Pr_{\nu}(id) \rightarrow$ **disallow**, $\neg awake \rightarrow$ **disallow**
  check-action: $Pr_{r,\text{diff}}(A_i) \rightarrow$ **disallow**

  if $roundState \neq SUC$ **then return**

  $roundState \leftarrow COM$
  if $j \notin \{q_e^{u,c} \mid (t_c, \left[q_e^{u,c}, q_s^{u,c}, \vec{\Delta}_m, completed\right]) \in D_{t_c}^u\}$ **then**
    $L_{\text{rCOMMIT}} \leftarrow L_{\text{rCOMMIT}} * m$

    if $|L_{\text{rCOMMIT}}| = |L_{\text{rREQUEST}}|$ **then**
      $D_{q_o} \leftarrow \emptyset$ ▷ Map of $q \rightarrow \left\{(t_c, \vec{\Delta}_{\text{EX}}, \varphi_{\text{setup}})\right\}$
      for all $T_{c_{\text{setup}}}^i \in \{T_{c_{\text{setup}}} \mid (m = (\text{commit}; id, A_i, T_{c_{\text{setup}}}) \text{ on edge } j) \in L_{\text{rCOMMIT}}\}$ **do**
        for all $(t_c, \vec{\Delta}_{\text{EX}}, \varphi_{\text{setup}}) \in T_{c_{\text{setup}}}^i$ **do**
          $[\vec{\Delta}, q_o^{u,c}, q_i^{u,c}] \leftarrow D_{t_c}^{\neg u}.\texttt{get}(t_c)$
          $D_{t_c}^{\neg u}.\texttt{put}(t_c, [\vec{\Delta}_{\text{EX}}, q_o^{u,c}, j])$ ▷ Store the incoming edge of the cycle

          $\texttt{pay}_{\text{cond}}(u, \texttt{Opp}(q_o^{u,c}, u), \vec{\Delta}_{\text{EX}}, \varphi_{\text{setup}})$

          $F \leftarrow D_{q_o}.\texttt{get}(q_o^{u,c})$ ▷ Update or create an entry for q_o
          if $F = \epsilon$ **then**
            $F \leftarrow \emptyset$
          $F \leftarrow F \cup \left\{(t_c, \vec{\Delta}_{\text{EX}}, \varphi_{\text{setup}})\right\}$
          $D_{q_o}.\texttt{put}(q_o^{u,c}, F)$

      $\texttt{commit}(D_{q_o})$
    else
      $L_{\text{rCycleCOMMIT}} \leftarrow L_{\text{rCycleCOMMIT}} * m$

      for all $(t_c, \vec{\Delta}_{\text{EX}}, \varphi_{\text{setup}}) \in T_{c_{\text{setup}}}$ **do**
        $\varphi_{\text{exec}} \leftarrow D_{t_c, \varphi_{\text{exec}}}.\texttt{get}(t_c)$
        $\texttt{pay}_{\text{exec}}(u, \texttt{Opp}(j, u), \vec{\Delta}_{\text{EX}}, \varphi_{\text{exec}})$
        send $(\text{exec}; \nu, A_l, t_c, \varphi_{\text{exec}})$ **on edge** j

  if $|L_{\text{rCOMMIT}}| + |L_{\text{rCycleCOMMIT}}| = |L_{\text{rREQUEST}}| + |D_{t_c}^u|$ **then**
    $roundState \leftarrow \text{EXEC}$
    $\texttt{checkIfExecutionSafe}()$

Algorithm 24 Pseudocode for the implementation of the transaction generation protocol for a node u (part 9/9)

268: **procedure** CHECKIFEXECUTIONSAFE
269: **if** $|D_{t_c}^{\neg u}| = 0$ **then**
270: $execSafe \leftarrow$ true
271: **if** $Pr_{\text{nextRound}}()$ **then**
272: `nextRound`()
273:
274: **upon receipt of** $m = (\text{exec};id, A_i, t_c, \varphi_{\text{exec}})$ **on edge** j **do**
275: **check-action:** $Pr_{\nu}(id) \rightarrow$ **disallow**, $\neg awake \rightarrow$ **return**
276: **check-action:** $Pr_{r,\text{diff}}(A_i) \rightarrow$ **return** , $roundState = \text{COM} \rightarrow$ **defer**
277:
278: **if** $roundState = \text{EXEC} \wedge t_c \in D_{t_c}^{\neg u}$ **then**
279: $[\vec{\Delta}, q_o^{u,c}, q_i^{u,c}] \leftarrow D_{t_c}^{\neg u}.\texttt{get}(t_c)$
280: $\texttt{pay}_{\text{exec}}(u, \texttt{Opp}(q_i^{u,c}, u), \vec{\Delta}, \varphi_{\text{exec}})$
281: $D_{t_c}^{\neg u}.\texttt{remove}(t_c)$
282:
283: **send** $(\text{exec};\nu, A_l, t_c, \varphi_{\text{exec}})$ **on edge** $q_i^{u,c}$
284: `checkIfExecutionSafe`()
285:
286: **upon receipt of** $m = (\text{next\_round};id, A_i)$ **on edge** j **do**
287: **check-action:** Not finished with participant discovery - **defer**, $Pr_{\nu}(id)$ - **disallow**
288: **check-action:** $Pr_{r,\text{early}}(A_i)$ - **return** , $Pr_{r,\text{future}}(A_i)$ - **defer**
289:
290: **if** $\neg forwardedNextRoundMes$ **then**
291: $forwardedNextRoundMes \leftarrow$ true
292: **for all** $e \in E_a$ **do**
293: **send** $(\text{next\_round};\nu, A_l)$ **on edge** e
294:
295: **if** $Pr_{\text{nextRound}}()$ **then**
296: `nextRound`()
297:

Appendix C

Additional dynamic simulations

In this appendix, we provide additional simulation results for the dynamic simulation as described in Section 5.3.4. The additional simulations consist of five scenarios where the Gini coefficient trigger point $\phi \in \{0.1, 0.2, 0.225, 0.25, 0.3\}$.

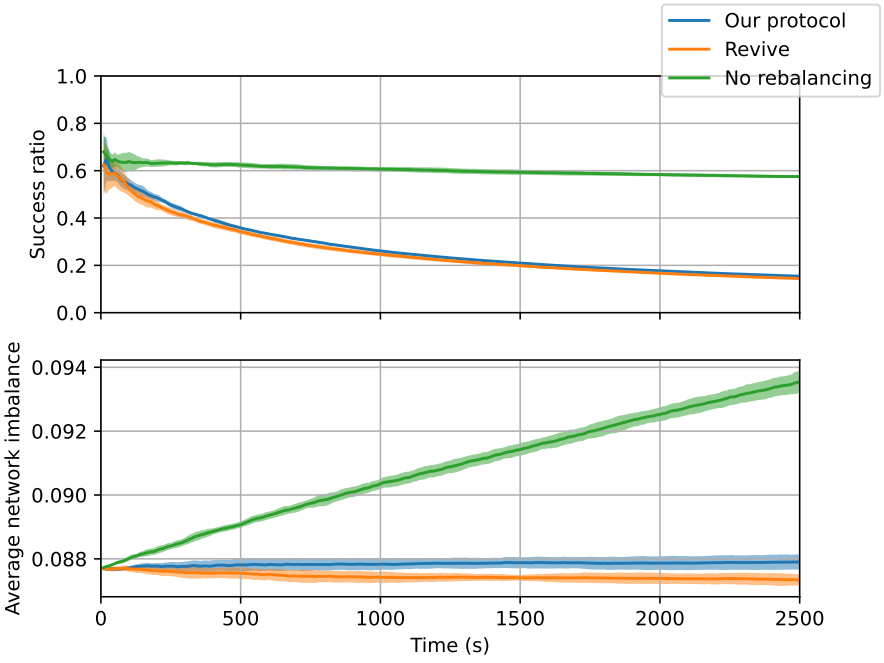


Figure C.1: Comparison of our transaction generation protocol, Revive and no rebalancing in a dynamic simulation of a PCN. The simulation was ran 10 times with $h_c = 3$, $I_m = 3$, $\rho = 20\,\%$, $\phi = 0.1$ and different seeds on $G_{\text{Lightning}}$. The solid lines represent the mean of the simulations and the confidence bounds represent one standard deviation from the mean.

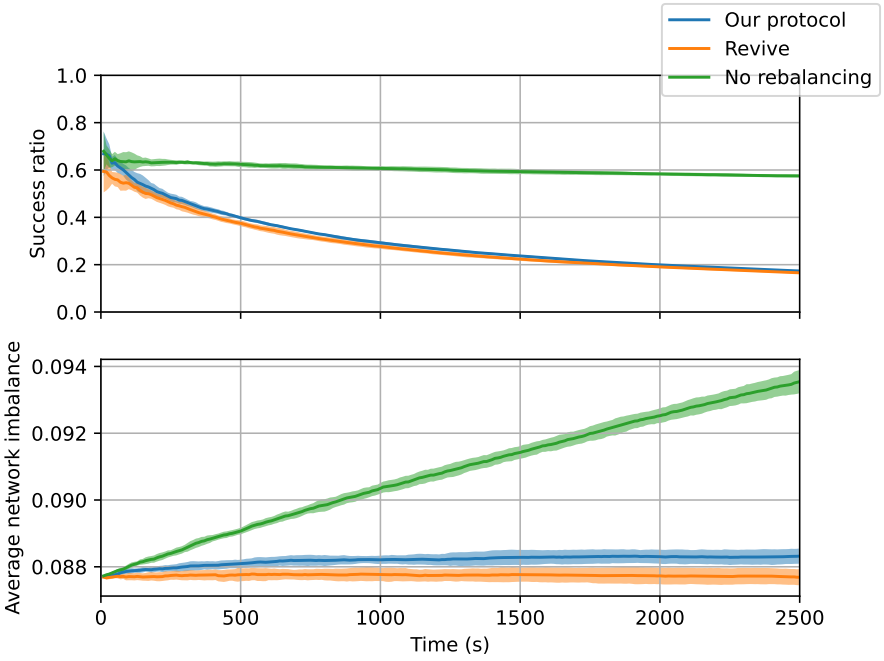


Figure C.2: Comparison of our transaction generation protocol, Revive and no rebalancing in a dynamic simulation of a PCN. The simulation was ran 10 times with $h_c = 3$, $I_m = 3$, $\rho = 20\,\%$, $\phi = 0.2$ and different seeds on $G_{\text{Lightning}}$.

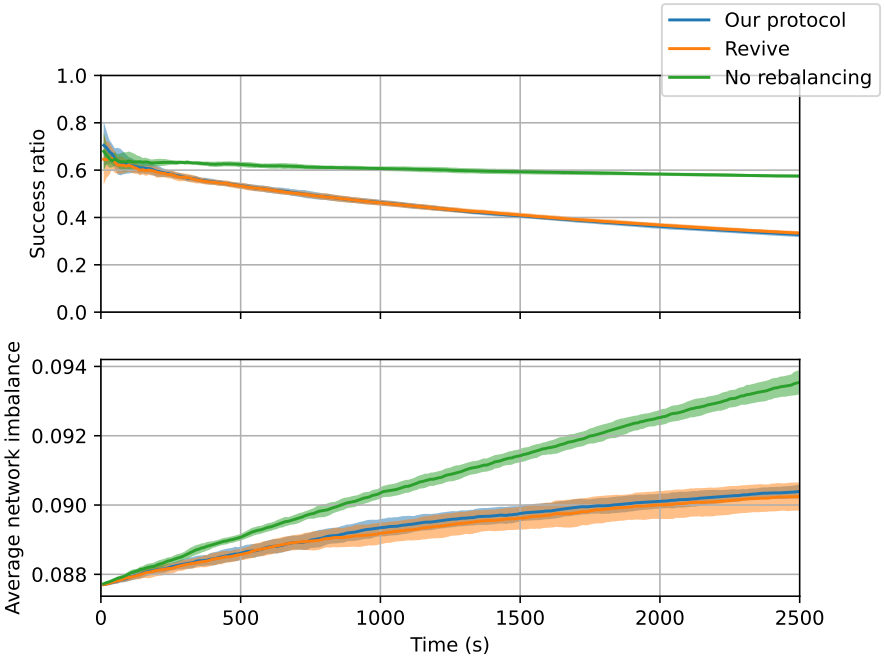


Figure C.3: Comparison of our transaction generation protocol, Revive and no rebalancing in a dynamic simulation of a PCN. The simulation was ran 10 times with $h_c = 3$, $I_m = 3$, $\rho = 20\,\%$, $\phi = 0.225$ and different seeds on $G_{\text{Lightning}}$.

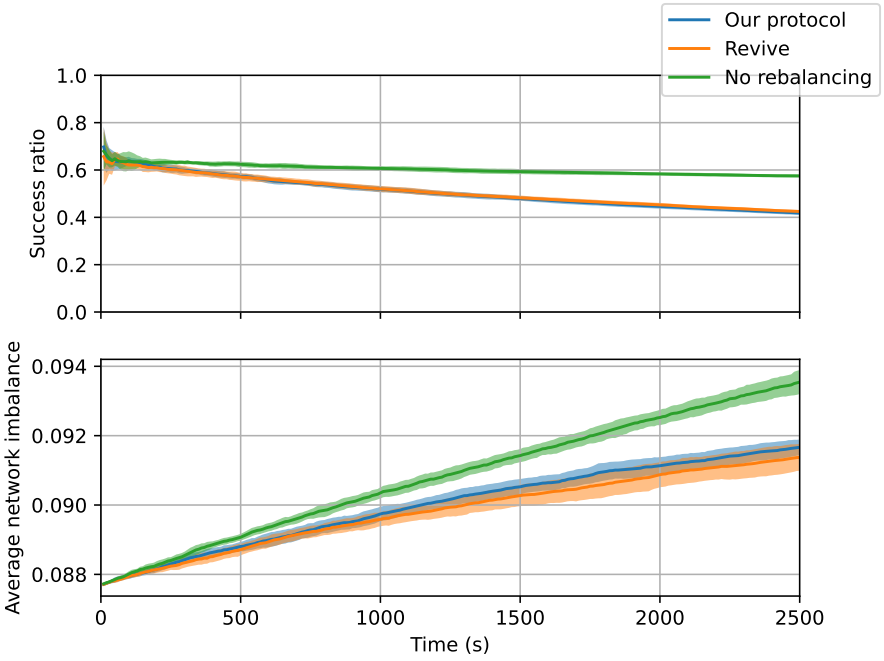


Figure C.4: Comparison of our transaction generation protocol, Revive and no rebalancing in a dynamic simulation of a PCN. The simulation was ran 10 times with $h_c = 3$, $I_m = 3$, $\rho = 20\,\%$, $\phi = 0.25$ and different seeds on $G_{\text{Lightning}}$.

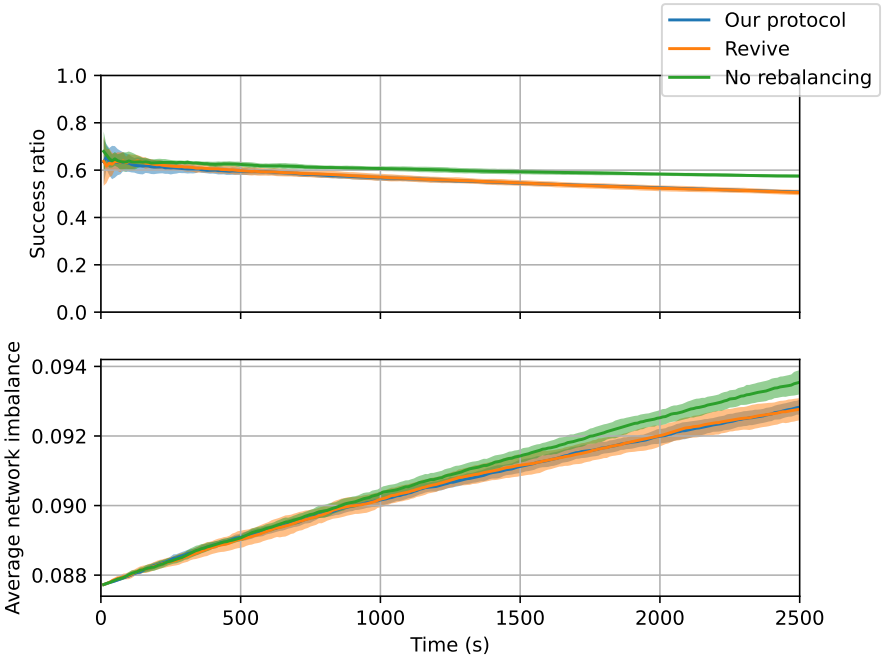


Figure C.5: Comparison of our transaction generation protocol, Revive and no rebalancing in a dynamic simulation of a PCN. The simulation was ran 10 times with $h_c = 3$, $I_m = 3$, $\rho = 20\,\%$, $\phi = 0.3$ and different seeds on $G_{\text{Lightning}}$.