SoftwareX 15 (2021) 100717

Contents lists available at ScienceDirect

SoftwareX

journal homepage: www.elsevier.com/locate/softx

Original software publication

CLoTH: A Lightning Network Simulator

Marco Conoscenti *, Antonio Vetrò, Juan Carlos De Martin

Nexa Center for Internet & Society (DAUIN), Politecnico di Torino, Corso Duca Degli Abruzzi, 24, Torino, Italy

ARTICLE INFO

Article history:
Received 23 January 2021
Received in revised form 14 May 2021
Accepted 19 May 2021

Keywords:
Lightning Network
Bitcoin
Blockchain
Payment-channel networks
Scalability
Simulator

ABSTRACT

Payment-channel networks are one of the most promising solution to the well-known issue of blockchain scalability. In this work we present CLoTH, a simulator of the Lightning Network — the mainstream payment-channel network, used in Bitcoin. CLoTH simulates the execution of payments in a payment-channel network and produces performance measures such as the probability of payment success and the average payment time. To the best of our knowledge, CLoTH is the only simulator that faithfully reproduces the Lightning Network code functions, and this ensures the reliability of simulation results. In this work we provide a detailed description of the new, refactored, publicly-usable version of CLoTH, and we show simulations on the multi-path-payment feature, a recent Lightning Network feature that aims to minimize payment failures.

Code metadata

Current code version	v1.1-beta
Permanent link to code/repository used for this code version	https://github.com/ElsevierSoftwareX/SOFTX-D-21-00019
Code ocean compute capsule	https://codeocean.com/capsule/1325800/tree
Legal code license	GNU General public license v3.0
Code versioning system used	git
Software code languages, tools, and services used	C, python, bash
Compilation requirements, operating environments & dependencies	Linux operating system
If available link to developer documentation/manual	https://github.com/marcono/cloth/blob/master/README.md
Support email for questions	marco.conoscenti@polito.it

1. Introduction

Blockchain-based cryptocurrencies cannot scale [1–3]: the transaction throughput is capped by a limit imposed to the blockchain growth, which aims to keep it small in size.

Payment-channel networks (PCNs) are one of the most promising and investigated solutions to the issue of blockchain scalabilty [4–11]. PCNs enable off-chain payments, i.e., payments that are not required to be registered on the blockchain and are not subject to its throughput limit. A payment channel is a bidirectional channel that allows two parties to exchange off-chain payments. A PCN is a network where off-chain payments are routed, so that parties not directly connected by a channel can exchange off-chain payments. In principle, PCNs are designed to work in a trustless way: a party does not lose its funds even if the other parties are not trustworthy and misbehave.

The Lightning Network (LN) [4], which is built on top of the Bitcoin blockchain, is the most used, developed and researched PCN [12–16]. At the time of writing, there are around 15 thousands of nodes and 36 thousands of channels, and more than one thousand of bitcoins (equivalent to 33 billions of dollars) is allocated in the LN.

The LN, however, presents issues that deserve thorough investigations. First, payment channels are characterized by a limited economic capacity, thus capping the amount of payments that can be exchanged on the LN. Second, payment channels are subject to unbalancing, a situation in which one of the channel directions becomes unusable because of lack of funds. Third, offline or malicious nodes can cause relevant damages, such as long-lasting locks of payments and increased payment time.

We developed CLoTH[1] [17], a simulator of the Lightning Network and PCNs, to study capabilities and limitations of such

* Corresponding author.
E-mail address: marco.conoscenti@polito.it (Marco Conoscenti).

[1] The name "CLoTH" is a play on words that contains the reverse of "HTLC", the main building block of the LN, explained in the next sections.

https://doi.org/10.1016/j.softx.2021.100717

networks. The originality of CLoTH is that it faithfully reproduces the LN code functions, and this ensures the reliability of the simulation results. CLoTH can serve for several purposes, such as test of new functionalities of PCNs before implementing them, simulation of attack scenarios, study of the scalability of PCNs.

As input, CLoTH takes a PCN and a list of payments. It runs a discrete-event simulation, which simulates the execution of the input payments on the input network. It produces statistical performance measures, such as probability of payment success and average payment time.

We already published a description of a previous alpha-version of CLoTH [18]. In the meantime we updated CLoTH to reflect the relevant modifications of the LN protocol. Moreover, we radically refactored the simulator in order to make it easy to use by anyone who aims to systematically analyze PCNs. Therefore, CLoTH is now beta, it is public (while in previous works we made it available only for reproducibility), and in this work we provide an exhaustive explanation of this new version of the simulator.

In addition, we show simulations on multi-path payments (MPP), a feature that consists in splitting large payments into multiple small ones. Simulation results prove that this feature reduces the probability that payments fail for the absence of a route of channels. These simulations are presented also as a showcase to illustrate and explain the functioning of the simulator.

The paper is structured as follows. In Section 2, we present the related work focusing on other LN simulators. In 3, we provide the background on the functioning of the LN. We illustrate CLoTH in detail in Section 4. In Section 5, we show and discuss simulation results on MPP, providing also an illustrative example on the usage of CLoTH. Finally, we conclude our work and discuss the impact of the simulator in Section 6.

2. Related work

The Million Channels Project [19] is a simulator of the LN designed by one of the LN developer, Rusty Russel. The main purpose of this simulator is to study the ability to scale of the LN. The simulator is able to create large networks, which are an accurate evolution of the current LN topology.

In [20], the authors propose and evaluate a routing protocol for PCNs by simulations. The authors of the Flare routing protocol [10] run simulations with 100,000 nodes to study the performance of their proposed protocol. In [21] Piatkivskyi et al. developed a LN simulator to evaluate the approach of splitting large payments into small ones. Their simulator is a multi-agent discrete-event simulator built for general purpose PCN simulations. In [22], Ruozhou Yu et al. implemented a simulator to evaluate a payment routing mechanism called CoinExpress. Their simulator is a PCN simulator based on the network simulator `ns-3`. Stasi et al. [23] developed a simulator to evaluate some improvements to the LN protocol. Finally, Reynolds [24] developed ocalm code for basic simulations on LN.

The key feature of CLoTH is that it accurately reproduces the code of the LN (specifically, the functions implementing routing and the mechanism of payment exchange — called HTLC). This ensures the validity of the simulation results produced by CLoTH, and it makes CLoTH unique and different from the other PCN simulators that we found in literature.

3. Background: the Lightning Network

The LN is the mainstream PCN, built on top of the Bitcoin blockchain, to enable unbounded off-chain payments. The LN protocol specifies how to open and manage payment channels and how to route off-chain payments in a network of payment channels. In the following, such specifications are detailed.

3.1. Payment channel

Let us consider an example in which Alice and Bob open a payment channel, Alice allocates 0.5 BTC in the channel and Bob allocates 0.5 BTC in the channel.

To open a payment channel in the LN, it is necessary to make a transaction in the Bitcoin blockchain, called *funding transaction*. Therefore, Alice and Bob create the funding transaction, where each of them inserts 0.5 BTC — i.e., the funds they want to allocate in the channel. Once the transaction is registered in the blockchain, the channel is considered open. The initial Alice's *balance* in the channel is 0.5 BTC and the initial Bob's balance in the channel is 0.5 BTC. The total *channel capacity* is the sum of the balances — in the example, 1 BTC.

Once the channel is open, the two parties can exchange off-chain payments. To do so, they update the state of their balances. For example, if Alice wants to transfer 0.1 BTC to Bob, Alice decreases its balance by 0.1 BTC and Bob increases its balance by 0.1 BTC. Therefore, at the end of the payment, Alice's balance is 0.4 BTC and Bob's balance is 0.6 BTC.

If the two parties want to close the channel, they have to make another transaction in the Bitcoin blockchain, called *commitment transaction*. This transaction returns the bitcoins of the channel to their respective owners, according to the last state of the balances in the channel. In the example above, the commitment transaction returns 0.4 BTC to Alice and 0.6 BTC to Bob.

3.2. Network of payment channels

The LN allows parties not directly connected by a payment channel to exchange off-chain payments. In this case, the payment is routed across multiple channels that connect the payment sender and the payment receiver.

In LN the exchange of a payment across multiple channels is done via a specific contract called *HTLC* (Hashed Timelock Contract). The HTLC ensures trustlessness: a party involved in a payment route is guaranteed not to lose money, even in case the other parties in the route misbehave. The HTLC implements off-chain conditional payments in a payment channel. For example, when Alice establishes an HTLC of value 0.1 BTC in the channel with Bob, it means that Alice will pay Bob 0.1 BTC if Bob shows a certain value $\mathcal{R}$ (called *preimage*). Otherwise, if Bob does not show $\mathcal{R}$ within a certain timeout, the payment does not take place.

It is called hashed timelock contract because it contains both an hash and a *timelock*. The first is the hash of the preimage $\mathcal{R}$ used to verify that a party knows $\mathcal{R}$ and therefore that the contract can be fulfilled (i.e., the payment can occur). The timelock is the Bitcoin implementation of a timeout: if the timeout expires, the contract is failed and the payment does not occur.

By means of the HTLC, it is possible to exchange a payment across multiple channels. Let us consider the example in which Alice wants to pay 0.1 BTC to David but she has not a direct channel with David. However, Alice has a channel with Bob, Bob a channel with Carol and Carol a channel with David. Alice can use all these channels to send 0.1 BTC to David.

To do so, an HTLC is established in each channel traversed by the payment. All the HTLCs require the same preimage $\mathcal{R}$ to be fulfilled. First, David generates $\mathcal{R}$ and gives Alice the hash of $\mathcal{R}$. After that, HTLCs containing the hash of $\mathcal{R}$ are established in all the involved channels. When the HTLCs have been established in all the channels, David shows $\mathcal{R}$ to Carol and Carol pays 0.1 BTC to David; Carol shows $\mathcal{R}$ to Bob and Bob pays 0.1 BTC to Carol; Bob shows $\mathcal{R}$ to Alice and Alice pays 0.1 BTC to Bob. At the end, 0.1 BTC was transferred from Alice to David and the balances in the channels were updated accordingly.

Finally, it is important to mention that nodes in the LN take some fees for forwarding payments. In the example above, Alice adds some fees to the amount of the payment, to be paid to Bob and Carol.

4. CLoTH: Software description

CLoTH is a PCN simulator written in C. As input, it takes a PCN and a list of payments. Then, it simulates the execution of the input payments in the input network, by running a discrete-event mapping of the LN code functions. As output, it produces payment-related performance measures (such as probability of payment success and average payment time).

The execution flow of the simulator consists of three phases: network and payment generation, simulation, and production of performance measures. In Sections 4.1, 4.2, 4.3, we describe the three phases of the simulator in details, while in Section 4.4 we illustrate the changes of this new version of CLoTH with respect to the previous one.

4.1. First phase: Network and payment generation

4.1.1. Data structures

Fig. 1 shows the main attributes of the data structures representing payments and PCNs in CLoTH. A `channel` connects two `nodes` (each one represented by an ID) and has a certain economic capacity. In addition, since a channel is bidirectional, namely, payments can traverse it both from `node1` to `node2` and from `node2` to `node1`, it contains two `edges`, each one representing a direction of the channel. An `edge` contains: the ID of the `channel` the edge belongs to; the available balance in the direction represented by the edge; and the *policies* it applies to the payments that flows through the edge. These policies are: base and proportional fee, which constitute the fee required for forwarding a payment in the direction of the edge (proportional fee depends on the payment amount, while base fee is constant); the timelock of the HTLCs established in the direction of the edge; and the minimum value allowed for payments forwarded in the direction of the edge. A `payment` is described by a sender, a receiver, the payment amount and the payment start time.

4.1.2. Input modes

The simulator provides two possible input modes for populating the data structures:

1. Random generation. In this input mode, nodes, edges, channels and payments are randomly generated basing on a few input parameters, such as the number of channel per node, the average channel capacity, the average payment amount.
2. Read from files. CSV files are provided to the simulator, where the exact attributes of each node, channel, edge and payment are specified. This second input mode allows the simulation of payments on real PCNs: for instance, it is possible to provide the current nodes, channels and edges of the LN and simulate payments on this network.

It is possible to mix the two input modes, e.g., nodes, channels and edges are read from files and payments are randomly generated.

In the case of the first input mode, to randomly generate the topology of the PCN the *scale-free* network model is used. The degree distribution of this network model follows a power law, which is commonly observed in network theory [25]. This allows the generation of a realistic network starting from a real network topology. Specifically, in CLoTH the random network is generated starting from the existing topology of the LN. New nodes are added to this existing network, and the probability of connecting nodes is directly proportional to their degree: i.e., the higher the number of already open channels of a node, the higher the probability that new nodes will open a channel with that node. In this way, the random network generated realistically reproduces the LN. Table 1 shows the input parameters of the first input mode.

Table 1
CLoTH input parameters.

Name	Description
`n_new_nodes`	The number of nodes of the random network, added to the ones already present in the LN topology (which serves as model for the random network).
`n_channels`	The number of channels for each one of the node specified in the previous parameter.
`capacity`	The average channel capacity expressed in satoshis[a] (the mean of a uniform gaussian distribution).
`faulty_probability`	The probability that a node is faulty when asked to forward a payment.
`payment_rate`	The average number of payments per second. In particular, the payment inter-arrival time is modeled as a negative exponential random distribution.
`n_payments`	The total number of payments to be simulated.
`payment_amount`	The average payment amount expressed in satoshis (the mean of a uniform gaussian distribution).
`mpp`	A 0/1 value that indicates whether to activate or not the multi-path-payment feature, which consists in splitting a large payment in small ones to maximize the chances of success

[a] 1 satoshi corresponds to 10^{-8} bitcoin.

4.1.3. Multithread execution

Once the input network and payments have been defined and before running the simulation, the simulator launches parallel threads. Each of this thread runs Dijkstra's algorithm to find an initial path for each of the payments. In fact, Dijkstra's algorithm is the most time-consuming task of the simulator, and executing it in parallel reduces the run time.

4.2. Second phase: Simulation

CLoTH is a discrete-event simulator. Events are extracted from a queue (implemented by a heap) where they are ordered according to their occurrence time. When an event is extracted, the simulation time (which is discrete) is advanced to the occurrence time of the event and the event is processed by a function.

4.2.1. Events

In CLoTH, an event always refers to a payment and indicates a processing phase of the payment: for example, the event `find_path` indicates that a path for the payment has to be found. Fig. 2 shows the state diagram of simulator events. It represents the flow of the simulation phase and it is as follows.

First, a path for a payment is searched using Dijkstra's algorithm specifically adapted to a PCN (`find_path` event). If a path is not found because channel capacities are lower than the payment amount, the payment is split in two sub-payments, whose amounts are half the original payment amount (this is the multi-path-payment feature). If paths for these payments are not found, the payment is definitely failed. Instead, If a path is found, the payment sender sends the payment to the first hop of the path (`send_payment` event). Each hop of the path forwards the payment up to the payment receiver (`forward_payment` event). If there are no errors, the payment arrives to the receiver (`receive_payment` event). Then, each hop propagates the success result of the payment back in the path (`forward_success` event), and at the end the payment sender receives the success result (`receive_success` event). If instead an error occurred (e.g., there is not enough balance in a channel to forward the

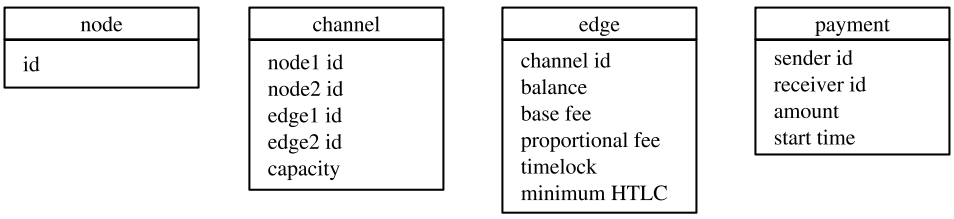


Fig. 1. CLoTH data structures.

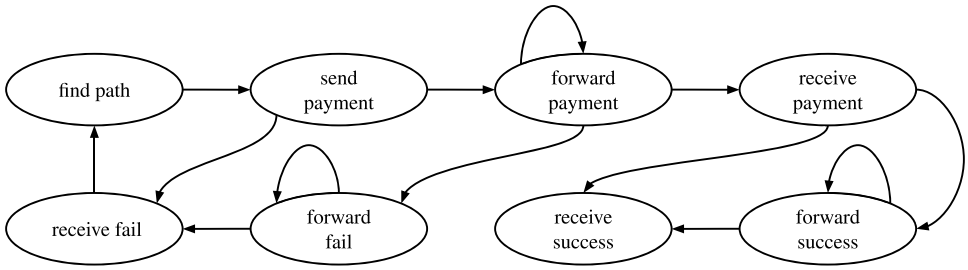


Fig. 2. CLoTH events state diagram.

payment), each hop propagates the fail result of the payment back to the sender (`forward_fail` event), which at the end receives the fail result (`receive_fail` event). Failed payments can be re-attempted, but the LN imposes a timeout of 60 seconds: if a payment does not succeed within that time, it fails definitely.[2]

4.2.2. Functions

Each event is processed by a function of the same name of the event. The functions of CLoTH reproduce the functions of the LN code. In particular, `lnd` is taken as a reference implementation, and the current version of CLoTH is based on `lnd-v0.10.0-beta`. `lnd` is the Golang implementation of LN and, as well as the other implementations (i.e., `c-lightning`[3] and `eclair`[4]), it fully conforms to the so-called *Basis of Lightning Technology* (BOLT), the LN specifications.[5] The reason of choosing `lnd` is that it is the most documented of the LN implementations in terms of comments to the code.

In particular, CLoTH simulates two modules of `lnd`: `routing` and `htlcswitch`. The `routing` module is in charge of finding the paths of payments, and in CLoTH it is simulated by the `find_path` function (plus the functions in CLoTH file `routing.c` internally called by `find_path`). The `htlcswitch` module implements the exchange of HTLC messages to manage sending, forwarding and receiving of a payment, and it is simulated by all the remaining functions in file `htlc.c` (except for `find_path`). Specifically, the `htlcswitch` module manages the exchange of three main messages: `UpdateAddHTLC`, `UpdateFulfillHTLC`, `UpdateFailHTLC`. `UpdateAddHTLC` is a message used to establish an HTLC in a channel, `UpdateFailHTLC` and `UpdateFulfillHTLC` to fail and fulfill an HTLC previously established.

Table 2 shows the mapping between simulation functions and the `lnd` functions. The table shows, for a function of CLoTH, the functions of `lnd` simulated, and it indicates also the type of HTLC message processed and the node processing it (since the `lnd` functions have different behaviors depending on the message and the node).

The mapping between CLoTH and `lnd` functions represents the originality of CLoTH: to the best of our knowledge, CLoTH is the unique simulator that accurately simulates the LN code.

In Appendix A we show the listings of the functions and we explain them in detail.

4.3. Third phase: Performance measures production

At the end of the simulation, CLoTH outputs some information on each payment in file `payments_output.csv`, namely, the payment start and end time, the result of the payment (success or not), the number of attempts, the route traversed by the payment, the fee of the payment.

To convert this per-payment information into statistical performance measures, we use the batch means method [26]. This method produces measures that are not influenced by the initial transient state of the simulation, where the system is not stable. The batch means method consists in removing the initial transient state and dividing a simulation run into multiple batches, which are statistically independent among each other. The performance measures are zeroed and re-computed at each batch. Each final performance measure is the statistical mean of that measure over the batches and is also characterized by variance and 95% confidence interval.

To ensure the validity of the batch means, it is necessary that the duration of the simulation run is greater than the maximum payment time (which in the case of the LN corresponds to 60 s, i.e., the payment timeout).

Table 3 shows the performance measures generated by CLoTH.

4.4. The new version of CLoTH

We already published previous works on CLoTH [18,27]. However, in the meantime the protocol of LN underwent important modifications, therefore, the code of CLoTH was updated to reflect the new version of the protocol. In addition, we performed an extensive refactoring, to make the simulator public and easy

[2] This timeout has not to be confused with the timelock discussed in Section 3. The timelock is a timeout set in an HTLC contract established in a single payment channel, to unlock the funds in case the HTLC is not fulfilled. The 60-seconds timeout, instead, refers to the entire life cycle of a payment: if the payment is sent back to the sender because some errors occurred, and if more than 60 s has elapsed since the first attempt of the payment, the sender definitely fails the payment.

[3] https://github.com/ElementsProject/lightning.

[4] https://github.com/ACINQ/eclair.

[5] https://github.com/lightningnetwork/lightning-rfc/blob/master/00-introduction.md.

Table 2
Mapping between CLoTH functions and `lnd-v0.10.0-beta` functions.

CLoTH function	lnd functions	Node	Message
`find_path`	`sendpayment`, `resumePayment`, `applyPaymentResult`, `RequestRoute`	Sender	–
`send_payment`	`handleLocalDispatch`, `handleDownStreamPkt`, `SendMessage`	Sender	`UpdateAddHTLC`
`forward_payment`	`handleUpstreamMsg`, `processRemoteAdds`, `handlePacketForward`, `handleDownStreamPkt`, `SendMessage`	Hop	`UpdateAddHTLC`
`receive_payment`	`handleUpstreamMsg`, `processExitHop`, `SendMessage`	Receiver	`UpdateAddHTLC`
`forward_success`	`handleUpstreamMsg`, `processRemoteSettleFails`, `handlePacketForward`, `handleDownStreamPkt`, `SendMessage`	Hop	`UpdateFulfillHTLC`
`forward_fail`	`handleUpstreamMsg`, `processRemoteSettleFails`, `handlePacketForward`, `handleDownStreamPkt`, `SendMessage`	Hop	`UpdateFailHTLC`
`receive_success`	`handleUpstreamMsg`, `processRemoteSettleFails`, `handleLocalResponse`	Sender	`UpdateFulfillHTLC`
`receive_fail`	`handleUpstreamMsg`, `processRemoteSettleFails`, `handleLocalResponse`	Sender	`UpdateFailHTLC`

Table 3
CLoTH performance measures.

Name	Description
`Success`	Probability of payment success.
`FailNoPath`	Probability of payment failure for no path. It occurs when Dijkstra's algorithm is not able to find a path between the payment sender and the payment receiver. This may be due to the fact that channel capacities are lower than the payment amount.
`FailNoBalance`	Probability of payment failure for no balance. It occurs when a node tries to forward the payment to the next node and there is not enough balance in the edge connecting the two nodes.
`FailOfflineNode`	Probability of payment failure for offline nodes.
`FailTimeout`	Probability of payment failure for timeout expiration.
`Time`	Average payment time (only for successful payments).
`Attempts`	Average number of attempts before completing a payment (only for successful payments).
`RouteLength`	Average number of hops in payment routes (only for successful payments).

to use by anyone. Therefore, the present work is intended to provide an exhaustive explanation of this new usable version of the simulator, which is enough different from the previous one to deserve its own publication and which is now completely public (differently w.r.t previous versions, which we made available only for reproducibility). In synthesis, the main changes to previous version of CLoTH are the following:

- Full-code refactoring (elimination of global variables, use of meaningful variable and function names, etc.) and minor bug fixes.
- Implementation of the scale free network model to generate random networks that realistically reproduce the LN.
- Implementation of an input parser which reads the simulation input parameters from a text file. This avoids to use an external library to parse more complex types of file (e.g. JSON), thus reducing the external dependencies of the code.
- Modularization. The organization in files of the simulator code now reflects the different modules: network and payments generation, simulation, and production of performance measures. Files `network.c` and `payments.c` implement network and payments generation, respectively. The discrete-event simulation core engine is implemented in `cloth.c`. The functions of the LN simulated are in `htlc.c` and `routing.c`. The production of performance measures is implemented in a python script named `batch-means.py`. Such a modularization guarantees that any other PCN can be simulated by CLoTH with minimum effort: it is just necessary to replace `htlc.c` and `routing.c` with the logic of the PCN to be simulated.
- Update of the LN code functions simulated, to be compliant to `lnd-v0.10.0-beta` (the previous version of CLoTH was based on `lnd-v0.5.0-beta`). In this regard, the main changes were:
  - A new version of Dijkstra's algorithm in which the distance metric also depends on the probability of successfully forwarding a payment (computed using the results of the previous payments).
  - Functions and data structures that manage the payment results (to be used in Dijkstra's algorithm).
  - The multi-path-payment (MPP) feature.
  - The non-strict-forwarding feature. When a node A forwards a payment to node B, this feature allows node A to use any of its channels with node B, instead of strictly using a specific one.[6]

In Table 4, we show a quantitative description of the changes: we select a few measurements from the analysis of the two versions of the code with the quality management tool SonarCloud[7][8]. The table shows that the overall quality of the code has notably improved, with a drastic reductions of code smells and the removal of duplicated code. Despite the new protocol features added, the dimensions of the codebase are more compact in the new version: this is due to the heavy refactoring described above. Also, both cyclomatic and cognitive complexity has been reduced, making the new version more intelligible.

[6] This feature is currently disabled in CLoTH, as it requires the simulation of the blockchain. The feature will be enabled in future work, when CLoTH will simulate also the blockchain.

[7] Analysis of the old version of CLoTH is available here: https://sonarcloud.io/dashboard?id=marcono_sonar-cloth.

[8] Analysis of the new version of CLoTH is available here: https://sonarcloud.io/dashboard?id=marcono_cloth.

Table 4
Quantitative changes between previous and current version of CLoTH.

		CLoTH alpha	CLoTH v1.1-beta
Maintainability	Code smells	919	247
Redundancies in code	Duplicated lines	76	0
	Duplicated blocks	4	0
Size	Lines of code	2796	2229
	Statements	1970	1491
	Functions	99	90
Complexity	Cyclomatic complexity	389	351
	Cognitive complexity	494	386

5. MPP simulations

In this Section we discuss the simulations that aim to show the effect of the multi-path-payment feature implemented in the LN. When a path for a payment is not found because channels capacities are lower than the payment amount, the MPP feature splits the payment into two sub-payments: the amount of each sub-payment is half the amount of the original payment, thus increasing the chances of finding channels able to forward the sub-payments to the receiver. More details on this feature are in Appendix A.1.

5.1. Simulations design

We ran two simulation campaigns using the current version of the simulator: in one the MPP feature is activated, in the other it is not. The remaining parts of the code (e.g., Dijkstra's algorithm of `lnd-v0.10.0-beta`) are exactly the same.

For populating network and payments of these simulations, we used the two different input modes of the simulator: the network was read from CSV files, and the payments were randomly generated using the simulator input parameters.

For what concerns the network, as input to the simulator we provided the real LN: we took a snapshot of nodes and channels of the LN on December 17th, 2020,[9] by launching command `describe-graph` on a `lnd` node. Therefore, nodes and channels of the simulations (together with their attributes: channel capacity, base and proportional fee, minimum HTLC policy, and timelock policy) are exactly the ones of the LN on that date. The initial balances of channels (i.e., the fraction of the channel capacity that each node of the channel owns), instead, were randomly generated in our simulations.[10] The reason is that balances are not publicly available: to guarantee privacy, balances are kept private in the LN. In addition, the probability of faulty nodes was set to zero, as we were not interested in studying the behavior of faulty nodes.

For what concerns the simulated payments, they were randomly generated using the following parameters: average payment rate, total number of payments, and average payment amount. The average payment rate was set to 100 payments per second, because the LN is supposed to support a high payment throughput to let the blockchain scale. The total number of payments was set to 50,000 in order to generate a long-lasting simulation, as required by the batch means analysis.

The average payment amount is the only varying parameter of the simulations: in each of the two campaigns (one with MPP and the other without MPP) we ran 5 simulations, one for each of the following average payment amounts (expressed in satoshis): 10, 10^2, 10^3, 10^4, 10^5. We did not consider the variation of the other input parameters because we were interested in studying the MPP feature only, and it is the payment amount that directly affects this feature: in fact, the higher the payment amount, the higher the probability that the MPP feature is activated and the payment is split in smaller shards.

The rationales of the interval boundaries of the average payment amount were the following. The default minimum HTLC policy in LN is 1 satoshi, meaning that most nodes will refuse payments below 1 satoshi, therefore we decided to set the minimum of the interval to 10 satoshis. We decided to set the maximum to 10^5 satoshis because in preliminary simulations we noticed that, when payment amounts are on average 10^6 satoshis, most of the payments (around 94%) failed, given the current limited capacities of LN channels (as said above, the average channel capacity in LN is 3.4 millions satoshi with a high standard deviation).

5.2. Illustrative example of a simulation run

A simulation is started by running the script `run-simulation.sh`. As input it takes the seed of the simulation (used for the random variables) and the directory where to store the output files.

The simulator reads input parameters from a file called `cloth_input.txt`. In Appendix B a listing of an input file shows the format and the entries this file should contain to run a simulation. In case of simulations with randomly generated network and payments, after the network and payments generation phase, the simulator stores these randomly generated data on files `channels.csv`, `edges.csv`, `nodes.csv` and `payments.csv`. In case instead of reading from files, those files must be provided to the simulator as input. Files `nodes_template.csv`, `channels_template.csv` `edges_template.csv`, `payments_template.csv` in the GitHub repository show the attributes that the files must contain to correctly run a simulation.

After the simulation phase, CLoTH generates files `nodes_output.csv` `channels_output.csv`, `edges_output.csv`, `payments_output.csv`, containing the state of nodes, channels, edges and payments at the end of the simulation. In particular, `payments_output.csv` is used by the batch-means script to compute the final performance measures, stored in `cloth_output.json`. In Appendix C we show the log of an entire simulation.

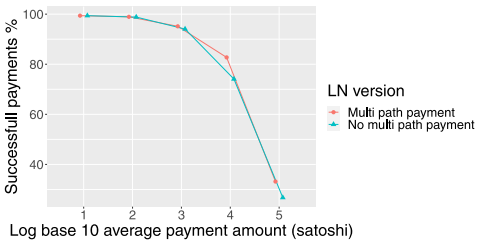
5.3. Simulation results

Figs. 3 show the statistical mean of the probability of success, failure for no path and failure for no balance of payments (see Section 4.3 for the explanation of the performance measures). We do not show confidence intervals because, in any simulation, they resulted almost identical to the mean. The x axis is the log of the average payment amounts. The two curves represent the case with MPP and without MPP. We slightly misaligned the dots to make them distinguishable in case of very similar values. The probability of failures for timeout expiration and offline nodes are not showed because they resulted zero in every simulation.

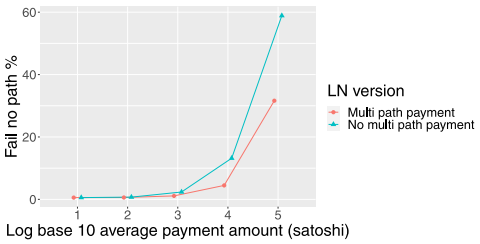
With regards to the failures for no path (Fig. 3b), especially for the highest payment amounts, they were significantly lower with MPP than without MPP: when average payment amounts are 10^5 satoshis, the MPP feature halves the failures for no path (around 30% against 60%). This shows the effectiveness of MPP in reducing failures for no path: large payments are split in two smaller payments, and nominal channel capacities are able to forward the split payments.

[9] At that moment, in the LN there were 6006 active nodes and 30457 active channels, the average channel capacity was around 3.4 millions of satoshis, and its standard deviation was around 9.6 millions of satoshis.

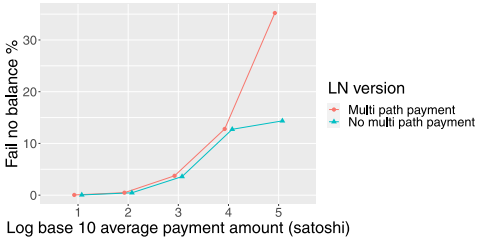
[10] For each channel, a random number uniformly distributed between 0 and 1 is generated and it corresponds to the fraction of the channel capacity that a node of the channel owns as balance.



(a) Probability of payment success.



(b) Probability of payment failure for no path.



(c) Probability of payment failure for no balance.

Fig. 3. MPP simulation results.

Despite this, the probability of payment success only slightly increased with MPP, as Fig. 3a shows. The maximum difference took place at average payment amount set to 10^4, where with MPP the success probability was around 8% higher than without MPP.

The reason of these results is that, although channels have enough capacities, there is not enough balance to forward the split payments. In fact, Fig. 3c shows that probabilities of failure for no balance were higher with MPP: the payments for which – thanks to MPP – a path was found, failed because in a channel of the path there was no balance.

We can conclude that MPP is effective in reducing failures for no path (which is the aim of this feature). However, in the current LN many payments still fail because of no balance. A possible solution is to combine MPP with re-balancing approaches in order to increase the success rate of payments, especially of payments of the order of 10^4 and 10^5 satoshis. For instance, MPP could be combined with the passive rebalancing approach that we proposed in [27] and that we proved effective against channel unbalancing.

6. Impact and conclusions

In this work we described CLoTH, a simulator of the Lightning Network and of payment-channel networks. CLoTH is a faithful reproduction of the LN code functions, therefore it produces reliable performance measures.

For what concerns the impact of the simulator, it can serve for several purposes, to researchers and also in commercial settings. Among the possible uses we mention the following:

- Test of new functionalities. New functionalities of PCNs (such as rebalancing approaches, path-finding algorithms) can be implemented in the simulator to study their effectiveness, before directly implementing them in the PCN software (we studied rebalancing approaches in [27]).
- Analysis of attack scenarios. This makes possible to understand the actual risks of attacks on PCNs and design countermeasures. For example, it is possible to study an attack in which irrational malicious nodes – after establishing HTLCs for payments – intentionally become unresponsive, thus locking the payments for a long time (because of the timelock set in the HTLCs). Another possible attack that can be simulated is the denial of service directed to the most central nodes of the LN, to understand whether the network still works even if the central nodes are not available.
- Estimation of fee revenues. By the simulator it is possible to answer questions that are useful in commercial settings, such as: how many fees can be earned by a node in a specific network position and with a specific set of open channels.
- Analysis of scalability. The simulator is able to simulate large PCNs (as we did in [18]), thus allowing researchers to study in which configurations (e.g. number of channels per node, average channel capacity) such networks can scale.
- Study of specific use cases. It is possible to simulate specific use cases of PCNs, e.g. the service providers scenario that we studied in [27], in which many payments are sent to specific service-provider nodes.

Finally, in future work we plan to conduct some simulations on the attack scenarios described above, and we will keep adding the features implemented in the LN to the simulator as well. We also plan to integrate CLoTH with other simulators: first, with a blockchain simulator [3,28], to study the interactions between blockchain and PCNs; second, with a network simulator (such as `ns-3`) to capture the impact of communication networks on PCNs performance.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Acknowledgment

Marco Conoscenti wishes to thank Dr. Federico Spini, who is one of the first that had the idea to simulate the LN (in the long hot summer of 2017). He supervised the author in the design and development of the first version of CLoTH.

Appendix A. Functions

In this Section we explain the main functions of CLoTH.

A.1. Find path

Listing 1 shows a simplified version of `find_path`, the function that simulates the search of a payment path done by the payment sender. First, the function checks whether more than 60 s has elapsed since the start of the payment: if so, the payment is terminated (because the LN imposes a timeout of 60 s for attempting a payment). Then, the function calls the `lnd` version of Dijkstra's algorithm for searching for a path. Such modified version leverages a specific distance metric, consisting of two parts.

The first part considers timelock and fees imposed by a channel (see Section 4.1.1 for details on timelocks and fees). The `lnd` version of Dijkstra's algorithm tends to find a path that minimizes both fees and timelocks. The second part of the distance metric is a probability based on the results of the previous payments. Each time a LN node sends a payment, it records the result of the payment. In particular, for each channel traversed by the payment, the node stores the amount of the payment and whether it was successfully forwarded in that channel or instead it failed. Having this information on previous payments, the node can calculate the probability that a new payment of a certain amount will be successfully forwarded by a channel. The `lnd` version of Dijkstra's algorithm tends to find paths that maximizes this probability.

If a path for the payment is found, it is sent along the path (see function `send_payment`). It may happen that a path for a payment is not found because channels do not have enough economic capacity to forward the payment. If this happens, if the multi-path-payment feature is activated, and if the payment was not already split, the payment is split in two shards, the amount of each is half of the original payment amount. In case a path is not found for the shards, the payment is definitely failed. Otherwise, the shards are sent along the paths found.

It is important to notice that `payment->amount`, in this and in the other functions, contain also the fees that must be payed to the forwarding nodes.

Listing 1: Find-Path Function.

```
void find_path(payment){
  if(payment->duration > 60000) {
    end_payment(payment);
    return;
  }
  path = dijkstra(payment);
  if(path != NULL){
    generate_event(send_payment, payment);
    return;
  }
  if(!mpp_active || payment->split){
    end_payment(payment);
    return;
    }
  payment->split = 1;
  shard1 = new_payment(payment->amount/2);
  shard2 = new_payment(payment->amount/2);
  path1 = dijkstra(shard1);
  path2 = dijkstra(shard2);
  if(path1 == NULL || path2 == NULL){
    end_payment(payment);
    return;
    }
  generate_event(send_payment, shard1);
  generate_event(send_payment, shard2);

}
```

A.2. Send payment

Listing 2 shows a simplified version of `send_payment`, the function that simulates the sending of a payment by the payment sender. The function first checks whether the next node of the path is offline: this simulates the situation in which a node is offline and cannot forward a payment. The function then checks whether there is sufficient balance in the edge to forward the payment. If the checks passed, the edge balance is decreased and the payment is forwarded to the next node, otherwise the failure is processed by the sender (see function `receive_fail`).

Listing 2: Send-Payment Function.

```
void send_payment(payment){
  if(next_edge_node_offline){
    payment->error.type = OFFLINENODE;
    payment->error.edge = next_edge;
    generate_event(receive_fail, payment);
    return;
    }

  if(next_edge->balance < payment->amount){
    payment->error.type = NOBALANCE;
    payment->error.edge = next_edge;
    generate_event(receive_fail, payment);
    return;
  }
  next_edge->balance -= payment->amount;
  generate_event(forward_payment, payment);
}
```

A.3. Forward payment

Listing 3 shows the simplified version of `forward_payment`. This function simulates the forwarding of a payment by an intermediate node in the route. Similarly to `send_payment`, this function both checks whether the next node is offline and the balance. It also checks whether the payment respects the policies of the forwarding edge (for the sake of simplicity, currently CLoTH does not simulate the case in which these policies are not respected). If the checks passed, the payment is forwarded to the next node in the path or to the payment receiver.

Listing 3: Forward-Payment Function.

```
void forward_payment(payment){
  if(next_edge_node_offline){
    payment->error.type = OFFLINENODE;
    payment->error.edge = next_edge;
    generate_event(forward_fail, payment);
    return;
    }
  can_send_htlc = check_balance_and_policy(next_edge);
  if(!can_send_htlc){
    payment->error.type = NOBALANCE;
    payment->error.edge = next_edge;
    generate_event(forward_fail, payment);
    return;
  }
  next_edge->balance -= payment->amount;
  generate_event(receive_payment, next_event_time);
}
```

A.4. Receive payment

Listing 4 shows a simplified version of `receive_payment`. This function simulates the reception of a payment by the payment receiver. It increases the balance of the edge of the receiver channel and forwards the success result of the payment back to the previous hop of the payment path.

Listing 4: Receive-Payment Function.

```
void receive_payment(payment){
  this_edge->balance += payment->amount;
  generate_event(forward_success, next_event_time);
}
```

A.5. Forward success and forward fail

Listings 5 and 6 represents the simplified functions `forward_success` and `forward_fail`, respectively. These functions simulates the forwarding of the success/fail result of a payment by an intermediate node of the payment path. They adjust the edge balances of the channels involved, according to the payment result, and propagates the result back.

Listing 5: Forward-Success Function.

```
void forward_success(payment){
  this_edge->balance += payment->amount;
  generate_event(receive_success, next_event_time);
}
```

Listing 6: Forward-Fail Function.

```
void forward_fail(payment){
  this_edge->balance -= payment->amount;
  generate_event(receive_fail, next_event_time);
}
```

A.6. Receive success

This function, showed in listing 7, simulates the reception of the success result of a payment by the payment sender. The sender records the result of the payment: for each channel traversed by the payment, it registers the amount of the payment successfully forwarded by the channel. As explained above, this information is used in Dijkstra's algorithm to calculate the probability that a channel successfully forwards a payment. Finally, the payment is terminated.

Listing 7: Receive-Success Function.

```
void receive_success(payment){
  record_payment_success_result(payment);
  end_payment(payment);
}
```

A.7. Receive fail

This function, showed in listing 8, simulates the reception of the fail result of a payment by the payment sender. Also in this case, the sender records the result of the payment: a different information is recorded depending on the reason of the failure, which can be either no sufficient balance in an edge or the presence of an offline node in the route. Finally, the payment is re-attempted, re-executing the function `find_path`.

Listing 8: Receive-Fail Function.

```
void receive_fail(payment){
  this_edge->balance += payment->amount;
  record_payment_fail_result(payment);
  generate_event(find_path, payment);
}
```

Appendix B. Input file

Listing 9 shows an example of the input file to be given as input to CLoTH. In addition to the input parameters, `cloth_input.txt` contains also the filenames of files where to read the nodes, channels and edges (in case `generate_network_from_file` is set to `true`) and where to read payments (in case `generate_payments_from_file` is set to `true`). To correctly run a simulation, the file `cloth_input.txt` must contain exactly these entries.

Listing 9: File `cloth_input.txt`.

```
generate_network_from_file=true
nodes_filename=nodes_ln.csv
channels_filename=channels_ln.csv
edges_filename=edges_ln.csv
n_additional_nodes=
n_channels_per_node=
capacity_per_channel=
faulty_node_probability=0.0
generate_payments_from_file=false
payments_filename=
payment_rate=100
n_payments=50000
average_payment_amount=10
mpp=1
```

Appendix C. Log of a simulation

Listing 10 shows the log of a simulation run by CLoTH: first, network, payments and simulation events are generated; secondly, the threads running Dijkstra's algorithm for each payment are executed; then the discrete-event simulation is run; and finally the batch-means script is executed.

Listing 10: Log of a simulation run.

```
NETWORK INITIALIZATION
PAYMENTS INITIALIZATION
EVENTS INITIALIZATION
INITIAL DIJKSTRA'S THREADS EXECUTION
Time consumed by initial Dijkstra's executions: 2861 s
EXECUTION OF THE SIMULATION
Time consumed by simulation events: 30.024121 s
COMPUTE SIMULATION OUTPUT STATS
Batch length: 15278.0 ms
Total simulated time: 458340.0 ms
SIMULATION OUTPUT STATS SAVED IN <cloth_output.json>
```

References

[1] Sompolinsky Y, Zohar A. Accelerating bitcoin's transaction processing. Citeseer; 2013, URL https://eprint.iacr.org/2013/881.pdf.

[2] Croman K, Decker C, Eyal I, Gencer AE, Juels A, Kosba A, Miller A, Saxena P, Shi E, Sirer EG, et al. On scaling decentralized blockchains. In: International conference on financial cryptography and data security. Springer; 2016, p. 106–25. http://dx.doi.org/10.1007/978-3-662-53357-4_8.

[3] Gervais A, Karame GO, Wüst K, Glykantzis V, Ritzdorf H, Capkun S. On the security and performance of proof of work blockchains. In: Proceedings of the 2016 ACM SIGSAC conference on computer and communications security. ACM; 2016, p. 3–16. http://dx.doi.org/10.1145/2976749.2978341.
[4] Poon J, Dryja T. The bitcoin lightning network: Scalable off-chain instant payments. 2016, URL https://lightning.network/lightning-network-paper.pdf.
[5] Decker C, Wattenhofer R. A fast and scalable payment network with bitcoin duplex micropayment channels. In: Symposium on self-stabilizing systems. Springer; 2015, p. 3–18. http://dx.doi.org/10.1007/978-3-319-21741-3_1.
[6] Miller A, Bentov I, Bakshi S, Kumaresan R, McCorry P. Sprites and state channels: Payment networks that go faster than lightning. In: Goldberg I, Moore T, editors. Financial cryptography and data security. Cham: Springer International Publishing; 2019, p. 508–26. http://dx.doi.org/10.1007/978-3-030-32101-7_30.
[7] Raiden network. URL https://raiden.network/.
[8] Burchert C, Decker C, Wattenhofer R. Scalable funding of Bitcoin micropayment channel networks. R Soc Open Sci 2018;5(8):180089. http://dx.doi.org/10.1098/rsos.180089.
[9] Khalil R, Gervais A. Revive: Rebalancing off-blockchain payment networks. In: Proceedings of the 2017 ACM SIGSAC conference on computer and communications security. ACM; 2017, p. 439–53. http://dx.doi.org/10.1145/3133956.3134033.
[10] Prihodko P, Zhigulin S, Sahno M, Ostrovskiy A, Osuntokun O. Flare: An approach to routing in lightning network. 2016, URL https://bitfury.com/content/downloads/whitepaper_flare_an_approach_to_routing_in_lightning_network_7_7_2016.pdf.
[11] Gudgeon L, Moreno-Sanchez P, Roos S, McCorry P, Gervais A. SoK: Layer-two blockchain protocols. In: International conference on financial cryptography and data security. Springer; 2020, p. 201–26. http://dx.doi.org/10.1007/978-3-030-51280-4_12.
[12] Lin J-H, Primicerio K, Squartini T, Decker C, Tessone CJ. Lightning network: a second path towards centralisation of the Bitcoin economy. New J Phys 2020;22(8):083022. http://dx.doi.org/10.1088/1367-2630/aba062.
[13] Tikhomirov S, Moreno-Sanchez P, Maffei M. A quantitative analysis of security, anonymity and scalability for the lightning network. IACR Cryptol ePrint Arch 2020;2020:303, URL https://eprint.iacr.org/2020/303.pdf.
[14] Malavolta G, Moreno-Sanchez P, Schneidewind C, Kate A, Maffei M. Anonymous multi-hop locks for blockchain scalability and interoperability. In: NDSS. 2019, http://dx.doi.org/10.14722/ndss.2019.23330.
[15] Harris J, Zohar A. Flood and loot: A systemic attack on the lightning network. In: Proceedings of the 2nd ACM conference on advances in financial technologies. AFT '20, New York, NY, USA: Association for Computing Machinery; 2020, p. 202–13. http://dx.doi.org/10.1145/3419614.3423248.
[16] Pérez-Solà C, Ranchal-Pedrosa A, Herrera-Joancomartí J, Navarro-Arribas G, Garcia-Alfaro J. Lockdown: Balance availability attack against lightning network channels. In: International conference on financial cryptography and data security. Springer; 2020, p. 245–63. http://dx.doi.org/10.1007/978-3-030-51280-4_14.
[17] Conoscenti M. Marcono/cloth: cloth-v1.1-beta. 2021, http://dx.doi.org/10.5281/zenodo.4457877.
[18] Conoscenti M, Vetrò A, De Martin JC, Spini F. The cloth simulator for htlc payment networks with introductory lightning network performance results. Information 2018;9(9):223. http://dx.doi.org/10.3390/info9090223.
[19] Russell R, Netti J. Letting a million channels bloom. 2019, URL https://medium.com/blockstream/letting-a-million-channels-bloom-985bdb28660b.
[20] van Schie A. Routing scalable bitcoin payments. 2015, URL http://pub.tik.ee.ethz.ch/students/2015-FS/BA-2015-12.pdf.
[21] Piatkivskyi D, Nowostawski M. Split payments in payment networks. In: Garcia-Alfaro J, Herrera-Joancomartí J, Livraga G, Rios R, editors. Data privacy management, cryptocurrencies and blockchain technology. Cham: Springer International Publishing; 2018, p. 67–75. http://dx.doi.org/10.1007/978-3-030-00305-0_5.
[22] Yu R, Xue G, Kilari VT, Yang D, Tang J. Coinexpress: A fast payment routing mechanism in blockchain-based payment channel networks. In: 2018 27th international conference on computer communication and networks (ICCCN). IEEE; 2018, p. 1–9. http://dx.doi.org/10.1109/ICCCN.2018.8487351.
[23] Di Stasi G, Avallone S, Canonico R, Ventre G. Routing payments on the lightning network. In: 2018 IEEE international conference on internet of things (IThings) and IEEE green computing and communications (GreenCom) and IEEE cyber, physical and social computing (CPSCom) and IEEE smart data (SmartData). 2018, p. 1161–70. http://dx.doi.org/10.1109/Cybermatics_2018.2018.00209.
[24] Reynolds D. Simulating a decentralized lightning network with 10 million users. 2017, URL https://link.medium.com/CmXQ4f0Q8fb.
[25] Barabási A-L, Albert R. Emergence of scaling in random networks. Science 1999;286(5439):509–12. http://dx.doi.org/10.1126/science.286.5439.509.
[26] Jain R. The art of computer systems performance analysis: Techniques for experimental design, measurement, simulation, and modeling. John Wiley & Sons; 1990.
[27] Conoscenti M, Vetrò A, De Martin JC. Hubs, rebalancing and service providers in the lightning network. IEEE Access 2019;7:132828–40. http://dx.doi.org/10.1109/ACCESS.2019.2941448.
[28] Alharby M, van Moorsel A. Blocksim: An extensible simulation tool for blockchain systems. Front Blockchain 2020;3. http://dx.doi.org/10.3389/fbloc.2020.00028.