Flare: An Approach to Routing in Lightning Network

White Paper

Pavel Prihodko* Slava Zhigulin* Mykola Sahno* Aleksei Ostrovskiy*
Olaoluwa Osuntokun†

Jul 07, 2016 (Version 1.0)

Abstract

This paper describes a hybrid routing algorithm Flare, which could be used for payment routing in Lightning Network. The design goal for the algorithm is to ensure that routes can be found as quickly as possible. This is accomplished at the cost of each node proactively gathering information about the Lightning Network topology. The collected information includes both payment channels close to the node in terms of hop distance and paths to beacon nodes, which are close to the node in the node address space. The usage of beacon node serves to supplement a node's local view of the network with randomly selected feeler nodes deeper in the network. The combination of local and beacon nodes allows a node to minimize routing state, while finding routes to any given node with high probability. We perform simulations of the routing algorithm and find it to be scalable to at least 100,000 nodes.

*lightning@bitfury.com, Bitfury Group
†laolu@lightning.network

Bitcoin is the world's most widely used and valuable digital currency [1], which allows anyone to send value without a trusted intermediary or depository. Bitcoin contains an advanced scripting system allowing users to program instructions for funds [2].

Bitcoin aggregates transactions into blocks with the expected time interval of 10 minutes between blocks. Bitcoin payments are widely regarded as practically irreversible after six confirmations (i.e., five additional blocks built on top of the block containing the transaction in question) [3], or about one hour. Micropayments (i.e., payments less than a few cents) could take a long time to get confirmed, with Bitcoin transaction fees rendering such payments unviable. These conditions necessitate the development solutions like overlays, which could combine the advantages of the Bitcoin blockchain (i.e., its security and censorship resistance) with guarantees of near-instant payment processing and affordable micropayments.

The Lightning Network (LN) [4] is one promising overlay solution to the problems mentioned above. LN operates as a network of bidirectional payment channels transferring value out of band, i.e., not recording transactions on the Bitcoin blockchain. LN is designed to be decentralized (i.e., a failure of a single party or a few parties would not render the network inoperable) and trustless (i.e., at no time would the custody of users' funds be delegated to trusted third parties). Security of the network is enforced by blockchain smart contracts using Bitcoin's built-in scripting without creating on-blockchain transactions for individual payments. LN could be deployed for other blockchains using a Bitcoin-like data model with unspent transaction outputs; furthermore, LN could be used for trustless inter-ledger payments. Other payment channel network concepts have emerged (e.g., Stroem [5], Impulse [6], Decker – Wattenhofer channels [7]); we focus our attention on LN as a more popular concept, which embraces the spirit of Bitcoin in its trustless and decentralized nature.

LN payments would not need block confirmations, thus being nearinstant in its normal case of operation. A single payment on LN might involve several payment channels; however, payments are designed to be atomic (either the entire payment succeeds or fails). Thus, LN could be used at retail point-of-sale terminals, with machine-to-machine transactions, or anywhere instant payments are needed. LN could also allow for scalable bitcoin micropayments (e.g., tips on social websites) and micropayment streams (e.g., payment for online videos), therefore contributing to the expansion of the Bitcoin ecosystem.

One of the defining features of LN is the ability to route payments between network users with one or more intermediaries without the need to trust them; i.e., it may not be necessary for parties to create a direct payment channel in order to complete a payment. Correspondingly, a problem of particular importance for LN is payment routing, i.e., finding a path of payment channels, which could route the payment from the sender to the recipient and would be optimal according to certain criteria (e.g., time to complete the payment and/or routing expenses). Without a fully automated solution to payment routing, it could be difficult for LN to establish a foothold.

Previous work. White papers [4, 8, 7] describe mechanisms how to open and close an LN channel, transfer payment and roll it back in the case of the uncooperative or hostile behavior of intermediate nodes. The path routing problem is not covered in the white papers and LN development repositories [9, 10, 11].

The draft implementation offered by Hashplex [12] includes a simple routing mechanism, in which every node floods channel update information to all nodes in the network when opening or closing a channel. Thus, every node accumulates the global map of the network, based on which the node performs routing. This solution could work for smaller networks, but may face scalability issues once the size of the network increases. Additionally, the early nature of the implementation prevents it from being thoroughly tested.

Our contribution. We present an algorithm that could be used to discover and select payment paths between nodes in LN and may be considered as a first step in designing routing for Lightning. The proposed algorithm, which we call Flare, utilizes a trustless source routing scheme that can be scaled to the network size of at least several hundred thousand nodes. We also describe results of simulations performed to test the applicability of the proposed algorithm. Finally, we propose a verification procedure for LN payment channels based on elliptic curve Schnorr signatures.

Structure of the paper.

- We formulate the requirements for a routing algorithm in LN in Section 1.
- In Section 2, we provide an overview of existing approaches to routing in LN and mobile ad hoc networks, which, as we argue, resemble LN in their core assumptions about nodes.
- We provide a hybrid payment routing algorithm, which may be the first step in defining routing in LN in Section 3.
- In Section 4 we provide results of simulations of our proposed algorithm.
- Some possibilities for further algorithmic improvements are briefly explored in Section 5.
- Technical details of LN channel verification are covered in Appendix A.

1 Lightning Network

As we mentioned earlier, a Lightning Network could be built on top of any blockchain that uses Bitcoin-like scripting. In this paper, we will consider the Lightning Network built on top of the Bitcoin blockchain, but most of our results could apply to other blockchain systems without much effort.

As per the LN white paper [4], LN consists of *nodes* (Fig. 1). Each node is identified with a long term public key (the *identity key*) in a certain public key cryptosystem. We assume the secp256k1 elliptic curve would be used (i.e., the main cryptosystem in Bitcoin), which would enable a simple fallback

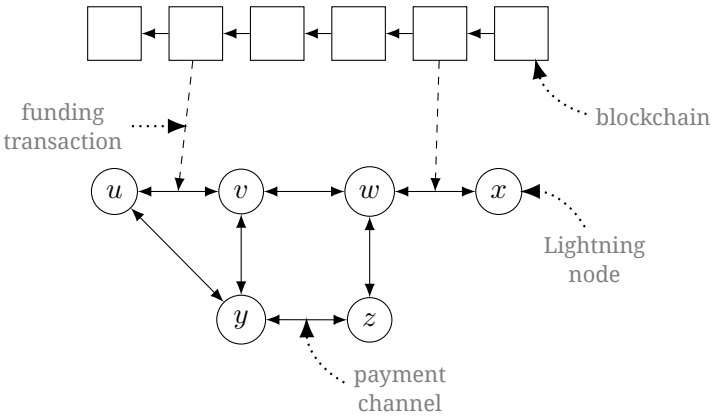


Figure 1: High level view of Lightning Network. Node y can send payment to node x through nodes v and w (provided channels $y - v$, $v - w$, and $w - x$ have enough bitcoin to forward the payment), through z and w, or through u, v and w. If y routes a payment to x through v and w, intermediate nodes could receive fees for routing a payment: an HTLC payment from y to v would be greater than the HTLC payment from v to w by the value of the fee paid to v, and so on.

mechanism: if a payment cannot be routed within LN, it can always be sent to the Bitcoin address uniquely determined by the receiving node's public key.

A Lightning node may operate on top of a full or an SPV (Simplified Payment Verification) Bitcoin node. Each Lightning node can ask any other Lightning node to open a bidirectional *payment channel* by sending a specially formed *funding transaction* to the Bitcoin blockchain. The funding transaction should specify the amount of money that each node locks in the channel and the information needed to identify participating nodes. The total capacity of channel is constant; it is equal to the total amount of money nodes opening the channel put into the funding transaction. On the other hand, the distribution of this value between the nodes in the channel can be changed off-chain by a mutual agreement. The counterparty's cooperation to close the channel is not required: both parties always have an option to unilaterally close the channel, thus ending their relationship.

A node can send a payment to another party in LN by transferring value through the route of channels (we will also use term *path* interchangeably). In order to accomplish this without the risk of losing money and without the need for a trusted third party to gain custodial ownership of transferred funds, LN introduces the hashed time-locked contracts (HTLC) mechanism [4, 8]. Nodes would take *forwarding fees* to transfer payments through their channels, which are expected to be quite small in most cases.

As per the original white paper and the general vision in the developer community, there are several requirements imposed onto LN.

Requirement (Peer-to-peer network). All nodes in the network would be able to act as senders or recipients of payments and as hubs that route payments between other nodes at the same time. At the moment, some members of the community believe [13] that LN could only exist as a hub-and-spoke network [14], as channels created by small nodes would be unlikely to provide enough liquidity for LN

to operate efficiently. While this may become true once LN is launched and high-throughput nodes are present in the network, we think that LN should evolve without imposing the hub-and-spoke design from the start. If routing based on a decentralized network could work in practice, it would benefit the network in terms of decentralization, agility, and stability (as the failure of several nodes would not affect the broader health of the network).

Though we design LN as a peer-to-peer network, we would like it to be able to seamlessly support other topologies if the reality check shows that they work better; i.e., we want to guarantee LN's flexibility to evolve.

Requirement (Source routing scheme). We assume source routing [15] would be used in LN, based on the following reasoning:

- **User privacy:** Source routing together with Tor-like onion data transport allows for strong user privacy where intermediate nodes routing a payment do not know its source or destination.
- **Censorship resistance:** Because of user privacy, intermediate nodes would not be able to censor a payment based on the identity of its sender or receiver.
- **Fee predictability:** Each LN node may require a certain fee to forward the payment, which would be managed by the channel owner. This would lead to the fact that taking different payment routes would require varying expenses from the sender. Hence, it is in the interest of the sender to optimize fees and make the final decision on which route to choose to the recipient (otherwise the sender gets potentially unpredictable expenses, as other nodes are not incentivized to optimize for the cheapest path).
- **Time-lock predictability:** If a payment is not routed in a timely fashion (e.g., because of non-cooperative behavior of intermediate nodes), the sender needs to have enough time to redeem the payment (this possibility is accounted for in the HTLC mechanism introduced in the LN white paper). Thus, time-locks for HTLCs when constructing the payment need to be selected by the sender.

Source routing leads to the requirement that the sender node should be able to collect information on fees and available channel capacity to pick the best route (as well as knowing which nodes are currently online). Thus, an overlay mechanism should exist to enable requesting information about a channel from any of its owners.

Further, to satisfy current Bitcoin users, LN must preserve Bitcoin's main appealing properties.

Requirement (Trustlessness). LN should be trustless in nature, at least in the sense that the safety of one's money should not depend on the behavior of a trusted third party.

This goal is, for the most part, already achieved in the LN white paper with the introduction of the HTLC mechanism.

Requirement (Fast payment processing). One of the goals of LN is to enable fast microtransactions, which imposes a requirement that LN payments should take much less than 1 hour to complete (cf. on-chain Bitcoin transactions). For real-time micropayments or IoT applications to be feasible, even stricter constraints on payment completion need to be imposed: LN payments should take less than one minute and ideally seconds.

In this paper, we achieve fast payment processing at the cost of the proactive collection of network information by each node (see Section 3 for details).

Requirement (Security). As some of nodes may demonstrate Byzantine behaviour (e.g., by lying about fees or routes), routing in LN should have protection mechanisms to prevent or mitigate harm from such actions and guarantee that a sender would not lose his money even in a worst-case scenario. Such mechanisms should include:

- the ability to verify route existence (see Appendix A for the details on a possible implementation of the verification procedure)
- the possibility to roll back the payment in case a node on the payment route provides the wrong information on channel properties (this is already discussed in the LN white paper).

Requirement (Probabilistic route discovery). The routing algorithm should allow finding a route between nodes with a reasonably high probability; it is allowable (but undesirable) to not find an applicable payment route.

Probabilistic routing is permissible since LN has a built-in fallback mechanism. If a sender cannot find a payment route to a recipient (e.g., if they are too far away in terms of the graph distance or in the case of limited or nonexistent connectivity), they can always open a new direct payment channel or send the funds with a Bitcoin transaction on the blockchain. Opening a new channel is expensive in terms of transaction confirmation time and fees, but it is beneficial to the network as it helps improve connectivity of the network graph over time.

Requirement (Moderate operational resources). A significant portion of Bitcoin wallets use simplified payment verification (SPV) [16]. It utilizes a moderate amount of storage and computing power of the hosting computer (making it possible to run on mobile platforms) while preserving relatively strong security. Correspondingly, a Lightning node should be able to function on top of a Bitcoin SPV node, requiring comparable amount of resources to operate.

Requirement (Scalability). It is expected that LN should work for at least millions of users, and routing capabilities should scale accordingly.

Requirement (Automation). The routing algorithm should be automated in a sense that a user should only need to set some high level options (e.g., maximum transaction fees he is ready to pay)

in order to keep a LN node operational. A user should not need to tune technical parameters of the algorithm or manually select payment routes (unless the user wants to).

According to the requirements above, the routing problem on LN could be formulated as follows.

Problem 1. Find a payment route from the given sender S to the given recipient R subject to the following constraints:

- The route must be found based on knowledge of LN accessible by the sender S. The decision on the route should be made by S (i.e., the decision should not be outsourced to trusted third parties, as it would defeat the trustless nature of LN)
- The algorithm must allow for situations when the sender's view of the network is incomplete (e.g., because of memory constraints) and/or outdated
- The route finding algorithm must execute in reasonable time and consume a reasonable amount of memory
- The discovered route must be optimal according to a certain cost function and the sender's view of the network. (In the simplest case, the routing algorithm could minimize the cumulative fees on the route.)

2 Overview of routing algorithms

In this section, we provide an overview of routing algorithms that were proposed for payment routing in Lightning Network and in similar networks.

2.1 Previous ideas

2.1.1 Hub-and-spoke Network

The Bitcoin community proposed several possible routing algorithms for LN. One of the earlier ideas [17] proposed splitting nodes into two classes: wallet nodes and routing nodes (hubs), where the former can only make payments and the latter can only route payments (and earn transaction fees in the process). This scheme could help make routing efficient from the start, since knowledge of all hubs (there are not too many of them and full list may be published) would guarantee a high probability of finding a path to any node. At the same time, this scheme may be prone to centralization. Furthermore, there are no incentives for wallet nodes to stay online; when a payment would be sent to a wallet, it would likely stick to the hub closest to the recipient and the hub would need to wait for the node to go online to finalize the payment. Another downside of pure hubs is the lack of fault-tolerance. If there are only a few hubs on the network, and one of them experiences a major global outage, the network may be crippled as a result.

2.1.2 Global Beacons

Another idea [18] was to select several nodes randomly as global beacons, rotating the beacon set on a certain schedule (e.g., every day). All nodes would try to find paths to these beacons and use the recovered paths to send payments. This scheme would be more decentralized than the previous one in the sense that the network (or a significant portion thereof) would not be controlled by a small group of entities. However, the scheme makes being a global beacon much more profitable than other nodes, which may lead to Sybil attacks (i.e., creating fake nodes to increase the probability of becoming a beacon). Additionally, because the information on current beacons would be public, beacons could be a target of DoS attacks that attempt to paralyze the network. Global beacons would need to have appropriate processing capabilities in terms of network bandwidth, computational power, and value stored in payment channels. Therefore, the pool of feasible beacon candidates would probably be quite limited. Lastly, beacons would need to create many channels during the time they are selected, most of which could become obsolete after other beacons are selected.

2.1.3 Local Beacons

The evolution of the beacon concept [19] resulted in a proposal where each node would have a list of personal beacons. When a payment is sent, the sender and the receiver would need to compare their beacon lists and see if there are intersections through which payment can be routed. Local beacons would enable a more uniform traffic distribution over the network. Additionally, in this scheme, every node has a chance to be selected as a beacon for some other node; thus, transaction fees could be distributed more evenly than in the scheme with global beacons. This could incentivize nodes to stay online, therefore improving the routing and value transfer capabilities of the network.

2.2 Routing Algorithms Solving Similar Problems

It is difficult to estimate the structure and behavior of the Lightning Network while it's still under development. However, in certain aspects, LN could resemble a mobile ad hoc network (MANET) [20]. This assumption is supported by the following arguments:

- The availability of channels between nodes can appear and disappear frequently, like connections between nodes in MANETs. This is true since all LN users are free to open and close channels whenever they want; available capacity in a channel may change frequently, making it impossible to forward payments through certain channels.
- New nodes sometimes join the network, while existing ones can go offline for a while. Since we are assuming that LN would be completely decentralized, we cannot expect that all nodes would be constantly available, similar to nodes in MANETs.
- The network needs to be self-configuring, as MANETs are. In other words, the functionality of the network must be preserved despite changes in topology (due to nodes going offline or channels

closing) or hostile behavior of a portion of surrounding nodes (thus, we must not assume that a significant portion of nodes will act cooperatively).

However, the analogy between LN and MANETs is not perfect; the source routing requirement and the fact that each additional hop on the route increases the sender's expenses are not common in MANETs. Nevertheless, the payment routing algorithm for Lightning Network could benefit from some ideas from data routing algorithms successfully deployed in MANETs.

Routing algorithms for MANETs could be classified into 3 categories: table-driven (proactive), on-demand (reactive), and hybrid [21].

Table-driven (proactive) protocols. These protocols maintain up-to-date routing information about all the nodes in the network in the form of a routing table. In general, proactive routing protocols have a large control overhead and are not scalable for large networks.

OLSR [22] is an example of a table-driven routing algorithm.

On-demand (reactive) protocols. The second type of routing strategy employed in MANETs is initiated on-demand (or reactive) source routing. Unlike table-driven routing protocols, an on-demand routing protocol executes the path-finding process and exchange routing information only when a path is required. Correspondingly, reactive protocols do not require a large passive overhead. Total information of the network graph is not maintained in on-demand routing protocols, thus making them less demanding in terms of memory usage.

Two examples of on-demand routing protocols are Ad hoc On-Demand Distance Vector routing (AODV) [23] and Dynamic Source Routing (DSR) [24].

Hybrid protocols. There is a fundamental trade-off between proactive and reactive routing approaches. The former provides good latency and reliability, but is not scalable due to large overhead; the latter sacrifices reliability for scalability. Hybrid routing protocols are a relatively new development trying to find a balance between reliability and scalability.

Examples of hybrid routing protocols are Safari [25] and cjdns [26]. According to the routing algorithm used in cjdns, each node keeps the routing table of the local part of the network, as well as routes to a few distant nodes that are close to original node in terms of the address distance (calculated based on node addresses). On the routing stage, nodes request missing parts of the path being discovered from intermediate nodes, which are generally chosen by minimization of the address distance between the packet receiver and the current intermediate node.

3 Proposed Hybrid Routing Algorithm

As a way to satisfy the requirements listed in Section 1, we define a hybrid routing algorithm, which we call Flare, consisting of proactive and reactive stages. The idea behind this approach is that the state of LN can be split into two distinct parts:

- slowly changing, or static, information (payment channels between nodes)
- quickly changing, or dynamic, information (status of nodes, distribution of funds within payment channels, fees for using a channel, etc.)

Thus, it could make sense to collect information about newly opened or closed payment channels, which could be of interest to an LN node, proactively (on a certain schedule and/or using restricted flooding). On the other hand, it makes little sense to gather quickly changing information proactively, as it could change unpredictably at any time.

A rough high-level description of the routing algorithm is as follows (Fig. 2).

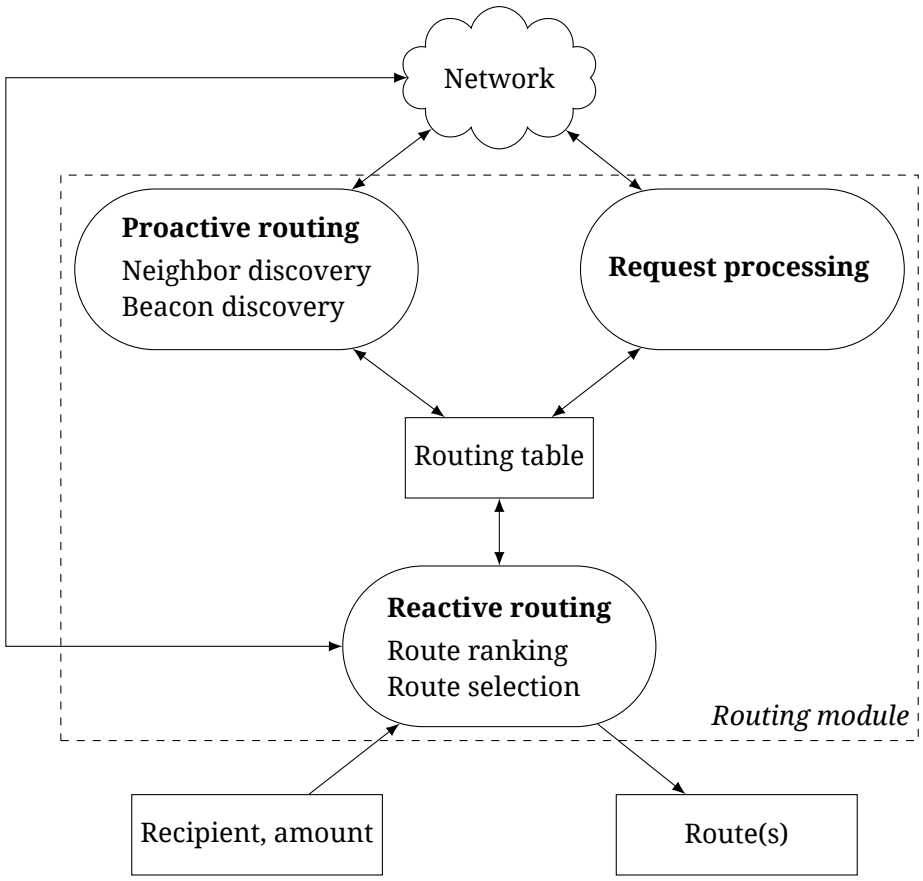


Figure 2: The place of the routing module in the LN node's business logic. The goal of the module is to provide one or more payment routes (i.e., lists of LN nodes) to the intended recipient for the specified payment amount. In order to accomplish this, a node utilizes information about the network by interacting with other nodes, which is collected both proactively (by discovering the node's neighborhood and beacons) and reactively (by selecting and ranking candidate routes). The collected information is placed into the node's routing table, which is used to process requests both from the node operator and from other nodes.

Route discovery (proactive part). Each node updates its knowledge of the network in the form of a routing table on a schedule. The routing table is a certain subset of channels that are known to exist in the network. Routing tables are constructed with an aim to find paths to the recipient or, if that is impossible, determine beacon nodes (or simply beacons) that are likely to help in finding paths.

After route initial route discovery, each node has a very clear view of the nodes within its neighborhood. The usage of beacons augments a node's visibility of the network at large by incorporating random nodes beyond the source node's local neighborhood.

As a result, a node's network view is fog of war like, with strong visibility close in a local radius, boosted by random feeler connections farther away. This allows a node to maintain routing state with space complexity logarithmic with respect to total network size, while ensuring reachability to all other nodes with high probability.

Route selection (reactive part). When a node decides to send a payment over LN, it uses the combination of its and the recipient's routing tables to find possible routes in the network from the sender to the recipient. If it is impossible to find appropriate paths using these two routing tables, the routing tables of the recipient's beacons and other carefully chosen nodes may be used as well. Due to the construction of route discovery, these nodes are more likely to know one or more routes to the recipient than random nodes in the network. The routes found are ranked with respect to a predefined cost function, which partially depends on static information, such as route length, and on dynamic information, such as fees to use channels. We assume ranking is performed in two independent and subsequent steps:

- **Static ranking** based on static information (i.e., on the information from routing tables only). Static ranking yields a relatively small sample of candidate routes.
- **Dynamic ranking** based on dynamic (and, possibly, static) information. In order to obtain information for dynamic ranking, an onion wrapped polling message is sent through every route among candidate routes collecting up-to-date information on channels and nodes in the route.

This heuristic would make the reactive step of the routing algorithm reasonably short while preserving its accuracy.

Finally, after all information is collected and channels are dynamically ranked, the sender selects the best route that will be used to forward the payment. To shorten the process in practice, we would like to sometimes stop dynamic ranking and send a payment immediately if we have received a "good enough" route ranking in the process of collecting dynamic data.

We require several paths on the reactive step for fault tolerance; if a single node in a path is non-responsive, we can fall back to the next path found and so on. If no viable routes are found, we sequentially request routing tables from beacon nodes and repeat the procedure. A high probability of finding a payment route stems from the beacon selection method based on address distance, similar to cjdns.

We describe proactive steps of the algorithm in Sections 3.2, 3.3, and 3.4. Sections 3.5 and 3.6 are dedicated to the aspects of reactive routing.

Network assumptions. We mostly abstract away from the network communication in LN, as it is a subject for another discussion. We assume there exists a way for any two nodes in LN to communicate

with each other, perhaps with some prior setup (such as a distributed hashtable overlay maintained among LN nodes that would store mapping of node identifiers to their current IP addresses). Similarly, an LN node can determine whether another LN node is in the network at any point in time.

We further assume that communication among nodes is confidential, ensures integrity, is authenticated (e.g., with the help of digital signatures or HMACs with a pre-established shared secret) and is not vulnerable to replay attacks (e.g., by including a monotonically increasing sequence number or a timestamp into each message). An onion routing scheme [27] together with the corresponding setup (e.g., setting and rotating onion routing keys) is assumed to be readily available in order to preserve payment privacy when necessary.[1]

We assume nodes' clocks are loosely synchronized, at least in the sense that a (reasonably large) time interval measured on one node and the same interval measured on another node may not differ significantly. Synchronized clocks are beneficial when defining anti-DoS timeouts for messages in the following algorithm description; we want all nodes to perceive timeouts in the same manner. Heartbeat messages could be utilized in order to achieve clock synchronization.

We do not define punitive measures for nodes that are caught in hostile behavior (e.g., providing misleading information); this is a subject for further research.

3.1 Mathematical Model

According to the description in Section 1, LN could be represented as an undirected graph $G = (V, E)$, where V is the set of nodes and $E \subseteq V^2$ is the set of opened channels. For every channel $e \in E$, define $\mathrm{Cap}(e)$ the total capacity of the channel; equivalently, we define binary symmetric function $\mathrm{Cap} : V^2 \to [0, +\infty)$, where for non-existent channels, capacity is equal to 0:

$$\forall v_1, v_2 \in V \quad (v_1, v_2) \notin E \Leftrightarrow \mathrm{Cap}(v_1, v_2) = 0.$$

Definition 1. *(Hop) distance* between two nodes is equal to the minimum number of channels in LN connecting the nodes:

$$d_{node}(x, y) \stackrel{\text{def}}{=} \min\{n \in \mathbb{N} : \exists v_1, \ldots, v_{n+1} \in V \quad x = v_1; y = v_{n+1}; \forall i \in 1, \ldots, n \quad (v_i, v_{i+1}) \in E\}.$$

Definition 2. *(Hop) distance* between a node $x \in V$ and a channel $e = (y, z) \in E$ is equal to the minimum of the distances between the node A and the channel's ends:

$$d_{chan}(x, (y, z)) \equiv d_{chan}(x, e) \stackrel{\text{def}}{=} \min\{d_{node}(x, y), d_{node}(x, z)\}.$$

In the following algorithm description, we will also use the following definitions.

Definition 3. *Adjacent nodes* $\mathrm{Adj}(v)$ for node $v \in V$ are nodes at distance 1:

$$\mathrm{Adj}(v) = \{z \in V : d_{node}(v, z) = 1\},$$

i.e., nodes, with which v has an opened payment channel.

[1] We plan to use Sphinx [28] in order to compactly encode payment paths and HORNET [29] as a general-purpose data transport. Furthermore, HORNET could be used in order to utilize rendezvous routing akin to Tor's hidden services [30].

Note that each node always knows a precise set of its adjacent nodes; it does not have to communicate with other nodes to retrieve it.

Definition 4. Nodes(T) is a set of nodes connected by edges from the set of edges $T \subseteq E$:

$$\text{Nodes}(T) \stackrel{\text{def}}{=} \{A : (A, \cdot) \in T \vee (\cdot, A) \in T\}$$

In order to ensure convergence of the routing algorithm, we introduce the concept of *node addresses*. A node address is a globally unique node identifier, with a binary distance function (conforming to standard properties of distance) defined between all pairs of addresses. (Note that the proximity in terms of the address distance does not generally correlate to the proximity of nodes in terms of hop distance.) Node addresses serve several purposes:

- as node identifiers in the routing table
- in implementing the beacons mechanism to increase the probability of route discovery in case a sender and a recipient are not close in the network.

We will denote the address distance between nodes $x, y \in V$ as $\rho(x, y)$.

We propose the following concrete instantiation of addresses and distances:

- **Address** is a 256-bit integer computed as SHA-256 [31] of the identity key of the node
- **Address distance** between the nodes is calculated as bitwise XOR of node addresses represented as unsigned integers (little-endian for explicit definition), i.e., similarly to Kademlia [32] and cjdns. Note that this function preserves all distance properties: non-negativity, identity of indiscernibles, symmetry, and subadditivity.

3.2 Routing Table

Route discovery maintains the node's routing table. A routing table for a node $v \in V$ is a set of long term (static) information about the LN topology, i.e., nodes participating in certain channels:

$$v.RT \stackrel{\text{def}}{=} \{(a_i, b_i) : (a_i, b_i) \in E\},$$

with the set of payment channels E defined as per Section 3.1. The routing table also stores additional static information about channels (Table 1), in particular, their total capacity, which could be retrieved in the process of channel verification (Appendix A). A routing table may also proactively maintain certain information for performance reasons, such as a route or a list of routes to each node from the table (which is useful in beacon discovery and route selection). We assume that the size of the routing table $|v.RT|$ is rather small (we give a rough estimate of the table size later).

The design goal of routing tables is to maintain enough up to date information about network topology in order to perform efficient route selection (Section 3.5) in terms of time, computational resources, and network bandwidth.

We propose storing the following channels in the routing table:

Table 1: Fragment of a routing table for node B from the graph depicted in Fig. 1. Addresses of nodes are denoted as letters

Node 1	Node 2	Total channel capacity
u	v	0.5 bitcoins
v	y	0.3 bitcoins
v	w	0.1 bitcoins
u	y	0.001 bitcoins
y	z	4 bitcoins
w	z	1 bitcoin

- **Neighborhood map:** Channels between all node neighbors (in terms of hop distance) within neighbor radius $r_{nb} \in \mathbb{N}$. This should guarantee that nodes removed from each other by no more than $2r_{nb}$ hops will find at least one route to each other (Fig. 3).
- **Beacon paths:** Channels that form paths to N_{bc} beacon nodes that are close to the node in the address space. As node addresses are assigned randomly, this would lead to each node having routes to nodes randomly scattered around the network.
- **Cached payment routes:** N_{cache} channels that form paths to nodes that the node has already successfully used before to transfer payments. This helps the node remember working paths once they are discovered during the reactive stage of the routing algorithm.

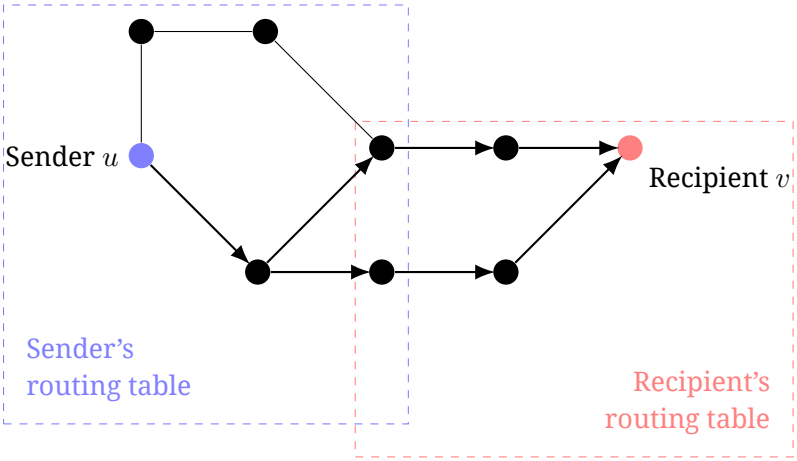


Figure 3: Route selection based on neighbor parts of routing tables only. We assume neighbor radius $r_{nb} = 2$. Sender u and recipient v are at distance 4 one from another; by requesting v's routing table and uniting it with its own, the sender is able to discover two routes of length 4 to the recipient.

The typical size of a routing table depends on the network topology. If each node has opened channels with n nodes on average, and the average length of path to a beacon is r_{bc}, the table size could be estimated as $n^{r_{nb}} + N_{bc}r_{bc} + N_{cache}$. If each node has $n = 5$ neighbors on average, $r_{bc} = 10$, $r_{nb} = 4$, $N_{bc} = N_{cache} = 100$, then the routing table would typically store data on $5^4 + 100 \cdot 10 + 100 = 1725$ channels. If a channel takes about 150 bytes to store, the typical size of a routing table would be about

250,000 bytes.

In some cases, it may be beneficial to store a larger routing table. There is a trade-off between the quality of the routing algorithm (i.e., how well a discovered route approximates a global minimum in the route space with respect to the chosen cost function) and the size of a routing table. The larger the routing table, the greater the number of nodes to which the node could successfully find a path (and shorter the length of paths found); however, network bandwidth and computational power required to maintain the table are increased too.

Adjusting parameters r_{nb}, N_{bc} and other parameters introduced later may require some coordination among nodes in LN; if the values of these parameters differ, the performance of the routing algorithm may be suboptimal. The mechanism of adjustment is beyond the scope of this paper.

3.3 Neighborhood Discovery

The mechanism of updating neighborhood information is based on messages **NEIGHBOR_HELLO**, **NEIGHBOR_UPD**, **NEIGHBOR_RST** and **NEIGHBOR_ACK**:

- **NEIGHBOR_HELLO** transmits the full routing table of an LN node.
- **NEIGHBOR_UPD** carries incremental changes to the node's routing table since the last sent update message. The proposed message format is shown in Table 2.
- **NEIGHBOR_RST** message is sent after update information was supposedly lost. It signals that the message sender wishes to receive a complete version of message recipient's routing table (i.e., repeat a **NEIGHBOR_HELLO** greeting). A node may send **NEIGHBOR_RST** if its routing table has become corrupted.
- **NEIGHBOR_ACK** is sent in response to **NEIGHBOR_UPD** or **NEIGHBOR_HELLO** in order to signal that the node has properly processed the message. A **NEIGHBOR_ACK** message carries a sequence number or another unique identifier (which we discussed earlier when laying out basic network assumptions) for the corresponding **UPD** or **HELLO** message. This is done in order to reliably identify on the recipient node which message **NEIGHBOR_ACK** responds to.

Table 2: Proposed format of a **NEIGHBOR_UPD** message. The message contains an incremental update to the sender's routing table, which is encoded as a list of opened and closed channels since the last update

Sender: u	**Recipient:** v	
Node 1	Node 2	Open/close bit
u	w	opened
w	x	closed

A node accepts **NEIGHBOR_*** messages only from adjacent nodes (i.e., nodes with which it has an open payment channel). Updates sent to adjacent nodes may be aggregated (more on that later). In

order to mitigate DoS attacks, a node may introduce a minimum interval between successive **NEIGHBOR_HELLO** and/or **NEIGHBOR_UPD** messages and inform adjacent nodes about it, e.g., when establishing network connections with these nodes.

After node u receives a **NEIGHBOR_HELLO** or a **NEIGHBOR_UPD** message, it modifies its routing table based on the channels received in the message (Algorithm 1, Fig. 4). The node only adds to its routing tables channels in its neighborhood; all other channels are ignored. For each such channel, the node performs the verification procedure described in Appendix A in order to ensure that the funding transaction corresponding to the channel is indeed present on the blockchain and that the channel has not been closed. In this and the following algorithms, we assume that channel verification could be cached.

Algorithm 1: Processing received **NEIGHBOR_HELLO** or **NEIGHBOR_UPD**

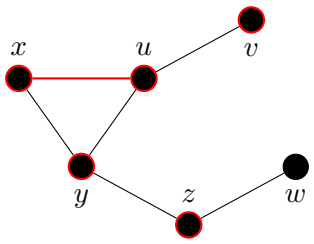
Input:

- Current node (message recipient) u
- Message sender v
- Set of new channels in the message $M \subset E$
- Set of closed channels in the message $M_r \subset E$ (for **NEIGHBOR_HELLO**, $M_r = \emptyset$)
- Neighborhood radius $r_{nb} \geq 1$

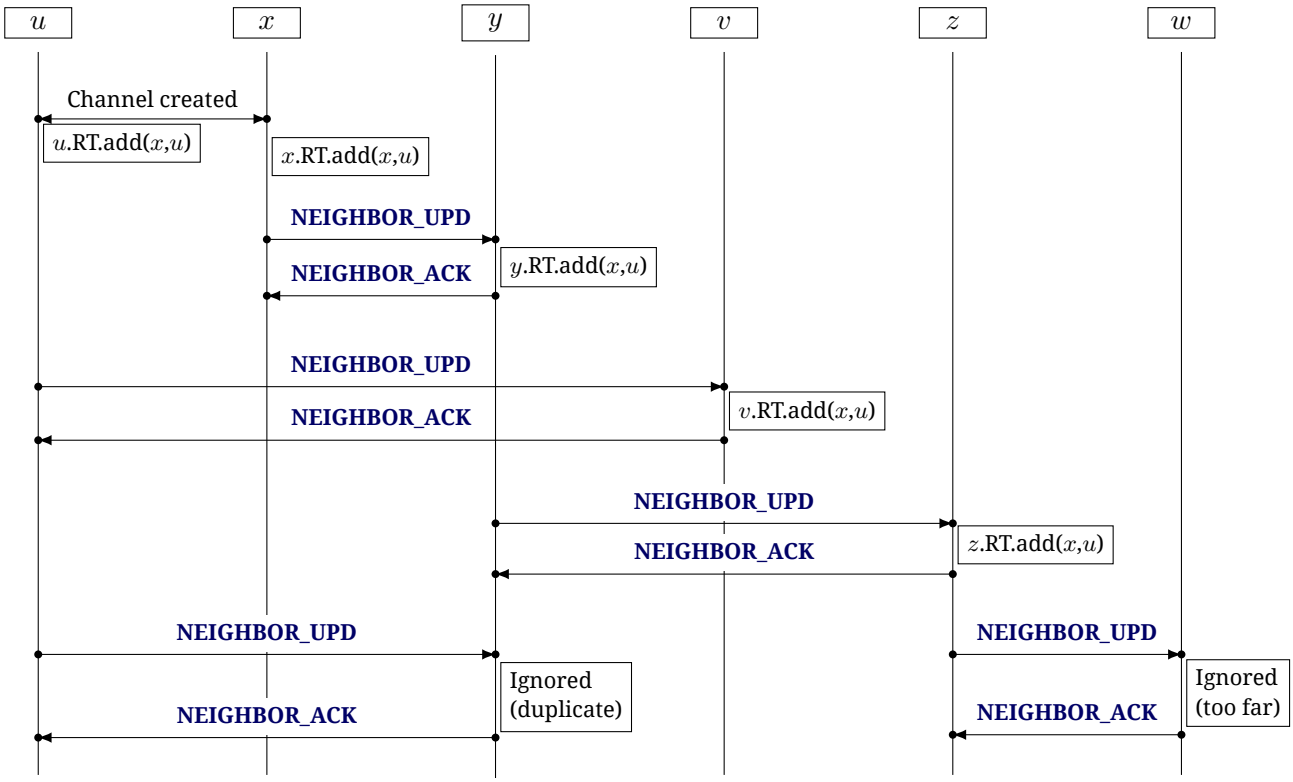
Ensure: Updated routing table $u.RT$

1: Preliminary merge new channels with u's routing table: $RT_{pre} := u.RT \cup M$
2: Assume $G_{pre} := (\text{Nodes}(RT_{pre}), RT_{pre})$ the view of the network built based on the information available to u
3: **for all** newly seen channels $e \in M \setminus u.RT$ **do**
4: Compute the minimum hop distance from u to e in the graph G_{pre}: $d := d_{chan}(v, e)$. If the distance cannot be be computed (e.g., because G_{pre} is disjoint), assume $d := +\infty$
5: **if** $d \leq r_{nb}$ **then**
6: Verify e on the blockchain; get total channel capacity $\text{Cap}(e)$
7: **if** e is verified as open **then**
8: $u.RT := u.RT \cup \{e\}$; save $\text{Cap}(e)$ in the table
9: **for all** closed channels present in the routing table $e \in M_r \cap u.RT$ **do**
10: Verify on the blockchain that is closed
11: **if** e is verified as closed **then**
12: $u.RT := u.RT \setminus \{e\}$

A node sends **NEIGHBOR_HELLO** and **NEIGHBOR_UPD** messages to the adjacent nodes based on how its routing table is updated, and on received **NEIGHBOR_RST** requests. Communication with each adjacent node is processed independently using an event-driven approach, with single event processing defined in Algorithm 2. The main event processing rules are:



(a) Sample network topology



(b) Message flow

Figure 4: Payment channel added between nodes x and u in a sample network topology and the corresponding message flow. In the sample network, neighborhood radius $r_{nb} = 2$; thus, all nodes except for w (which is too far from the created channel) eventually update their routing table to include a new channel. Some nodes receive duplicate notifications, with the later update ignored.

- Adjacent nodes may be active or inactive, depending on whether they are in the network at the moment. Messages are not sent to inactive nodes in order to save bandwidth and computational resources.
- All **NEIGHBOR_UPD** messages send an incremental update relative to the synchronized view of the routing table, defined the last **NEIGHBOR_HELLO** or **NEIGHBOR_UPD** request **ACK**-ed by the adjacent node.
- There is never more than one pending **NEIGHBOR_HELLO** or **NEIGHBOR_UPD** message (i.e., a message that has not timed out and for which the node has not received a **NEIGHBOR_ACK** or **NEIGHBOR_RST** response). If an update to the node's routing table occurs while there is a

pending message, the update waits until the message is finalized (either by a timeout or by an **ACK** response). If the pending message times out, the adjacent node is considered inactive.

- **NEIGHBOR_HELLO** is sent on channel creation or in a response to a **NEIGHBOR_RST** message. In all other cases, the adjacent node is informed about updates with a **NEIGHBOR_UPD** message.

For didactic reasons, the message dispatch described in Algorithm 2 is substantially simplified:

- We do not detail the mechanism used to determine whether an adjacent node is present in the network. This is related to LN networking and is outside the scope of this paper.
- To mitigate **NEIGHBOR_RST** spam, we can introduce a timeout t_{rst} for this type of message. A node should ignore a **NEIGHBOR_RST** if it is received more frequently.
- Incremental updates sent by **NEIGHBOR_UPD** may be grouped. A node may implement a minimum interval between outbound updates (e.g., according to the corresponding anti-DoS setting advertised by the adjacent node).
- In order to reduce consumed bandwidth, a node tracks the origin of routing table updates and does not relay the update to the nodes that are already guaranteed to know it (i.e. to the sender of messages and nodes adjacent to the sender).
- In certain cases (e.g., when the synchronized view is too far in the past), sending an incremental update may be less efficient than sending a complete table with **NEIGHBOR_HELLO**. We assume the node can pick the most efficient way to transmit updates.
- It may be decided by the owner of each node to provide filtered updates to the node's routing table (e.g., for privacy reasons). The incentive for a node owner to share routing information would be that giving it to other nodes increases chances to discover routes passing through the node and thus to earn transaction fees.

Resource consumption. Traffic generated by a **NEIGHBOR_HELLO** message is approximately equal to the size of a routing table, or around 250 Kb per message according to calculations in Section 3.2. A node needs to receive a **NEIGHBOR_HELLO** message once it opens a new payment channel, and around one several per day if some update package was lost or node went offline.

Network bandwidth consumed by **NEIGHBOR_UPD** messages is more difficult to estimate. Assume that

- A regular node has some 10 opened channels per each update message (which is probably an overestimation)
- **NEIGHBOR_UPD** message is received once in 5 minutes from each adjacent node
- Each such message on average carries information about 10 channels, up to 200 bytes for each channel

Under these assumptions, **NEIGHBOR_UPD** messages produce 20 kilobytes of traffic per node per 5 minutes, or 5.76 megabytes per node per day.

Algorithm 2: Event-based updates between adjacent nodes (simplified)

Input:

- Current node u
- Adjacent node $v \in \text{Adj}(u)$
- $u.pending(v)$ – is there a pending **NEIGHBOR_UPD** or **NEIGHBOR_HELLO** request to v (i.e., a request that did not receive an appropriate response and is not timed out)?
- $event$: exactly one of
  - INIT: Node v has opened a channel with u
  - ACT: Node v has changed its network status from inactive to active
  - RT: u's routing table has changed
  - ACK: u has received **NEIGHBOR_ACK** from v in a reply to the pending message
  - RST: u has received **NEIGHBOR_RST** from v
  - TO: u's pending **NEIGHBOR_UPD** or **NEIGHBOR_HELLO** message to v has timed out

Ensure: Event processing by node u

if $event =$ TO **then** ▷ the pending message has timed out
    Declare v as inactive; $u.pending(v) :=$ false
if v is inactive **then**
    exit
if $event =$ ACK **then**
    Remember the synchronized state of u's routing table based on the pending request. Future **NEIGHBOR_UPD** messages will build upon this state
    $u.pending(v) :=$ false
else if $event =$ RST or $event =$ INIT **then**
    Send **NEIGHBOR_HELLO** to v
    $u.pending(v) :=$ true
    exit
if $u.pending(v)$ **then**
    exit ▷ Do not send v more than one message at a time
if $u.RT$ changed since the last synchronized state **then** ▷ here, $event \in \{\text{ACT}, \text{RT}, \text{ACK}\}$
    Send **NEIGHBOR_UPD** to v with the increment of $u.RT$
    $u.pending(v) :=$ true

Complexity. Assuming that verification of a single channel on blockchain takes O_{check} operations, the complexity of Algorithm 1 is $\mathcal{O}((|M| + |M_r|)(O_{check} + \log|R|))$, where

- $|M|$ is the number of added channels,
- $|M_r|$ is the number of channels that need to be removed,
- N_{rt} is the number of records in node's routing table.

The complexity comes from the fact that each channel should be located in the node's routing table (assuming that records in the node's routing table are indexed by node addresses) and verified on the blockchain. The complexity of Algorithm 2 is determined by the cost for a node to form **NEIGHBOR_UPD** or **NEIGHBOR_HELLO** messages, which is $\mathcal{O}(|M| + |M_r|)$ and $\mathcal{O}(N_{rt})$ correspondingly.

Parameter choices. An interesting edge case is the neighborhood radius $r_{nb} = +\infty$ for all nodes in the network. In this case, neighborhood discovery would allow each node to collect an almost complete LN map (lacking changes in topology that happened recently, as their propagation over the network would take time). However, that would require substantial resources from nodes. If we assume a network with 1 million nodes and 10 payment channels per node on average, each node would have to store about 10 million channels, i.e., hundreds of megabytes of space. If 2% of all channels (which is 200,000 channels in the example above) are opened or closed in the network every day, each node needs to perform 200,000 channel validations daily. This may be acceptable for lightning nodes that live on top of full Bitcoin nodes, but appears infeasible for the ones that live on top of SPV wallets (as they would need to send request to confirm each channel to the full Bitcoin node on the network).

Based on the routing table estimates in Section 3.2, $r_{nb} \in \{3, 4, 5\}$ seems to be a reasonable default for the neighborhood radius. As the network size grows, neighbor radius could be adjusted over time accordingly to balance size of routing table and chance of route discovery. This would seamlessly lead to each node increasing its network knowledge and keeping the chance to discover route to recipients at an acceptable level.

3.4 Beacon Discovery

Beacon discovery allows each node to find its neighborhood to expand its awareness of the network. Beacons for node u, denoted as $u.bc \in V^{N_{bc}}$, are selected as the closest nodes in the address space; i.e., ideally we would want

$$\forall v \in u.bc, \forall z \in V \setminus (\{u\} \cup u.bc) \quad \rho(v, u) < \rho(z, u).$$

In practice, some closest nodes in the address spaces may not be found during beacon discovery. However, this would not substantially impact the role of beacons, which is the node knowing routes to far away nodes randomly distributed across the network. The address distance is used in defining beacons to ensure the convergence property of the beacon discovery procedure; we briefly explore manually defined beacons in Section 5.

Besides a list of beacons, each node u maintains $u.rb \in V^*$ a list of of nodes that have chosen u as a beacon (i.e., $v \in u.rb \Leftrightarrow u \in v.bc$). $u.bc$ and $u.rb$ sets could be used in anti-DoS protection during reactive routing (Section 3.5).

Beacon discovery is based on **BEACON_REQ**, **BEACON_ACK** and **BEACON_SET** messages.

- **BEACON_REQ** message is sent to tell the node that it is considered as a beacon candidate. It does not carry any additional data.
- **BEACON_ACK** message is sent as a reply to a **BEACON_REQ** message and confirms that node agrees to be a candidate. It may also contain paths to the nodes in the routing table that are better beacon candidates (i.e., nodes with an address closer in the address space to the sender of **BEACON_REQ** than the current node). More concretely, if **BEACON_ACK** is sent from node v to u, then it contains a set of nodes $C_v = \{z \in \text{Nodes}(v.RT) : \rho(z,u) < \rho(v,u)\}$ and a mapping $M_v : C_v \to v.RT^+$, where $M_v(z)$ are known routes from v to z. The structured nature of M_v allows for its easy verification by the recipient. C_v (and therefore M_v) may be empty.
- **BEACON_SET** message is sent to tell a node that it was selected as a beacon.

To find beacons, a node looks at its routing table for the nodes that are closest to it in the address space and informs them with a **BEACON_REQ** message that they are chosen to be beacon candidates. When a node receives such a message, it searches its routing table for nodes that are even closer in the address space to the sender, informing the sender of the findings with the help of a **BEACON_ACK** message. After that, the sender can consider newly found nodes as beacon candidates and repeat the process, until enough responsive beacon candidates have been considered (Fig. 5). Finally, a node selects responsive candidates as beacons, starting from those closest in the address space, and informs them about its decision (Algorithm 3). After selecting a beacon, a node updates its routing table accordingly.

When node v receives a **BEACON_REQ** from node u and decides to accept its role as a beacon:

1. Node checks routing table and finds $C_v := \{z \in v.RT : \rho(u,z) < \rho(u,v)\}$.
2. Node sends **BEACON_ACK** message to u that contains paths to nodes in C_v.

When node v receives a **BEACON_SET** message from node u, to which it previously sent **BEACON_ACK**, the node remembers for some time that it was selected as a beacon of u: $v.rb := v.rb \cup \{u\}$.

Beacon discovery starts after the node has discovered its neighborhood as described in Section 3.3 during the initial node setup. The algorithm always knows a path to each responsive node required on step 21; this could be proven by induction. Indeed, each responsive beacon candidate z is either a neighbor added to B on step 1 (in which case the route to the node was discovered before the start of Algorithm 3), or it was added to B on step 16 by querying some other node v. In the latter case, the route to z could be defined as a union of the route from the current node u to v (which is known by the induction hypothesis) and the route from v to z (which is reported by v in the **BEACON_ACK** message).

Beacon discovery may be initiated at any moment in order to update the beacon set. In this case, the set of beacon candidates formed on step 1 of Algorithm 3 would include most of current node

Algorithm 3: Beacon discovery

Input:

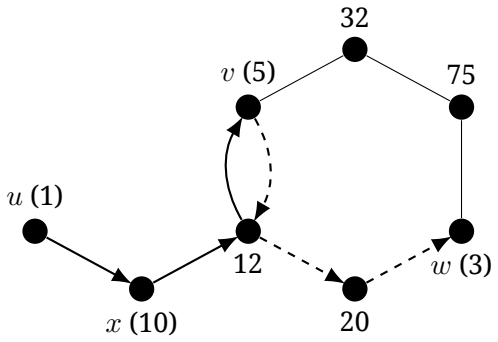
- Current node u
- Number of beacons to find $N_{bc} > 0$

Ensure: Updated set of u's beacons $u.bc$

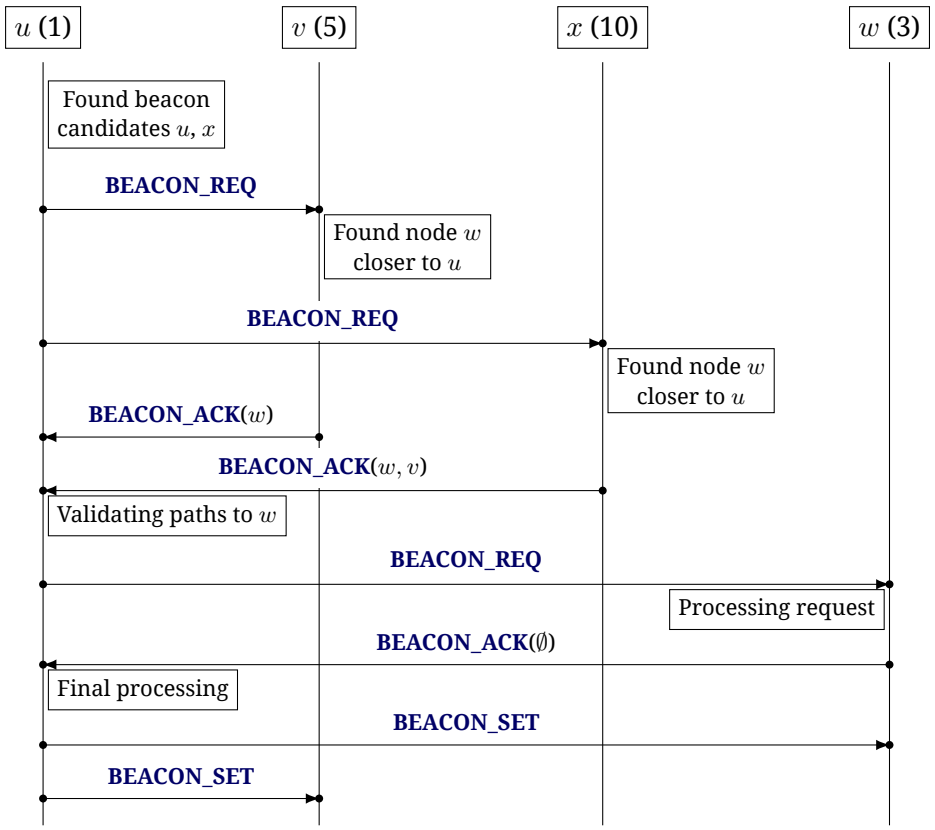
Initialize the set of unprocessed beacon candidates B as N_{bc} nodes from the u's routing table closest to u in the address space
Initialize processed nodes $U := \emptyset$
Initialize responsive nodes $R := \emptyset$
while $|B| > 0$ **do**
    Select an unprocessed beacon candidate v closest to u in the address space: $v := \arg\min_{z \in B} \rho(z, u)$
    Mark v as processed: $U := U \cup \{v\}; B := B \setminus \{v\}$.
    Send **BEACON_REQ** to v. Wait for **BEACON_ACK** response
    if **BEACON_REQ** times out **then**
        continue to the next node
    Assume the **BEACON_ACK** response is (C_v, M_v) as described above
    Mark v as responsive: $R := R \cup \{v\}$
    for all $z \in C_v \setminus (B \cup U)$ **do**
        Verify $\rho(z, u) < \rho(v, u)$
        Verify that $M_v(z)$ is a path from v to z
        Verify channels in $M_v(z)$ on the blockchain (Appendix A)
        Add z to the beacon candidates: $B := B \cup \{z\}$
    while $|B| > N_{bc}$ **do**
        Remove the unprocessed beacon candidate closest to u in terms of the hop distance:

$$z^* := \arg\min_{z \in B} d_{node}(u, z); B := B \setminus \{z^*\}.$$

for all $v \in R$, sorted by increasing $\rho(u, v)$ **do**
    Send **BEACON_SET** to v and mark v as a beacon: $u.bc := u.bc \cup \{v\}$
    Add the discovered path(s) from u to v to $u.RT$
    if $|u.bc| = N_{bc}$ **then**
        exit



(a) Sample network topology



(b) Message flow

Figure 5: Beacon discovery for node u in a sample network topology and the corresponding message flow. Each node is denoted by its address and an optional letter identifier. We assume neighborhood radius $r_{nb} = 3$ and $N_{bc} = 2$ beacons per node. For simplicity, we consider address distance defined as the absolute difference between addresses (e.g., $\rho(u, v) = |1 - 5| = 4$). Initially, u finds v and x as the closest nodes in the address space among nodes in its routing table, and marks them as a beacon candidates. v is aware of node w, which is closer to u in the address space and x is aware of both v and w. Thus, v sends u a **BEACON_ACK** message with w as a new beacon candidate and a route from v to w, while x sends a similar message with routes from x to v and w. The process finishes with u sending **BEACON_SET** messages to w and v.

beacons (which should also be nodes closest in address space). Note that if all beacon candidates become unresponsive, the node should exclude them from consideration and form a new candidates list from the remaining nodes that are closest in the address space to the node.

In order to protect against DoS attacks, a node should not accept **BEACON_*** messages from one node more often than once in certain specified time (e.g., 5 minutes). Messages with extraordinarily long routes (e.g., longer than 100 channels) should be disregarded.

As channels that form paths to and from beacons may be far away from the node, **NEIGHBOR_UPD** messages may fail to deliver information on channel closing. For channels that are farther then r_{nb}, the node should check whether a channel has been closed on the blockchain proactively on a certain schedule. Discovered closed channels should be removed from the routing table; graph components disjoint with the current node, which could possibly arise as a result of channel removal, may be removed too.

Resource consumption. **BEACON_SET** messages can be very small. **BEACON_ACK** messages could be significantly larger, due to multiple paths being sent. **BEACON_ACK** message size would be about 32 bytes per each node in the path.

The amount of traffic caused by **BEACON_ACK** messages heavily depends on the network topology and is intractable in the general case. See Section 4 for results obtained on simulated networks.

3.5 Route Selection

As we mentioned previously in the high-level overview of the routing algorithm, route selection consists of two parts:

- **Static ranking,** during which the sender uses information from routing tables. In order to accomplish this, the sender node requests routing tables from the recipient and, possibly, other nodes closest to the recipient in the address space. Because of the node selection, requested routing tables are likely to contain routes to the recipient. Based on the routing tables, the sender extracts a subset of shortest (in terms of hop distance) routes to the recipient (candidate routes), which then serve as inputs for the next stage of route selection.
- **Dynamic ranking,** during which the sender checks validity of the found routes using validity proofs (Appendix A), retrieves dynamic characteristics of the channels comprising the routes with probe messages, thus forming the final route ranking.

We use two types of messages on the route selection phase:

- **TABLE_REQ** requests a routing table from the recipient. This message has no additional workload
- **TABLE_RESP** is sent in response to **TABLE_REQ** and contains a full routing table of the sender

The outline of candidate route selection, which corresponds to the static ranking procedure described above, is presented in Algorithm 4. First, we attempt to find routes to the recipient in the sender's routing table (which may be effective in the case of repeated payments, or if the sender is close to the recipient in terms of either hop distance or address distance). Then, we request the recipient's routing table, merge it with the sender's table and attempt to find routes in the united table. If this fails, we start requesting routing tables from nodes to which the sender has a known path, and where the node's address is close to the recipient in the address space.

Algorithm 4: Forming list of candidate routes

Input:

- Current node (payment sender) s
- Payment recipient r
- Maximum number of paths to find $k \geq 1$
- Maximum number of queried nodes $N_{tab} \geq 1$

Output: P, a set of routes from s to r ($P = \emptyset$ means refusal of routing)

Routine: Paths(u, v, E, k) finds at most k shortest paths from node u to node v in a graph G inferred from its set of edges E, i.e., $G = (\text{Nodes}(E), E)$. Yen's algorithm [33] could be used for this purpose; see also Section 5 for the discussion of the routine.

1: Initialize the combined routing table as the sender's table: $RT_{co} := s.RT$
2: Initialize the found routes $P := \emptyset$
3: Initialize $U := \emptyset$ the set of nodes close to the recipient for which we retrieved the routing table
4: **while** $|P| < k$ **and** $|U| < N_{tab}$ **do**
5: Find k shortest paths from the sender to the recipient using channels from RT_{co}:

$$P := P \cup \text{Paths}(s, r, RT_{co}, k)$$

6: **if** $|P| < k$ **then**
7: Retrieve the table of closest node to the recipient with regards to the address space, which we did not process yet, among nodes in the combined routing table:

$$c := \operatorname*{arg\,min}_{z \in \text{Nodes}(RT_{co}) \setminus U} \rho(z, r)$$

8: Request c's routing table $c.RT$ by sending **TABLE_REQ**. Merge it with the united routing table: $RT_{co} := RT_{co} \cup c.RT$
9: $U := U \cup \{c\}$
10: **return** P

We may filter the combined route table RT_{co} by the total channel capacity: if a certain channel $e \in RT_{co}$ has a lower capacity than the value being routed f, the channel should be ignored when finding shortest paths in RT_{co}. Correspondingly, message **TABLE_REQ** could contain f, meaning that

a node should return only channels in its routing table with total capacity exceeding or equal to f. On the other hand, filtering channels by capacity may be counterproductive if a payment is forwarded through multiple routes (see Section 5).

In practice, we may define a timeout t_{sel} for the total time spent on candidate route selection. If the algorithm takes too long to complete (e.g., due to computational difficulty or because of delays in network communication), it should be terminated, with the currently retrieved set of routes P considered final. Similarly, we may define a shorter timeout t_{tab} for the retrieval of a routing table on step 8. If the retrieval takes longer to accomplish, we assume that $c.RT = \emptyset$ and proceed to the next closest node c (indeed, it is guaranteed that the next execution of step 5 will recover no new route candidates, so we may skip it).

In order to mitigate DoS attacks and reduce network traffic, each node maintains a cache of routing tables recently received by **TABLE_RESP**, keyed by a node address and with an expiration time interval of several minutes. A node should ignore incoming **TABLE_REQ** from the same node if they are repeated more frequently. In order to further reduce the possibility of abusing **TABLE_REQ** messages, Algorithm 4 could be modified as follows: Instead of requesting routing tables from the nodes closest in the address space to the recipient, query nodes close to the *sender*. In this case, a node could check that it is close to **TABLE_REQ**'s sender in the address space and do not respond to the message otherwise. In the extreme case, node u may not respond to **TABLE_REQ** from node v unless the latter is a beacon for the node or has chosen the node as its beacon (i.e., $v \in u.bc \vee v \in u.rb$).

Parameter choices. The number of path candidates to find k should not be too small (as all candidate routes could be unusable) or too large (as it could make the algorithm too slow). $k \approx 10$ seems to be the reasonable default. For the same reasoning, $N_{tab} \approx 10$ could be a reasonable default number of nodes to query. $t_{sel} = 30$ seconds and $t_{tab} = 5$ seconds could be reasonable default timeout values.

Resource consumption. Denote V_{co} the nodes and E_{co} the channels in the combined routing table RT_{co}. Assuming a typical routing table size is 100 Kb and $N_{tab} = 10$, Algorithm 4 consumes no more than 1 MB of network traffic for the sender. Computational complexity comes mostly from the path finding algorithm and is $\mathcal{O}(k|V_{co}|(|E_{co}| + |V_{co}| \log |V_{co}|))$ in the worst case if Yen's algorithm is used. If graph inferred by RT_{co} is well connected, the expected complexity of Yen's algorithm is instead

$$\mathcal{O}(k \cdot \log |V_{co}| \cdot (|E_{co}| + |V_{co}| \log |V_{co}|)).$$

Memory consumption is determined by the path finding algorithm and is $\mathcal{O}(k|V_{co}| + |V_{co}|^2)$ in the case Yen's algorithm is used.

3.6 Ranking Candidate Routes

After Algorithm 4 gathers candidate routes P, they are ranked according to their dynamic characteristics. In order to save time, candidate ranking can start the process of searching for other candidates.

That is, on step 5 in Algorithm 4, each time a new candidate route p is found, the sender node launches its asynchronous ranking procedure, which computes the route ranking $rr(p) \in [-\infty, +\infty)$ (the more, the better; $-\infty$ corresponds to untraversable routes). We may further reduce the duration of the routing algorithm by comparing the returned ranking with a predefined "good" ranking threshold r_{good}; if $rr(p) \geq r_{good}$, then the corresponding route p is immediately selected as the final route to the recipient.

In order to obtain dynamic characteristics of candidate routes, we utilize a **RANK_POLL** message sent from the payment sender to the recipient along the candidate route. Polling messages use an onion-like, ballot box data structure in order to to preserve sender's and recipient's privacy according to requirements described in Section 1. The message contains the value to transfer (intermediate nodes would need it to calculate routing fees). The polling message collects:

- **For all channels:** the distribution of value in the channel authenticated by both channels participants
- **For all intermediate nodes:** forwarding fee to process the payment authenticated by the corresponding node. The node's authentication means that the node accepts its role as payment router and will reserve channel capacity to process the payment for a small amount of time (order of 1 minute).

Each intermediate node writes corresponding information to the relayed message; modern onion routing schemes allow for storing such information securely so that it can be decrypted only by the payment sender.

After a **RANK_POLL** message reaches the recipient node, it sends a response message back to the sender. Finally, collected dynamic data is used to determine the route rank (Algorithm 5).

Parameter choices. $t_{poll} \approx 5$ seconds seems to be a reasonable default, assuming route length 10 channels and a node taking 0.5 seconds to process a polling message.

Resource consumption. Onion routing messages could be reasonably compact – order of 100 bytes per hop. Thus, bandwidth consumed by Algorithm 5 is small. Algorithm 5 in itself has a negligible memory and CPU footprint both for the sender and for other nodes. However, channel verification on step 1 could be a resource-consuming operation as it requires access to the Bitcoin blockchain.

4 Simulations

We performed simulations on route discovery in order to understand if the proposed scheme would allow a node to obtain adequate network coverage with reasonable parameter values.

We provide detailed results for the network of $n = 2{,}000$ nodes, each having 4 neighbors on average. Then we give results for network of $n = 100{,}000$ nodes. We assume neighborhood scan radius $r_{nb} = 2$, maximal number of queried nodes during path search $N_{tab} = 10$, maximal number of candidate routes $k = 10$. The number of beacons N_{bc} varied from 0 to 12.

Algorithm 5: Candidate route ranking

Input:

- Value to transfer to the recipient f
- Route $p \in E^+$
- Polling message timeout t_{poll}

Output: route ranking $rr(p)$

Verify all channels in the route on the blockchain.
if any channel does not pass the check **then**
    return $-\infty$
Send a polling message with f along the route. Wait for the response.
if the message times out according to t_{poll} **then**
    return $-\infty$
if any channel cannot route the payment (including fee overhead) **then**
    return $-\infty$
Calculate the route rank according to cumulative fees on the route and other static and dynamic information. (In the simplest case, $rr(p) := 1/C$, where C is the cumulative fee.)
return route ranking $rr(p)$

It is difficult to estimate how the LN graph would end up looking. Thus, we have considered simple topologies in order to get a crude estimation of how routing could work and what algorithm parameters should be. Namely, we consider the Watts – Strogatz graph [34], where each node is joined with its 4 nearest neighbors in a ring, and the probability of rewiring each edge is 0.3.

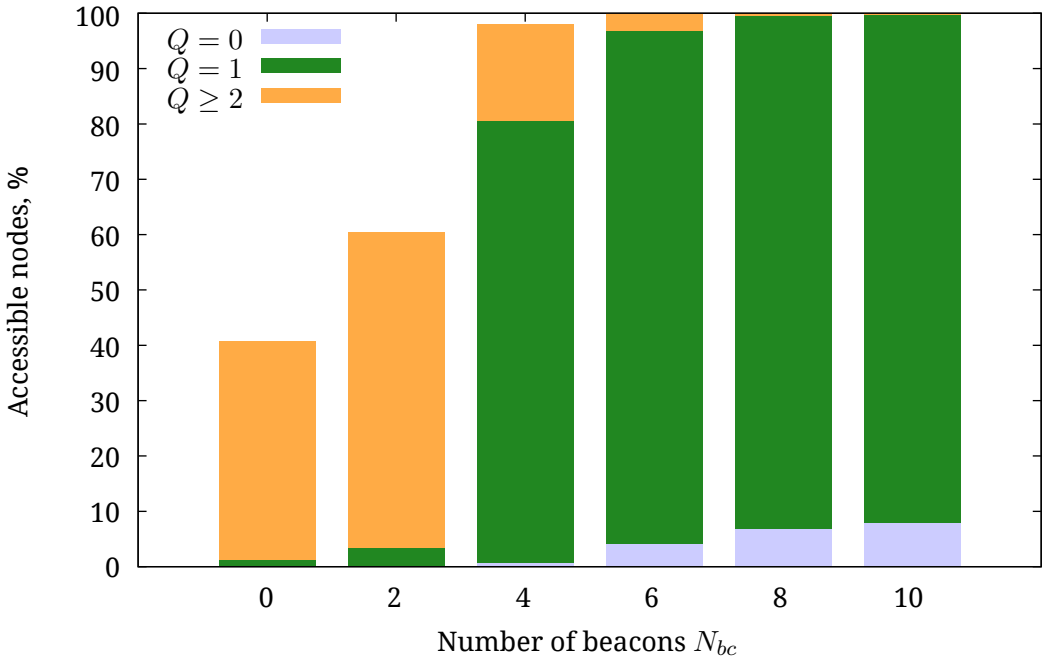
The experiment was conducted as follows:

1. Create channels among nodes according to the network topology assumptions.
2. Initiate the beacon search for each node selected from the node set in a random order.
3. Choose $n_{samp} = 10$ sample nodes. For each sample node, find paths to all other nodes.

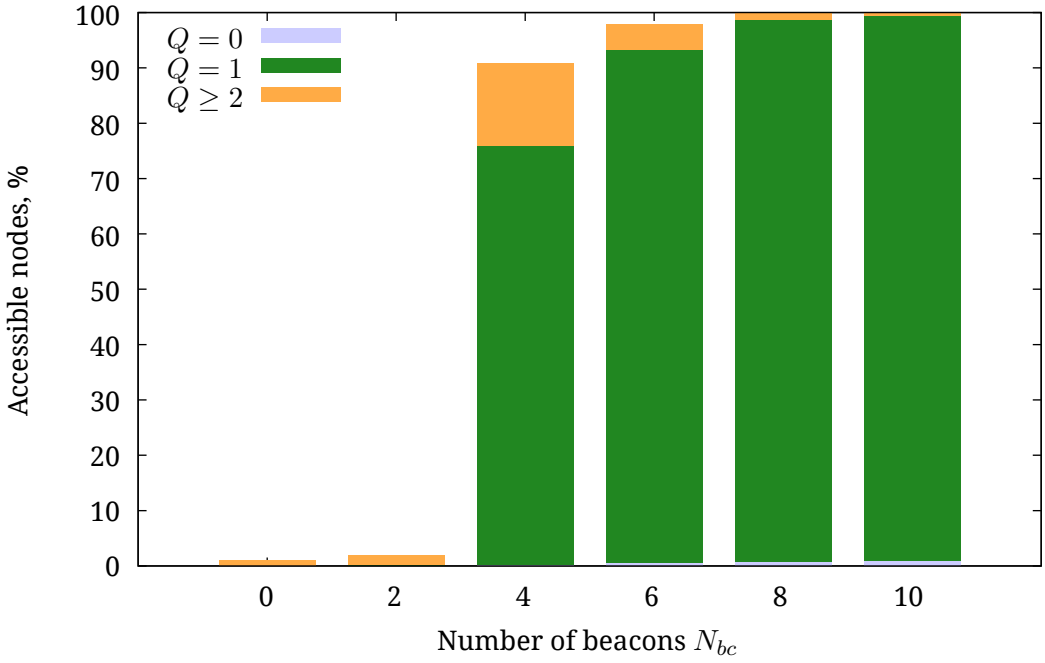
During simulation, we collected the number and the total size of messages aggregated by their type. The statistics for each message type was normalized:

- For messages related to neighborhood scan and beacon discovery, the number and size of messages was divided by the number of nodes n.
- For messages related to path finding, the number and size of messages was divided by the total number of searches $(n-1) \cdot n_{samp}$.

As the simulation shows, $N_{bc} = 5$ beacons is enough to find routes to all other nodes in a smaller graph; in a graph with 100,000 nodes, the effective minimum $N_{bc} = 6$ (Fig. 6). For a number of beacons exceeding the threshold, almost all paths are found on one **TABLE_REQ** query, i.e., when the sender merges his routing table with recipient's table.



(a) Graph with 2,000 nodes



(b) Graph with 100,000 nodes

Figure 6: Distribution of the number Q of **TABLE_REQ** queries necessary to discover a route among nodes

The routing algorithm yields paths longer than the ideal route obtained using Dijkstra algorithm with knowledge of the entire graph (Fig. 7). This is due to the fact that beacon finding is essentially depth-first search; thus, the routes found to beacons are longer than the shortest ones. After beacon discovery is finished for a node, it becomes possible to find shorter routes to beacons using the full routing table of the node. The excess length of paths to beacons obtained in this way is smaller, but it is still comparable to a graph diameter of 13.

The average size of a routing table increases with the number of beacons N_{bc}, but stays within acceptable limits even for large N_{bc} (Fig. 8). The main source of traffic is route finding and beacon search (Fig. 9). Note that increasing the number of beacons leads to a dramatic increase in beacon-related traffic.

5 Directions for Further Research

The routing algorithm described in Section 3 could be modified to increase its resilience and performance. We describe several potential modifications below.

Network setup. Setting up and maintaining a decentralized network with a reasonable level of privacy may require non-trivial solutions, as witnessed by the Bitcoin peer-to-peer overlay network. We briefly outlined a possible implementation of LN network logic that would use a distributed hash table (DHT) to maintain persistent information about LN nodes together with a mixed network overlay (Sphinx and HORNET). As alternative some information, such as channels existence proofs can be transmitted with routing messages. More work in this direction is required.

Node rating system. Route ranks (Section 3.6) could take into consideration social properties of nodes in addition to other dynamic characteristics. Examples of such properties are node uptime and total amount of payments routed through the node. The addition of such properties would require that other nodes (e.g, neighbors of a target node) attest to them; designing a Sybil-resistant node rating is a non-trivial task.

Anti-DoS measures. In a more simple case, a node could maintain a local score for the nodes it interacts with, similar to the DoS score in Bitcoin Core. The same score could be utilized to implement punitive measures against misbehaving nodes. The score system could be combined with anti-DoS message timeouts briefly described throughout Section 3; the applicability and efficiency of timeouts is a subject of another research.

Adjusting network parameters. The performance of the routing algorithm may be suboptimal if parameters (e.g., neighbor radius r_{nb}) differ across neighboring nodes in the network. A mechanism for nodes to communicate and agree on these parameters could be devised (e.g., a node could inform adjacent nodes about the values of routing parameters when establishing connections with these nodes).

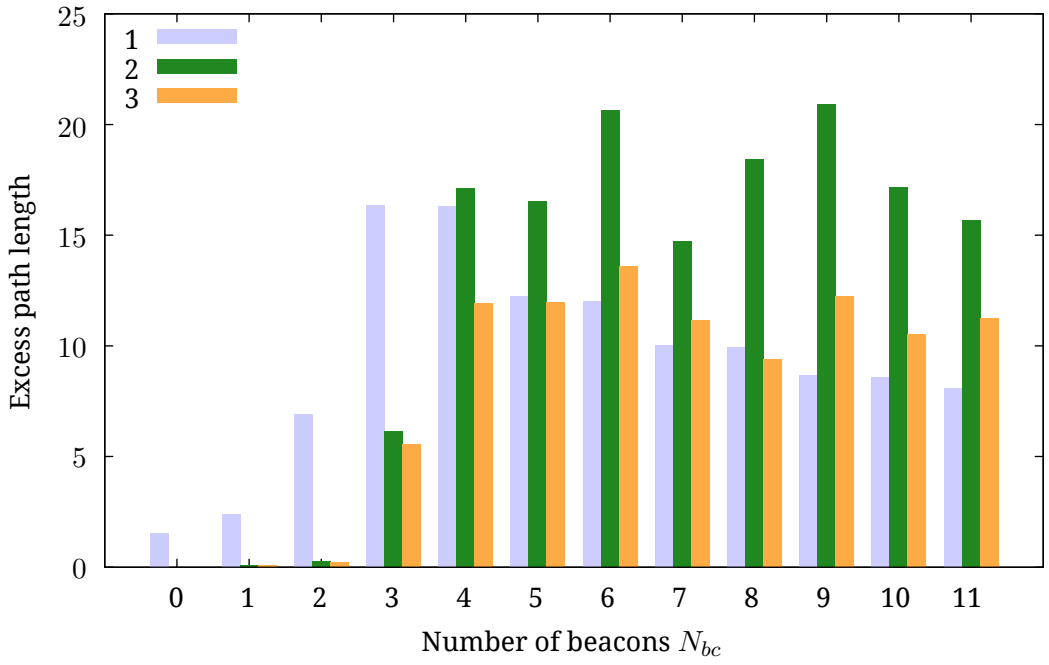


Figure 7: Average excess path length for a graph with $n = 2000$ nodes depending on the number of beacons: **1** – to all nodes in the network; **2** – to the beacons during beacon discovery; **3** – to the beacons after the beacon discovery is completed. The excess path length is defined as the difference in hops between the path found by the routing algorithm from Section 3 and the path obtained by applying Dijkstra algorithm with knowledge of the entire network. Probability of finding a path reaches almost 100% at 5 beacons; adding more beacons just increases awareness of the network and allows finding shorter paths.

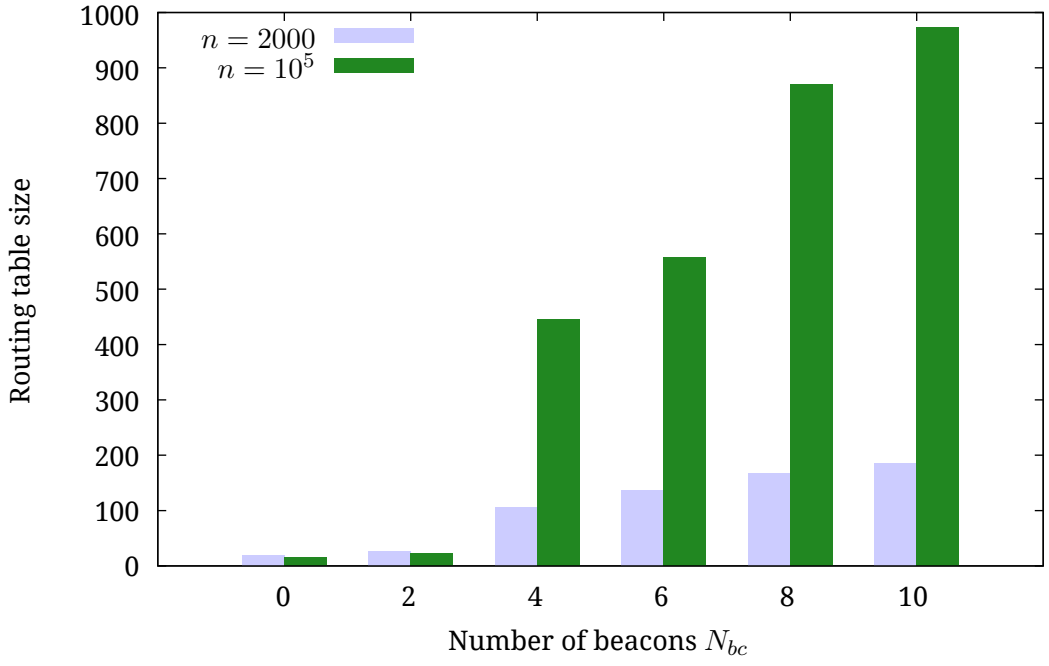
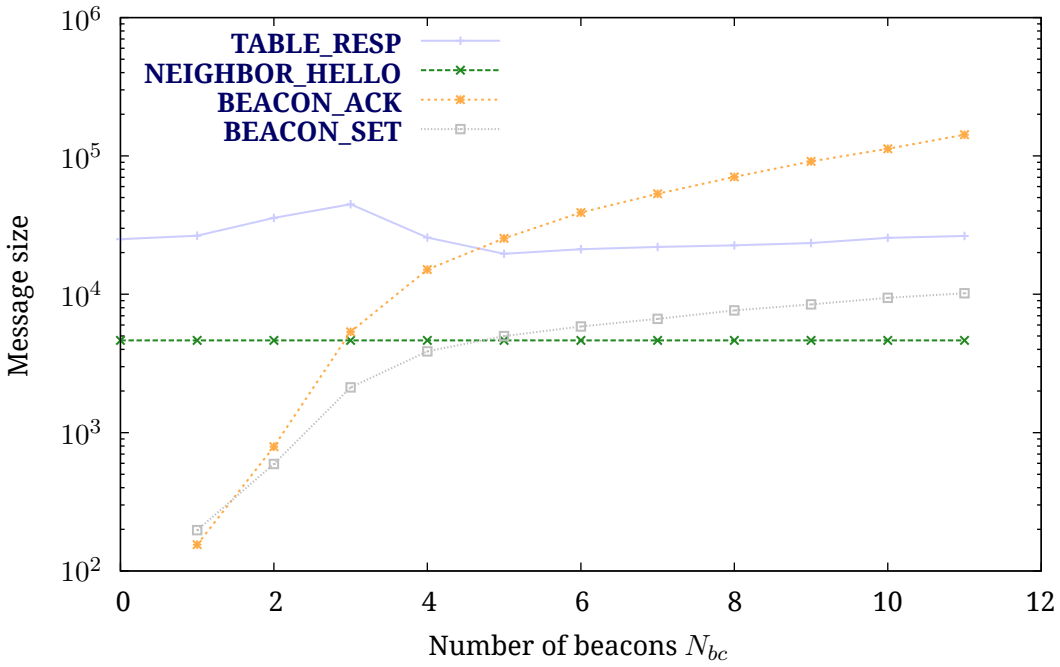
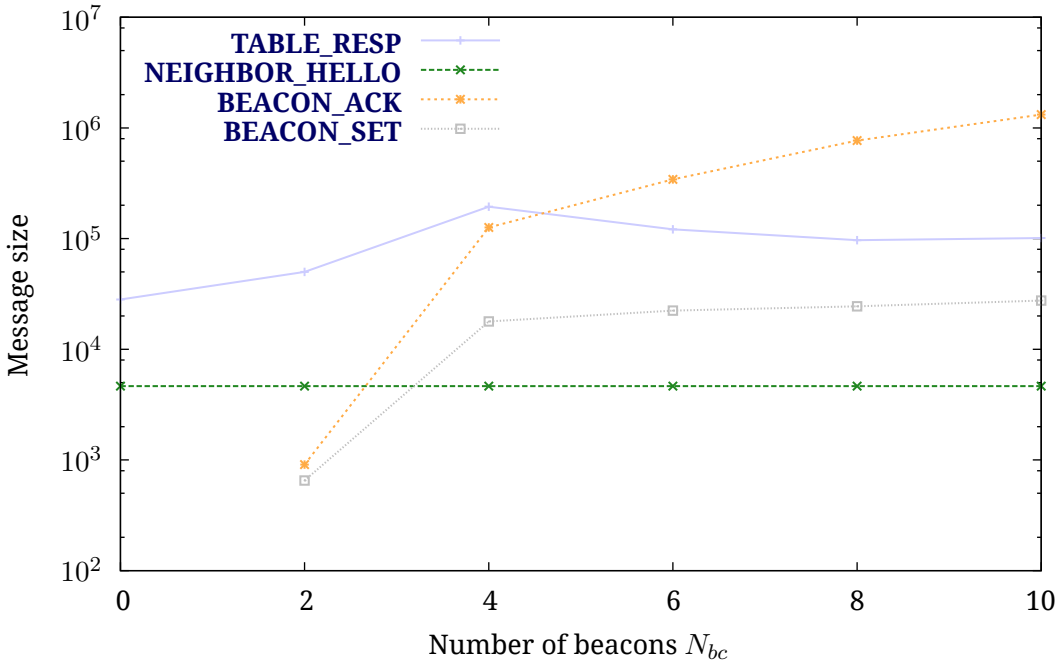


Figure 8: Size of routing table (number of nodes) depending on number of beacons, for a smaller and a larger studied graphs



(a) Graph with 2,000 nodes



(b) Graph with 100,000 nodes

Figure 9: Normalized total size of messages depending on the message type

Manually selected beacons. In the proposed routing algorithm, beacons for each node are selected automatically based solely on their proximity in the address space. With an update to the algorithm, we may introduce a mechanism for a node to advertise itself as a beacon, which could be beneficial for nodes with a significant number of channels.

Diverse k shortest paths. Yen's algorithm (or a similar approach) to find k shortest paths in a graph, which is used in route selection (Algorithm 4), may result in all candidate routes returned by the algorithm having a common node and/or a payment channel. This could mean that the failure of a single node may render all the routes found untraversable. To prevent selecting routes with a single point of failure, an optimal k-path selection would need to be *diverse:* each node or edge should be present in no more than $m < k$ of the selected paths. We are currently working on developing the corresponding algorithm.

Min cost flow problem to increase manageable payment size. The routing scheme we propose is best suited for small payments. The amount of value the network could transfer via a single payment may be increased if we consider minimum-cost flow problem [35] on the routes found during the proactive phase of the routing algorithm, i.e., split the payment and pass it through several channels.

Negative transaction fees. In order to balance channels with lopsided value distribution, we can introduce negative fees that would pay fees to other channels on the route. Nodes would be more likely to pick a channel with a negative fee during route selection; thus, the channel would balance its state increasing its ability to participate in routing.

Loops to balance the network. Short loops of channels with lopsided value distribution could be discovered locally by a node participating in such a loop. After discovery, channels in the loop could be balanced by sending a payment along the loop in the direction opposite to channel exhaustion.

6 Discussion

In this paper, we have proposed a hybrid source routing algorithm that may be used to forward payments in the peer-to-peer version of Lightning Network in Bitcoin. The algorithm consists of two phases:

- Proactive update of the node's routing table, which stores information about long-term network topology (i.e., existence of payment channels between nodes)
- Reactive collection of information based on a routing request, which collects dynamic information about LN topology (e.g., distribution of funds and forwarding fees for channels) and determines a list of viable payment routes optimized according to certain criteria

To expand awareness of LN nodes of the network topology, the proactive phase utilizes the concept of node beacons, which is similar to earlier concepts of LN routing and to beacons in cjdns. The algorithm is compatible with onion routing (thus providing a reasonable level of privacy for its users) and is trustless (i.e., its operation does not critically rely on presence of trusted parties in the network).

Due to design decisions (e.g., usage of beacons), the proposed routing is probabilistic. The routing algorithm is not guaranteed to find a path between the sender and the recipient in the most general case; however, the probability of a path discovery is kept at a reasonably high level. This property may be acceptable in LN, as the network architecture provides an option to open a new channel (thus benefiting the connectivity of the network) or to send money on-chain. Fallback payment processing could be integrated into LN clients, so that it would be easy for a user to create a new payment channel in the case a payment cannot be forwarded through channels discovered by the routing algorithm.

We estimate that the algorithm could scale for at least millions of network users. As simulations show, it is enough to store paths to about the square root of the number of nodes in the network (provided nodes are reasonably well connected). For a network with 100,000 nodes, it is enough to store paths to just 600 nodes.

One of the unsolved issues with the algorithm (and LN itself) is that for the routing to work, a significant number of nodes should be continuously online. This requirement is absent in Bitcoin and virtually all centralized payment systems. On the other hand, LN provides a measurable incentive for nodes to stay online, as an online node could earn transaction fees by routing payments through it.

Finally, in the initial days of LN, we may encounter a kind of "bootstrapping problem". For the network to operate, a sufficient number of channels with enough value stored in them must be created. At the same time, unless the network functions efficiently, there may be little incentive for people to put their Bitcoin in LN channels. Ideally, a wide array of industry and community participants would run LN nodes, helping to bootstrap the network. As a result, new users connecting to LN would get routing capabilities from day one, and could then route some of the network traffic themselves, thereby expanding the network. If enough nodes participate to enable a scale-free, decentralized network, it may be possible to avoid excessive LN centralization from the start.

Acknowledgements

We would like to thank Joseph Poon, Thaddeus Dryja, and Elizabeth Stark for their cooperation and their contributions to the routing algorithm. Bitfury Group would like to thank Olaoluwa Osuntokun for designing the channel verification procedure described in Appendix A.

Appendix A Channel Authentication in LN

In order to thwart adversaries seeking to tie up the funds of a node by advertising fake, non-existent LN channels, for each channel advertised by a node there must exist a proof on the blockchain based

on a funding transaction with a sufficient number of confirmations. We require that such channel proofs be strongly unforgeable. Additionally, we also require that for each channel, only a *single* pair of identities on the network can produce a valid proof of channel existence. In order to do this, we require cryptographic binding between the identity public keys, and channel funding multisig keys.

In addition to the identity public keys of two nodes, A_1 and A_2, each node generates two new keys (independent of their identity keys), B_1 and B_2. We shall refer to B_i, $i \in \{1, 2\}$, as a *channel derivation point* (CDP). Using this set of independent keys, the channel keys C_i for each node are computed as $C_i = A_i + B_i$ in the additive notation.

Note that currently, either A_i or B_i may be advertised as an identity key for a node. In order to ensure only A_i can be advertised as a valid identity point, we enforce an additional constraint on both the identity key and CDP. Globally within the network, all identity keys have an *even* y-coordinate and all CDPs have an *odd* y-coordinate. Additionally, with this modification only the x-coordinates of all points need to be transmitted, resulting in a 32-byte serialization, rather than a 33-byte serialization which includes an identifier byte.

The rationale for this separation is threefold.

- First, the independence of identity keys and channel keys allows for offline signing of updates to the channel state, opening the possibilities for a portion of the channel balance to be secured by cold storage.
- Second, the split compartmentalizes the implications of key compromise. In the case the identity key is compromised, the attacker can authenticate on behalf of the node, but not sign for new channel states. In the case of a compromise of the channel key, the attacker can only sign for that particular channel, not any past or future channels, and is also unable to authenticate on behalf of the node.
- Finally, requiring the attestation to come from the computed channel key binds both the identity key and CDP for each node. This serves to eliminate a class of attack which would allow an attacker to create fraudulent channel proofs or allow colluding nodes to advertise their channel several times, creating a non-canonical channel graph.

A.1 Proof Format

In order to authenticate an open channel, the following information must be provided:

- The outpoint (transaction ID and the output index) of the funding output
- Both channel derivation points supplied by each node participating in the channel
- The identity public keys of both nodes within the network
- Signature of all of the above authenticated by both derivation points and both identity keys in the form of a single aggregated Schnorr signature [36].

Optionally, in order to facilitate the existence of SPV Bitcoin clients within the network, an SPV proof of the inclusion of the funding transaction within a block may be provided.

In order to uniquely determine the signed information, we need to introduce deterministic ordering between the channel nodes. As an example, we may assume that the x-coordinate of the identity public key of the first node needs to be smaller than the x-coordinate of the identity public key of the second node: $A_1.x < A_2.x$.

Elliptic curve (EC) Schnorr signatures are used due to their compact nature, support for public key recovery, and group signature aggregation. The aggregation property means that a single Schnorr signature could be used instead of a Bitcoin ECDSA 4-of-4 multisignature authenticated by 2 derivation points and 2 identity keys. We adhere to a variation of EC Schnorr signatures from the secp256k1 library [37] bundled with Bitcoin Core, which uses the eponymous elliptic curve; in principle, the same approach could be used to construct and verify signatures using other elliptic curves.

Given a public key X with private key x, a Schnorr signature of a message digest m on the elliptic curve with the generator G and prime order n is computed as [38]

$$\langle \text{signature} \rangle \stackrel{\text{def}}{=} (R.x, s); R = kG; s = [k - H(R.x \parallel m) \cdot x] \bmod n,$$

where

- $k \in [1, n)$ is a cryptographic nonce
- $R.x$ denotes the x coordinate of the EC point R
- H is a secure hash function
- $\parallel$ denotes byte sequence concatenation

The implementation [38] requires that R has the even y-coordinate; thus, R can be uniquely restored from $R.x$ alone. This results in 64-byte signatures for secp256k1 (both $R.x$ and s contain 32 bytes) that support public key recovery without the need for an additional identification byte. Correspondingly, ignoring the optional SPV proof, an open channel proof is 228-bytes. If 4 individual signatures were used rather than a group signature, the size of a channel proof would nearly double.

A Schnorr signature $(R.x, s)$ could be verified by checking the group identity

$$sG = R - H(R.x \parallel m) \cdot X.$$

The same identity could be used to recover the public key X. Indeed, $X = q(R - sG)$, where q is the inverse of the hash value in the prime field $\mathbb{Z}_n$: $q = H(R.x \parallel m)^{-1} \bmod n$.

In order to verify a channel proof, a node executes Algorithm 6. To facilitate quick verification on step 1, a node should either have a copy of the blockchain indexed by the transaction identifier, or some other means to quickly locate a transaction given its identifier.

An aggregated signature serving as the input for Algorithm 6 could be conjointly generated by the nodes participating in the channel, using the two-phase protocol (Algorithm 7).

Algorithm 6: Channel proof verification

Input:

- Funding outpoint $p = (txid, idx)$, where
  - $txid$ is the transaction identifier
  - idx is the output index
- Channel derivation points B_1, B_2
- Identity public keys A_1, A_2, $A_1.x < A_2.x$
- Schnorr signature $(R.x, s)$

Output: **true**, iff the channel proof is valid

Locate the funding transaction within the blockchain using the specified $txid$
if unable to locate the transaction **then**
    return false
Using A_i and B_i, $i \in \{1, 2\}$, compute $C_i := A_i + B_i$
Using C_1 and C_2, reconstruct the 2-of-2 multisig redeem script: **OP_2** C_1 C_2 **OP_CHECKMULTISIG**. Obtain the script hash of the reconstructed redeem script using SHA-256
Compute a P2WSH (pay-to-witness-script-hash) [39] public key script $pub :=$ **OP_0** ⟨32-byte hash⟩
if pub does not match the public key script at p **then**
    return false
if exists a transaction spending the output p **then**
    return false ▷ the channel is closed
Compute message digest $m :=$ SHA256$(A_1 \parallel A_2 \parallel B_1 \parallel B_2 \parallel p)$, with all 4 public keys encoded as compressed points
Using the signature $(R.x, s)$, perform public key recovery to obtain the group channel public key Q, which serves to uniquely identify a channel link globally within the network
if $Q \neq C_1 + C_2 \equiv A_1 + A_2 + B_1 + B_2$ **then**
    return false ▷ invalid signature
return true

1. In the first phase, both sides generate their respective signing nonces k_i, and from that generate a point $R_i = k_i G$. Both R_i points will be combined into a single point R_g which constitutes one half of the group signature. In order to ensure both arrive at the same R_g point, the first node according to the introduced deterministic ordering commits to R_1, opening this commitment after she receives R_2 from the other node.
2. In the second phase, a partial signature $(R_g.x, s_i)$ is created by both sides and then exchanged in order to form the full group signature $(R_g.x, s_g)$.

The protocol requires 2 cumulative round trips: 1.5 round trips for phase 1, and 0.5 round trip from each side for phase 2.

Algorithm 7: Channel group signature generation

Input:

- Secure hash function H (instantiated as SHA-256)
- Elliptic curve with generator G and order n (instantiated as secp256k1)
- Funding outpoint $p = (txid, idx)$, where
  - $txid$ is the transaction identifier
  - idx is the output index
- For the first node: CDP B_1, identity public key A_1, and channel key pair (C_1, c_1)
- For the second node: CDP B_2, identity public key A_2, and channel key pair (C_2, c_2)
- Ordering of nodes defined by $A_1.x < A_2.x$

Output: aggregated Schnorr signature $(R_g.x, s_g)$

First node:

1: Generate a random nonce $k_1 \in [1, n)$; compute $R_1 := k_1 G$
2: **if** R_1 has an odd y-coordinate **then**
3: Reassign $k_1 := -k_1$; $R_1 := -R_1$
4: Send $H(R_1)$ to the second node
5: Receive R_2; send R_1 to the second node
6: (skip)

Second node:

1: Generate a random nonce $k_2 \in [1, n)$; compute $R_2 := k_2 G$
2: **if** R_2 has an odd y-coordinate **then**
3: Reassign $k_2 := -k_2$; $R_2 := -R_2$
4: Send R_2 to the first node
5: (wait)
6: Receive R_1; verify that it corresponds to the commitment

Both nodes:

7: Compute message digest $m :=$ SHA256$(A_1 \parallel A_2 \parallel B_1 \parallel B_2 \parallel p)$, with all 4 public keys encoded as compressed points
8: Compute $R_g := R_1 + R_2$
9: Compute $e := H(R_g.x \parallel m)$
10: **if** $e = 0$ **or** $e \geq n$ **then**
11: **restart** the protocol
12: Compute $s_i := k_i - c_i e$, $i \in \{1, 2\}$
13: Exchange s_i
14: **return** $(R_g.x, s_g) \equiv (R_g.x, s_a + s_b)$

References

[1] Cryptocurrency Market Capitalizations
URL: http://coinmarketcap.com

[2] Script. In: Bitcoin Wiki
URL: https://en.bitcoin.it/wiki/Script

[3] *Meni Rosenfeld* (2012). Analysis of hashrate-based doublespending
URL: https://bitcoil.co.il/Doublespend.pdf

[4] *Joseph Poon, Thaddeus Dryja* (2016). The Bitcoin Lightning Network: scalable off-chain instant payments
URL: http://lightning.network/lightning-network-paper.pdf

[5] *Jarl Fransson* (2015). A protocol for microtransactions: using Bitcoin with a medieval money contract
URL: https://www.strawpay.com/docs/stroem-payment-system.pdf

[6] (2015). Inter-channel payments
URL: http://impulse.is/impulse.pdf

[7] *Christian Decker, Roger Wattenhofer* (2016). A fast and scalable payment network with Bitcoin duplex micropayment channels.
URL: http://www.tik.ee.ethz.ch/file/716b955c130e6c703fac336ea17b1670/duplex-micropayment-channels.pdf

[8] *Rusty Russell* (2015). Reaching the ground with Lightning
URL: https://github.com/ElementsProject/lightning/blob/master/doc/deployable-lightning.pdf

[9] Elements Project Lightning Network implementation
URL: https://github.com/ElementsProject/lightning

[10] Onion routed micropayments for the Bitcoin Lightning Network
URL: https://github.com/LightningNetwork/lightning-onion

[11] P2P Network to send off-chain Bitcoin payments
URL: https://github.com/matsjj/thundernetwork

[12] Experimental Lightning node
URL: https://github.com/hashplex/Lightning

[13] *Chris Pacia* (2015). Lightning Network skepticism.
URL: https://chrispacia.wordpress.com/2015/12/23/lightning-network-skepticism/

[14] Spoke hub distribution paradigm. In: English Wikipedia
URL: https://en.wikipedia.org/wiki/Spoke%E2%80%93hub_distribution_paradigm

[15] Source routing. In: English Wikipedia
URL: https://en.wikipedia.org/wiki/Source_routing

[16] *Satoshi Nakamoto* (2008). Bitcoin: a peer-to-peer electronic cash system.
URL: https://bitcoin.org/bitcoin.pdf

[17] *Anthony Towns* (2015). Network topology and routing. In: Lightning Network development discussion
URL: http://lists.linuxfoundation.org/pipermail/lightning-dev/2015-September/000188.html

[18] *Rusty Russel* (2015). Ionization protocol: flood routing. In: Lightning Network development discussion.
URL: http://lists.linuxfoundation.org/pipermail/lightning-dev/2015-September/000199.html

[19] *Amos Bairn* (2015). Ionization protocol: flood routing. In: Lightning Network development discussion.
URL: http://lists.linuxfoundation.org/pipermail/lightning-dev/2015-September/000212.html

[20] Mobile ad hoc network. In: English Wikipedia
URL: https://en.wikipedia.org/wiki/Mobile_ad_hoc_network

[21] *Elizabeth Royer, Chai-Keong Toh* (1999). A review of current routing protocols for ad hoc mobile wireless networks In: IEEE Personal Communications, Vol. 6 (2).
doi:10.1109/98.760423

[22] *Thomas Clausen, Philippe Jacquet et al.* (2003). Optimized Link State Routing Protocol (OLSR). No. RFC 3626
URL: https://tools.ietf.org/html/rfc3626

[23] *Charles Perkins, Elizabeth Belding-Royer* (2003). Ad hoc on-demand distance vector (AODV) routing. No. RFC 3561
URL: https://tools.ietf.org/html/rfc3561

[24] *David B. Johnson* (1994). Routing in ad hoc networks of mobile hosts. In: Mobile Computing Systems and Applications (WMCSA 1994), First Workshop on, pp. 158–163.
doi:10.1109/WMCSA.1994.33

[25] *Rudolf Riedi et al.* (2005). Safari: A scalable architecture for ad hoc networking and services. Research report TR04-433.

[26] *Caleb James DeLisle* (2015). cjdns white paper
URL: https://github.com/cjdelisle/cjdns/blob/master/doc/Whitepaper.md

[27] *Olaoluwa Osuntokun* (2015). An alternative onion-routing proposal. In: Lightning Network development discussion
URL: http://lists.linuxfoundation.org/pipermail/lightning-dev/2015-December/000384.html

[28] *Georg Danezis* (2009). Sphinx: a compact and provably secure mix format
URL: http://www.cypherpunks.ca/~iang/pubs/Sphinx_Oakland09.pdf

[29] *Chen Chen et al.* (2015). HORNET: High-speed onion routing at the network layer
URL: http://www.scion-architecture.net/pdf/2015-HORNET.pdf

[30] *Roger Dingeldine, Nick Mathewson, Paul Syverson* (2004). Tor: the second-generation onion router
URL: http://www.onion-router.net/Publications/tor-design.pdf

[31] *National Institute of Standards and Technology* (2015). Secure Hash Standard (SHS).
doi:10.6028/NIST.FIPS.180-4

[32] *Petar Maymounkov, David Mazieres* (2002). Kademlia: a peer-to-peer information system based on XOR metric. In: Peer-to-Peer Systems, First International Workshop, IPTPS 2002, pp. 53–65.
doi:10.1007/3-540-45748-8_5

[33] *Jin Y. Yen* (1971). Finding the k shortest loopless paths in a network. In: Management Science, Vol. 17 (11), pp. 712–716
doi:10.1287/mnsc.17.11.712

[34] *Duncan J. Watts, Steven H. Strogatz* (1998). Collective dynamics of 'small-world' networks. In: Nature, vol. 393, pp. 440–442.
doi:10.1038/30918

[35] Minimum-cost flow problem. In: English Wikipedia
URL: https://en.wikipedia.org/wiki/Minimum-cost_flow_problem

[36] *Claus-Peter Schnorr* (1989). Efficient identification and signatures for smart cards. In: Proc. Advances in cryptology – CRYPTO'89, pp. 239–252
doi:10.1007/0-387-34805-0_22

[37] Optimized C library for EC operations on curve secp256k1 repository
URL: https://github.com/bitcoin/secp256k1/

[38] Schorr signature source code
URL: https://github.com/bitcoin/secp256k1/blob/master/src/modules/schnorr/schnorr_impl.h

[39] *Eric Lombrozo, Johnson Lau, Pieter Wuille* (2015). Segregated witness (consensus layer)
URL: https://github.com/bitcoin/bips/blob/master/bip-0141.mediawiki