Flash: Efficient Dynamic Routing for Offchain Networks

Peng Wang
City University of Hong Kong
Hong Kong, China
pewang4-c@my.cityu.edu.hk

Hong Xu
City University of Hong Kong
Hong Kong, China
henry.xu@cityu.edu.hk

Xin Jin
Johns Hopkins University
Baltimore, USA
xinjin@cs.jhu.edu

Tao Wang*
New York University
New York, USA
tw1921@nyu.edu

ABSTRACT

Offchain networks emerge as a promising solution to address the scalability challenge of blockchain. Participants make payments through offchain networks instead of committing transactions on-chain. Routing is critical to the performance of offchain networks. Existing solutions use either static routing with poor performance or dynamic routing with high overhead to obtain the dynamic channel balance information. In this paper, we propose Flash, a new dynamic routing solution that leverages the unique transactions characteristics in offchain networks to strike a better tradeoff between path optimality and probing overhead. By studying the traces of real offchain networks, we find that the payment sizes are heavy-tailed, and most payments are highly recurrent. Flash thus differentiates the treatment of elephant payments from that of mice payments. It uses a modified max-flow algorithm for elephant payments to find paths with sufficient capacity, and strategically routes the payment across paths to minimize the transaction fees. Mice payments are sent directly by looking up a routing table with a few precomputed paths to reduce probing overhead. Testbed experiments and trace-driven simulations show that Flash improves the success volume of payments by up to 2.3x compared to the state-of-the-art routing algorithm.

CCS CONCEPTS

• **Networks** → **Network protocol design**; **Peer-to-peer networks**.

KEYWORDS

Blockchain, Off-chain, Payment Channels, Routing

ACM Reference Format:
Peng Wang, Hong Xu, Xin Jin, and Tao Wang. 2019. Flash: Efficient Dynamic Routing for Offchain Networks. In *The 15th International Conference on emerging Networking EXperiments and Technologies (CoNEXT '19), December 9–12, 2019, Orlando, FL, USA.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3359989.3365411

1 INTRODUCTION

Blockchain is the fundamental infrastructure to support a new generation of decentralized Internet applications. It has enabled many innovations from cryptocurrencies to smart contracts [25, 35]. *Scalability* is the primary challenge for blockchain to support decentralized applications at scale [18, 23, 25, 27, 32]. As a concrete example, Bitcoin only processed fewer than 20 transactions per second at peak from November 2018 to January 2019 [12], whereas Visa was reported to process more than 47,000 transactions per second at peak during the 2013 holiday seasons [27].

Blockchain is intrinsically difficult to scale because it aims to achieve global consensus between all participants, which involves complex protocols to consistently replicate all state changes. Despite many efforts to improve the efficiency and reduce the overhead of blockchain protocols [32, 37], their performance is ultimately limited by the network bandwidth and processing capability of the participants to replicate state changes.

Offchain networks (or payment channel networks) emerge as one of the most promising solutions to solve this dilemma [18, 23, 27]. Two participants can use a bidirectional payment channel to make *multiple* payments with each other, without the need to commit every transaction to the blockchain. The blockchain is only involved when the participants set up and tear down the payment channel, or when they have disagreements on the transaction results. A network of payment channels form an *offchain network*, and allows two participants without direct channels to transact via a multi-hop path. A transaction can be safely committed in an offchain network as soon as the participants on the payment path achieve an agreement using a multisignature contract such as a Hashed Timelock Contract (HTLC) [27]. This obviates the need to consistently commit the transaction to *every* participant on the chain. As a result, offchain networks have the potential to significantly improve the transaction throughput and reduce the transaction latency of blockchain. Examples including Lightning Network [10],

---

[*] The work was done when Tao was with City University of Hong Kong.

CoNEXT '19, December 9–12, 2019, Orlando, FL, USA

ACM ISBN 978-1-4503-6998-5/19/12...$15.00
https://doi.org/10.1145/3359989.3365411

Raiden Network [11], and Ripple [9] are increasingly being adopted and used in practice.

Routing is critical for offchain networks to fulfill their promise. Efficient routing can successfully deliver most payments in an offchain network, minimizing the operations on the blockchain. While routing is a well-studied problem in computer networking, there is a key difference between an offchain network and a computer network. In a computer network, each link has static capacity which does not change as the link sends packets. However, in an offchain network, the initial balance of a payment channel (i.e. channel capacity) is deposited by the participants during the channel setup, and the balance is updated after *every* payment in the channel. As a result, offchain networks are more dynamic than computer networks. The balance dynamics of the payment channel are described in detail in §2.1.

At a high level, there are two major classes of protocols for network routing. The first uses static routing where the path for each flow is fixed after (periodical) path discovery. Many traditional routing protocols such as OSPF and IS-IS fall into this class. Static routing guarantees reachability, and is typically used when the network topology and traffic are mostly static, or if the network capacity is abundant and the performance is not a concern. Early routing protocols for offchain networks, such as Flare [28], SlientWhispers [24] and SpeedyMurmurs [29], leverage this approach. But they suffer from low transaction throughput, because they do not consider dynamic channel balance in offchain networks. The limitation of static routing motivates the second class of protocols that use dynamic routing, where the path for each flow or flowlet is dynamically updated based on real-time network characteristics. Many emerging solutions in datacenters and inter-datacenter WANs [14, 20, 21, 31, 33] fall into this class. Spider [30] applies dynamic routing to offchain networks and achieves higher performance than earlier static protocols.

Dynamic routing, however, is not a panacea. It is well-known that there exists a trade-off between path optimality and probing overhead. Using an optimal path comes at the cost of probing the network in the first place. This is especially true for offchain networks, as the channel balance changes after each payment, and one needs to pay the probing overhead for every single payment if an optimal path was to be chosen.

Classical solutions in computer networking suggest to strike a balance between path optimality and probing overhead by differentiating the treatment of elephant flows from that of mice flows [13, 16]. A small number of elephant flows are dynamically scheduled on different paths for high performance, and the vast mice flows are randomly mapped to static paths for low probing overhead. Realizing the idea of elephant flow routing in offchain networks is challenging for at least two reasons.

- First, we need to understand whether elephant flow routing is suitable for offchain networks. If all payments in offchain networks have similar size, then there do not exist elephants in the first place. Further, if mice payments are highly scattered, i.e. a sender involves a different receiver every time to send a payment, significant probing overhead for mice payments still can not be avoided.
- Second, we need to design an efficient protocol to satisfy the unique requirements of offchain networks. Because offchain networks are more dynamic than computer networks, elephant payments need to probe more paths aggressively in order to find efficient routing. The problem is exacerbated by the distributed nature of offchain networks, unlike datacenter networks and backbone WANs that have a control plane to coordinate participants.

To address these challenges, we first conduct a measurement study on the payment traces of two real-world blockchain networks, Ripple [9] and Bitcoin [3]. By analyzing the traces, we find two important characteristics of cryptocurrency transactions. First, the payment sizes exhibit heavy-tailed distributions. Most transactions are small, while a small number (<10%) of transactions constitute most (over 94%) of the total transaction volume. This demonstrates the existence of elephant transactions in offchain networks. Second, payments are highly recurrent. Within a period of 24 hours, over 80% transactions happen between existing pairs of participants. This suggests that most mice payments can leverage existing paths with no extra probing overhead, instead of discovering new paths.

Based on these findings, we design Flash, a new dynamic routing solution for offchain networks. Different from Spider [30] which uses a fixed set of paths for each payment, Flash separates elephant payments from mice to achieve higher throughput with low probing overhead. Flash uses a modified max-flow algorithm based on Edmonds-Karp [15] to probe and find candidate paths with sufficient capacity for elephant payments, and carefully distribute the payments over these paths to minimize transaction fees (charged by intermediate nodes). Mice payments are sent through a few existing paths if they have already been computed to minimize the need of probing. Payments are divided into trunks to be sent through multiple paths when needed, since the atomicity of the transactions can be ensured with recent proposals such as Atomic Multi-Path Payments (AMP) [1].

In summary, we make the following contributions.

- We perform a measurement study on the payment traces of real blockchain networks to understand the traffic characteristics of cryptocurrency transactions.
- We design Flash, a new routing protocol for offchain networks that differentiates elephant payments from mice in order to achieve high throughput with low probing overhead.
- We implement a prototype of Flash on a cluster of commodity servers. Testbed experiments and trace-driven simulations show that Flash improves the network throughput by up to 2.3x compared to the state-of-the-art routing algorithms. The code of Flash and the offchain network traces are open source on Github.[1]

2 BACKGROUND AND MOTIVATION

In this section, we first give a brief introduction of offchain networks, and then motivate our design with our findings in real traces from Ripple [9] and Bitcoin [25].

2.1 A Primer on Offchain Networks

Payment channels. Payment channels are a basic building block of offchain networks. A payment channel is established between

---

[1] https://github.com/NetX-lab/Offchain-routing-traces-and-code

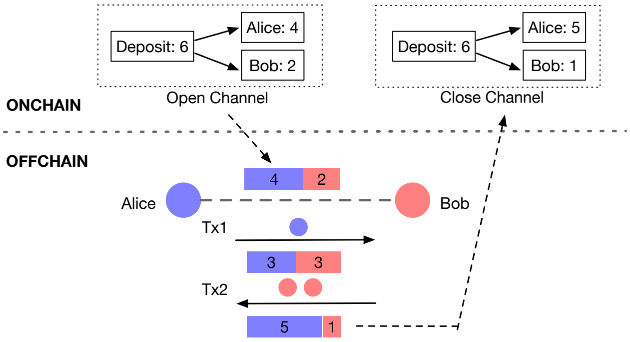


Figure 1: A payment channel between Alice and Bob. Alice and Bob deposit 4 and 2 satoshis respectively to open a channel. Two payments are made off the chain. Alice first pays Bob 1 satoshi, and then receives 2 satoshis from Bob. Finally, the channel is closed by committing the latest state to the chain.

two parties, and allows them to make multiple payments without the need to commit every payment to the chain. To ensure the offchain security, both parties maintain a multi-signature address which guarantees that any balance updates on the channel require mutual agreement. The chain is only involved when there is a dispute regarding current balance or setting up and tearing down the channel. By moving payments away from the chain, it reduces computation and replication overhead, improves transaction throughput, and lowers latency. Furthermore, because sending payments over payment channels does not need to reward "miners", payment channels provide more competitive transaction fees and better support the mice payments.

A toy example in Figure 1 illustrates the basic operations of a payment channel using bitcoin as the cryptocurrency. To open a channel, Alice and Bob make a transaction on the chain in which they deposit funds to the channel. The channel is established after the transaction is committed to the chain. In this example, Alice funds the channel with 4 satoshis and Bob with 2 satoshis (Satoshi is the smallest unit of bitcoin). At this point, the balance—which limits the maximum amount of bitcoin one party can send to the other—becomes 4 satoshis for Alice and 2 satoshis for Bob. The balance of each party is then updated after each successful transaction executed based on mutual agreement. Thus if Alice pays Bob 1 satoshi, both would have a balance of 3 satoshis. As long as the channel remains open and the payment from one to the other does not exceed its balance, Alice and Bob can repeatedly make any number of transactions. Finally, Alice and Bob can choose to close the channel if no more transactions are needed. The final state of the channel is committed to the chain, and both parties can withdraw their balances.

Offchain networks. It is clearly impractical for a user to open a payment channel with whoever it needs to transact with; the channel opening cost and the latency of doing this on the chain would be prohibitive. Payment channel networks, or *offchain networks*, are therefore developed to support indirect payments between two parties who do not have a payment channel. An offchain network is composed of many payment channels. Figure 2 shows an example of a simple offchain network with two payment channels. Two parties can make a transaction so long as there is a directed path between them and the payment amount is no larger than the minimum channel balance of the path. In order to guarantee the atomicity and security of payments via multiple payment channels, an offchain network usually relies on the hash time-locked contracts (HTLC) [27]. For example, if Alice wants to pay Bob 1 satoshi through Charlie as in Figure 2, HTLC guarantees that Charlie receives funds from Alice if and only if Bob receives the payment from Charlie successfully. Otherwise the funds are returned to the payer Alice. HTLC also guarantees that either the balances of all channels on the path are updated or none is updated after the transaction.

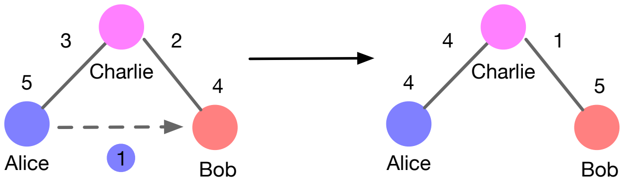


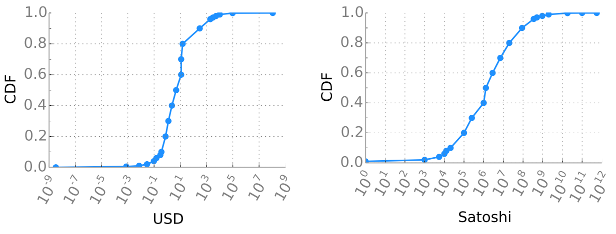
Figure 2: An indirect payment on a payment channel network. Alice can pay Bob 1 satoshi through Charlie, but cannot do more than 2 satoshis since the payment channel from Charlie to Bob has a balance of 2 satoshis.

Offchain networks have seen rapid development over recent years and is increasingly adopted in many scenarios. Lightning Network [10], Raiden Network [11], and Ripple [9] are prominent examples in practice. Lightning and Raiden are offchain networks for Bitcoin and Ethereum, two of the most popular cryptocurrencies. Lightning for example has 2,700+ active nodes, 21,000+ channels, and 560+ bitcoins (~$2M USD) network capacity as of January 2019 according to [8]. Ripple is another large offchain network using its own cryptocurrency called XRP as the main transaction medium. Its network has 200+ enterprise customers including banks and payment providers worldwide. All three offchain networks allow transactions involving multiple payment channels.

2.2 Motivation

We believe a sensible first step of investigating offchain routing is to understand the workloads carried by these newly emerged networks, that is the cryptocurrency transactions. Surprisingly this aspect has received little attention so far compared to other features of offchain networks such as their topologies [29].

We conduct a measurement study of the transactions in the Ripple network, which to our knowledge is the only offchain network whose transaction data are publicly available. We use a dataset from [2] that includes over 2.6 million transactions in Ripple from January 2013 to November 2016. Each data entry includes sender, receiver, volume, and time information. In addition, we crawl Bitcoin's onchain transactions as our second dataset. We believe the characteristics of onchain and offchain transactions are similar as more onchain transactions are moving to offchain networks for faster turnaround and lower cost. We run a full Bitcoin node with



(a) Ripple **(b) Bitcoin**

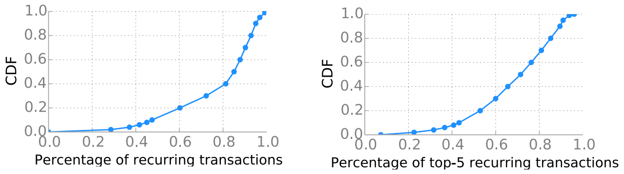
Figure 3: Payment size distributions for Ripple and Bitcoin transactions.

Bitcoin Core[2] to synchronize all blocks and collect all the transactions (over 103 million) from its launch in January 2009 to October 2018. We exclude coinbase transactions which represent new coins mined by the miners.

We now highlight two unique characteristics of cryptocurrency transactions observed from the traces.

Payment sizes are heavy-tailed. We first investigate the payment size distribution. Figure 3 shows the CDF of payment size in Ripple and Bitcoin traces. We observe that most payments are small, while a few large payments contribute the most volume. For Ripple, 10% of the payments are for balances of $1,740 USD or more and they account for 94% of the total volume. The median payment size is only $4.8 USD. For Bitcoin, 10% of payments larger than 8.9×10^7 satoshis constitute 94.7% of the total volume, while the median payment size is only 1.293×10^6 satoshis. This is intuitive to understand since in practice large transactions only happen among a small group of enterprises and financial institutions. For example, the transaction volumes between banks can be substantial, but the trade frequency is relatively low. Most of the time transactions happen between individuals and merchants, such as money transfer between friends and families, and purchases of goods and services. These constitute the vast majority of the use of cryptocurrencies in the same way traditional currencies are used.

The design implication of the heavy-tailed payment size is salient. On one hand, small or *mice payments* are naturally less likely to saturate a payment channel, and tend to be sensitive to the settlement time. The settlement time is the amount of time elapsed from when the payment is placed at the sender to the moment all funds from the payment are received at the receiver. As such, they can be delivered with high probability using just one or a few paths, and the paths do not have to be carefully chosen to minimize the delay. On the other hand, a large payment or *elephant payment* consumes much more funds and using a single path may not be sufficient. Elephant payments are more important to an offchain network as their successful delivery would greatly improve the success volume (i.e., the total size of all successful payments) and ratio (i.e., the percentage of successful payments over all payments). The transaction fee in the offchain network consists of a fixed base fee and a liquidity fee proportional to the payment size. Thus minimizing transaction fees is also meaningful to elephant payments given their significant volume. We thus believe a more delicate and



(a) The CDF of percentage of recurring transactions over all transactions in a 24-hour period across 1,306 days in Ripple.

(b) The CDF of percentage of top-5 most frequent recurring transactions over all recurring transactions in a 24-hour period across 465 days in Ripple.

Figure 4: Analysis of the recurring transactions in the Ripple trace.

optimized routing solution is justified for elephant payments to thoroughly consider all the factors involved. The solution needs to strategically choose a good set of paths with enough capacity, and carefully schedule the elephant payment across the paths with varying fees. The increased settlement time and probing overheads are acceptable given the low frequency of elephant payments.

Our separate treatment of elephant and mice payments is markedly different from prior work that treats all payments equally through the same routing mechanism [24, 28–30]. As we will show, exploiting this characteristic gives us more flexibility to improve success volume and ratio of the network while maintaining the overheads.

Payments are highly recurrent and clustered. We next investigate the relationship between the sender and receiver of the offchain transactions. Due to the lack of this information in the Bitcoin trace, we only analyze the Ripple trace. We examine each of the 1,306 days with recurring transactions the Ripple trace covers, and identify the recurring transactions as those with the same sender-receiver pairs within a 24-hour period.

We observe that the median percentage of recurring transactions among all transactions of the day stands at 86% across 1,306 days as shown in Figure 4a. Thus most of the transactions in Ripple is actually recurring within a 24-hour time frame. Moreover, we find that recurring transactions happen within a small set of users. Among 1,306 days with recurring transactions, 465 days with no fewer than 10 sender-receiver pairs having recurring transactions are chosen. For each of the 465 days, we rank sender-receiver pairs according to their number of recurring transactions and select top-5 with the most frequent recurring payments. Figure 4b shows that the number of recurring transactions from the top-5 sender-receiver pairs accounts for over 70% of the daily transactions. These properties again make intuitive sense since the real-world financial relationship for most people is stable and clustered. One mostly transacts with a small number of parties such as their favorite online merchants and offline businesses (shops, diners, etc.) near work and home, as well as their friends and family.

The design implication of recurring transactions and clustered receivers is also interesting. It allows the use of a routing table to store the paths for the recurring receivers, so the path finding process can be simplified to table lookups especially for mice payments. A small routing table would be enough to cover most recurring transactions due to their clustered nature. This is instrumental towards

---

[2] https://bitcoin.org/en/full-node

reducing the overhead of processing (mice) payments without much performance sacrifice.

To quickly recap, the transaction characteristics presented here enable us to explore a larger design space for offchain routing, and motivate our design of Flash which we now introduce.

3 DESIGN

Flash is a distributed online routing system that processes each transaction as it arrives at the sender, because a centralized offline approach is inherently infeasible for decentralized offchain networks with constantly changing channel balances. Flash differentiates elephant and mice payments and applies different routing algorithms in order to achieve a better performance-overhead tradeoff. For elephants that have a significant impact on overall performance, Flash first adopts a modified max-flow algorithm to find and probe paths with sufficient balance to satisfy their demands, and then solves an optimization program to split the payment over paths to minimize the transaction fees. For mice payments whose demands are easy to satisfy, Flash uses a lightweight solution that simply routes them randomly through a small set of precomputed paths whenever possible in order to reduce the probing overhead.

3.1 Prerequisites

Flash's design relies on two prerequisites about the offchain networks.

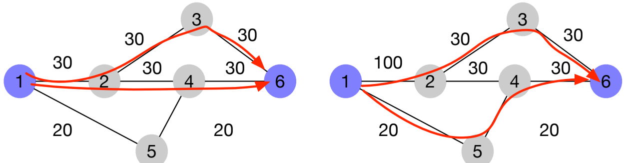
Locally available topology. The topology of an offchain network, without the channel balance information, is fairly stable and changes on an hourly or daily scale. This is because opening or closing a payment channel requires onchain transactions which take at least tens of minutes, and a channel usually remains in the network after establishment. Therefore practical offchain routing protocols in Lightning and Raiden require each node to locally store the topology of the offchain network and periodically update it using some gossiping protocols [5, 7]. Flash assumes similar mechanisms are in place and the connectivity topology is locally available at each node. Note the topology is a directed graph since payment channels are bidirectional: funds can flow in either direction and channel balances on different directions are different.

Atomic multipath payments. To improve the network utilization, Flash uses multipath routing whenever possible and assumes the atomicity of multipath payments is guaranteed, similar to prior work [30]. This can be achieved by mechanisms such as Atomic Multipath Payments (AMP) proposed for Lightning [1]. Building upon HTLC, AMP allows a payment to be split over multiple paths while ensuring the receiver either receives all funds from several partial payments, or gets nothing (i.e. payment fails). The design and implementation of such a mechanism are beyond this paper.

3.2 Routing Elephant Payments

The design challenges for routing elephant payments are: (1) how to find good paths with sufficient capacity to satisfy demand as much as possible, and (2) how to carefully split the payment across the paths in order to minimize the transaction fees.

Path finding with modified max-flow. We discuss some strawman solutions to the first challenge on path finding and why they do not work, and then present Flash's solution with a modified max-flow algorithm.



(a) k simple shortest paths share the same bottleneck path

(b) k edge disjoint path routing is not always effective

Figure 5: An illustrative example of common shortest path schemes. Node 1 is the sender, and node 6 is the receiver. In each scheme two paths are used, i.e. $k = 2$.

Strawman solutions. With the network topology locally available, a first attempt at the path finding problem would be to simply have the sender compute k good paths. Shortest paths, for example, are a natural choice since they minimize the number of hops and helps reduce transaction fees. However, restricting to shortest paths may lead to severe underutilization when they share a common bottleneck. To see this, we consider an example in Figure 5(a). Two simple shortest paths from node 1 to 6 share the same bottleneck link from node 1 to 2. Using them provides a total capacity of 30 while the other path of 1-5-4-6 is underutilized. To overcome this one may consider edge-disjoint shortest paths, which are used in Spider [30]. Yet they may not always work either especially when the common bottleneck has abundant capacity. Figure 5(b) shows that using 2 edge-disjoint shortest paths yields a total capacity of 50, while using 2 simple shortest paths that traverse from 1 to 2 yields a total capacity of 60 since the common link from 1 to 2 has abundant capacity now.

It is thus important to consider channel capacity in path finding for elephant payments. This naturally motivates us to resort to max-flow algorithms [17]. A max-flow algorithm such as Edmonds-Karp [15] is used to find the maximum flow between a pair of nodes in a flow network. However they cannot be directly applied to offchain networks. Max-flow algorithms require a weighted graph, meaning that the balance or capacity of all edges of the graph should be known. This is infeasible in our problem: the channel balance is dynamically changing in offchain networks, and probing each channel of each path whenever an elephant payment arrives does not scale for a network with thousands of nodes and tens of thousands of channels [8].

Flash's solution. We thus develop a modified max-flow algorithm based on Edmonds-Karp [15] to sequentially find k paths and their maximum flow without excessive overheads. Algorithm 1 shows the pseudocode.

Each node has the network topology G without capacity information. When a new elephant arrives the sender s invokes Algorithm 1 to route it. It uses a capacity matrix C to record the probed channel capacity of the paths, and a residual capacity matrix C' to record the remaining capacity of channels as in Edmonds-Karp [15]. Both C and C' are initialized to infinity (lines 4–5 in Algorithm 1). It then enters a loop with at most k iterations to find at most k paths. In

Algorithm 1 Modified Edmonds-Karp for elephant payment routing

```
1:  Input: Topology graph $G$, a payment $(s, t, d)$ from $s$ to $t$ with demand $d$, maximum number of paths needed $k$
2:  Output: Path set $P$, capacity matrix $C$
3:  $P = \emptyset, f = 0$                          ▷ Initialize maximum flow $f$
4:  $C = \infty$                                     ▷ Initialize capacity matrix $C[n \times n]$
5:  $C' = \infty$                                    ▷ Initialize residual capacity matrix $C'[n \times n]$
6:  while $|P| < k$ do                               ▷ Find at most $k$ paths
7:      $p$ = Breadth-First-Search($G, C', s, t$)    ▷ Return a list of nodes on path $p$
8:      if $p == \emptyset$ then
9:          break
10:     Add $p$ to $P$
11:     Probe each channel on $p$ to obtain their capacity $C_p$
12:     Find the bottleneck capacity $c = \min\ C_p$
13:     $f = f + c$
14:     for each edge $(u, v)$ on $p$ do
15:         if $C[u, v] = \infty$ then               ▷ Set channel capacity for the first time
16:             $C[u, v] = C_p[u, v]$
17:             $C'[u, v] = C_p[u, v]$
18:         if $C[v, u] = \infty$ then
19:             $C[v, u] = C_p[v, u]$
20:             $C'[v, u] = C_p[v, u]$
21:         $C'(u, v) = C'(u, v) - c$                ▷ Reduce channel capacity
22:         $C'(v, u) = C'(v, u) + c$                ▷ Increase capacity of the channel in the reverse direction
23: if $f \geq d$ then
24:     return $P, C$                                ▷ Return paths found and capacity
25: else
26:     return $\emptyset$
```

each loop Flash first runs the Breadth-First-Search on topology G with the residual capacity matrix C' to find a feasible shortest path p with non-zero capacity (line 8), and adds p to the solution set P. It then sends probes along p to obtain the capacity of each channel on it, and obtains the bottleneck capacity c. This indicates that we can send c on path p (line 14). It updates the capacity of channels that have been probed for the first time in C according to the probing results C_p. It also updates the residual capacity of channels on path p in the residual capacity matrix C' using c to reflect the new flow found by p. After the loop terminates, Algorithm 1 returns the paths P and capacity matrix C if the maximum flow f over these paths satisfies the payment demand d.

Compared to Edmonds-Karp with $O(|V||E|)$ iterations, our algorithm ends with at most k iterations and k paths to probe when there are at least k paths between s and t on G. This helps reduce the probing overhead. We find that setting k between 20 to 30 provides good performance in practical offchain network topologies with thousands of nodes and tens of thousands of channels. Also our algorithm works without the capacity matrix as input by assuming each channel has non-zero capacity. It is thus possible, though rare in our evaluation, that our algorithm finds a path but its effective capacity is zero after probing. Flash will not send the payment along this path. In the next iteration of path finding, Breath-First-Search function will skip the channel with zero capacity.

Path selection. Given a set of paths with sufficient capacity from Algorithm 1, the next step is to determine how to route over them to minimize the total transaction fees. The fee information is collected during the probing process with the capacity information. We take a principled approach and solve this using mathematical optimization.

Specifically, we have the path set P and the capacity matrix C. We represent the fee collected by a channel (u, v) with a charging function $f_{u,v}$. We assume f is convex. Thus the fee amounts to $f_{u,v}(r_p)$ if we route a partial payment of r_p to (u, v). The objective of the optimization program is to minimize the total fees subject to constraints that the payment demand d is met, and channel capacity is respected:

$$\begin{aligned} \min \quad & \sum_{p\in P}\sum_{\{(u,v)\}} a^p_{u,v} f_{u,v}(r_p) \\ \text{subject to} \quad & \sum_{p\in P} r_p = d, \\ & \sum_{p\in P} r_p a^p_{u,v} - \sum_{p\in P} r_p a^p_{v,u} \leqslant C(u,v), \forall (u,v). \end{aligned} \tag{1}$$

Here $a^p_{u,v}$ indicates whether p uses channel (u, v) or not. Note that partial payments on different direction of the same channel can offset each other in terms of balance. It does not require any sort of synchronization. When the capacity of the channel is changed, the latest state is logged in the local ledger of the channel. Payments can offset the change later.

The optimization program (1) is a convex optimization and can be solved using standard solvers quickly due to the small problem size with k paths. According to the codebase of the Lightning Network Daemon (LND) [5], the fee charging function is typically linear with a fixed fee plus a volume-dependent component, which means (1) is a simple linear program and even easier to solve.

3.3 Routing Mice Payments

The design challenge for mice payment routing is to simplify the protocol and minimize overhead due to their large quantity. Applying elephant routing design here would be an overkill. We now present our design for mice payments which also consists of path finding and path selection.

Path finding. Each node maintains a routing table for mice payments. It contains paths for the unique receivers of this node. Upon seeing a new receiver that does not exist in the routing table, the node computes top-m shortest paths (i.e. using Yen's algorithm [36]) on the local topology G, and adds them to the routing table. If the receiver is in the routing table, Flash simply re-uses the existing paths. Since most payments are recurring as explained in §2.2, this design simplifies path finding into table lookups in most cases without any computation. The recurring nature of mice also ensures the routing table size is not too large. We use top-m shortest paths where m is much less than k the number of paths used for elephant routing in §3.2, because mice payments do not require much capacity, and typically a few shortest paths provide good performance ($m = 4$ in our evaluation).

The routing table is periodically refreshed when the local network topology G is updated (by the underlying gossip protocol):

all entries are re-computed using the latest G. Also when a payment encounters an inaccessible path with zero effective capacity or no connectivity, Flash replaces it with the next top shortest path. Timeouts are used to remove receivers and their entries that have not been accessed for a long time to limit the routing table size.

Path selection. With m shortest paths from the routing table, the sender determines path selection using a trial-and-error loop. It first sends the full payment along a random path p. If successful the protocol ends. Otherwise, the sender probes p to find its effective capacity c_p and sends a partial payment of volume c_p along p. It then updates the remaining demand of the payment and continues the iteration. This ensures low probing overhead since Flash only probes when it is necessary and at most m paths are probed. The use of multiple paths also improves the success ratio of delivering the payment. Instead of following a fixed order (say in descending order of path length), Flash randomly picks the paths to better load balance them without knowing their instantaneous capacities. Lastly, when m paths are exhausted and demand is not satisfied, Flash declares the payment fails.

4 SIMULATION

In this section, we evaluate the performance of Flash against existing offchain routing algorithms using simulation. Our evaluation aims to answer the following questions:

- How does Flash perform under realistic offchain network topologies and traces?
- How do channel capacity and network load affect Flash's performance?
- How effective is differentiating elephant and mice payments in Flash?
- How effective is the mice payment routing algorithm?

4.1 Methodology

Setup. We implement offchain network topologies and routing schemes using the NetworkX package in Python [6] in the simulation. Our simulation focuses on the routing performance in a large-scale real offchain network, and does not concern the implementation of the underlying security mechanism (say HTLC).

We evaluate Flash with two real-world offchain network topologies: Ripple and Lightning. We obtain crawls of Ripple's active nodes and channels from January 2013 to November 2016 from [2]. This topology includes 93,502 nodes and 331,096 edges. We remove nodes with only a single neighbor and channels with no funds from the topology. The processed topology we use in the simulation includes 1,870 nodes and 17,416 edges. The distribution of funds on payment channels in Ripple is extremely skewed. All schemes perform poorly in this scenario. In order to bootstrap the network with a more balanced distribution, we redistribute the funds by evenly assigning the total funds over both directions of a channel. For Lightning topology we run the `c-lightning` [4] node on `mainnet` and connect it to an existing node by opening a channel with the node. We use commands `listchannels` and `listnodes` to get information of nodes and channels as a snapshot of the Lightning network on a particular day of December 2018. The number of nodes is 2,511 and the number of channels is 36,016. Since the lightning network preserves the privacy of channel balances, we are only able to get a bound on the balance of the channel rather than the exact balance distribution. We thus evenly assign funds over both directions of a channel, the same as what we do for Ripple.

We generate payments by randomly sampling the Ripple trace for the Ripple topology. Due to the lack of sender-receiver information in the Bitcoin trace for Lightning, we randomly sample the Bitcoin trace for transaction volumes, and sample a sender- receiver pair from the Ripple trace and map it to nodes in the Lightning topology. Payments arrive at senders sequentially.

Benchmarks. We compare four offchain routing algorithms.

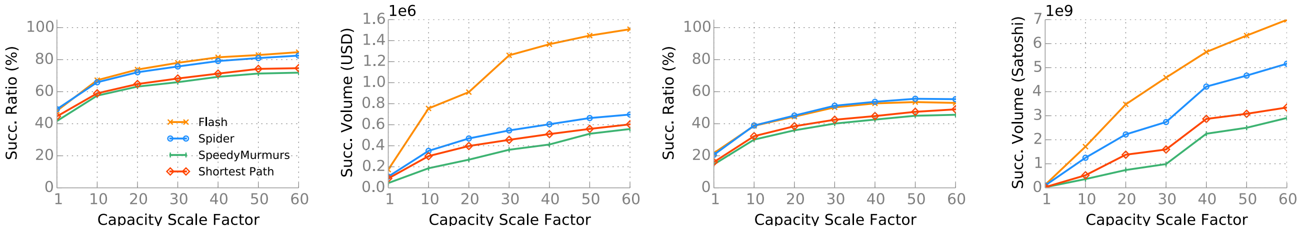
- *Flash*: Our routing algorithms. Unless stated otherwise, we set the number of shortest paths for each receiver in mice payment routing to 4, i.e. $m = 4$, and the number of paths for elephant routing to 20, i.e. $k = 20$. The elephant-mice threshold is set such that 90% of payments are mice. In §4.3, we show how different thresholds affect the performance.
- *Spider* [30]: The state-of-the-art offchain routing algorithm which considers the dynamics of channel balance. It balances paths by using those with maximum available capacity, following a "waterfilling" heuristic. It uses 4 edge-disjoint paths for each payment.
- *SpeedyMurmurs* [29]: An embedding-based routing algorithm that relies on assigning coordinates to nodes to find short paths with reduced overhead. The number of landmarks is 3 as [29] suggests.
- *Shortest Path (SP)*: This is the baseline. SP uses the path with the fewest hops between the sender and receiver to route a payment.

Metrics. Similar to prior work [29, 30], we use success ratio, success volume and number of probing messages as the primary metrics in the simulation. The success ratio is defined as the percentage of successful payments whose demands are met over all generated payments. The success volume describes the total size of all successful payments. Before sending payments, the sender probes the effective capacity of paths to the receiver (one probe message per path). The number of probe messages describes the probing overhead. We report the average results over 5 runs.

4.2 Overall Performance and Overhead

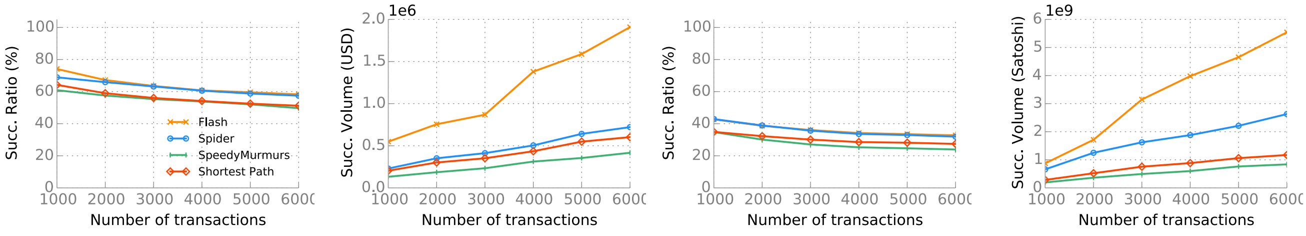
We now examine the performance and overhead of Flash with different settings of the offchain network.

Performance with different capacities. We first evaluate the performance of Flash with various link capacities. The medium channel capacity in Lightning is around 500,000 Satoshi and in Ripple is 250 USD. As offchain networks are still in their infancy and the capacity provided may be limited, we scale the link capacity by a factor of 1 to 60 in the simulation similar to existing work [29, 30]. The number of transactions used is fixed at 2000. Figure 6 shows the success ratio and volume results. For both Ripple and Lightning, Flash performs ~20% better than SpeedyMurmurs and Shortest Path on success ratio. Flash and Spider are both able to fulfill most mice payments. As the success ratio is dominated by mice payments, Flash and Spider achieve similar performance. For



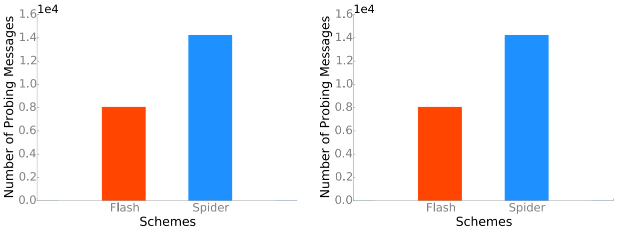
(a) Succ. ratio in Ripple **(b) Succ. volume in Ripple** **(c) Succ. ratio in Lightning** **(d) Succ. volume in Lightning**

Figure 6: Performance results with varying link capacities in Ripple and Lightning.



(a) Succ. ratio in Ripple **(b) Succ. volume in Ripple** **(c) Succ. ratio in Lightning** **(d) Succ. volume in Lightning**

Figure 7: Performance results with varying number of transactions in Ripple and Lightning.



(a) Ripple **(b) Lightning**

Figure 8: Probing message comparison results.

success volume, Flash performs up to 4.5x better over Shortest Path, 5x better than SpeedyMurmurs, and 2.3x better than Spider. The success volume benefits of Flash are due to its delicate elephant payment routing that uses more capacity and carefully schedules the partial payments to deliver them successfully. As the network capacity increases, we observe more successful payments and thus the increase of both success ratio and volume. Flash consistently outperforms other schemes.

Performance with varying transaction numbers. We also vary the number of transactions flowing into the network to emulate different loads. The capacity scale factor is 10. With the increase of the number of transactions, the success ratio of all schemes degrades as shown in Figure 7. One possible reason is that, as more payments, especially elephant payments are accepted, channels are easier to be saturated in one direction. Although the number of successful payments keeps increasing, the probability to fulfill a payment decreases. Observe that Flash consistently outperforms other schemes. It shows significant benefits on success volume: the performance gains over Shortest Path, SpeedyMurmurs, and Spider are up to 4.7x, 6.6x, and 2.6x, respectively. We also observe that Flash's performance gains increase with more transactions, suggesting that it scales better than other solutions.

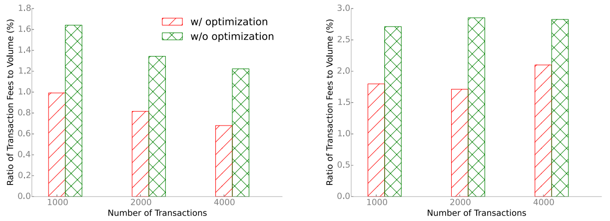
Probing message overhead. We have demonstrated the performance improvement of Flash in terms of success ratio and volume. We now evaluate the number of probing messages of Flash to see if our algorithms can curb the overhead of routing. Figure 8 shows the comparison results with 2000 transactions and a capacity scale factor of 10. Note that SpeedyMurmurs and Shortest Path are static routing schemes without probing. Without probing they suffer from poor performance as discussed just now. We thus exclude them from the comparison here. The number of probing messages along a path is proportional to the number of hops of the path.

Observing from Figure 8, compared to Spider which also uses multiple paths, Flash saves 43% message overhead in Ripple and 37% in Lightning. Spider treats mice and elephant flows the same and always uses 4 shortest paths. Flash differentiates mice and elephants: though it uses many more paths (20) for elephants, it uses at most 4 paths for the vast majority of the mice payments in order to balance the performance-overhead tradeoff. Moreover, Flash's mice payment routing relies on a trial-and-error approach to further reduce probing overhead: it only probes a path when it cannot deliver the payment in full, which usually does not happen for mice payments. We observe that most mice payments are delivered with 1 or 2 paths. Thus the results here demonstrate that Flash indeed achieves a better tradeoff between performance and overhead compared to state of the art.

4.3 Flash Microbenchmarks

We now take a deep dive into Flash by evaluating microbenchmarks about the impact of some key parameters to its design. Through the microbenchmarks, we also verify our design choices. In all experiments here we use 2000 transactions in each run and a capacity scale factor of 10 unless stated otherwise.

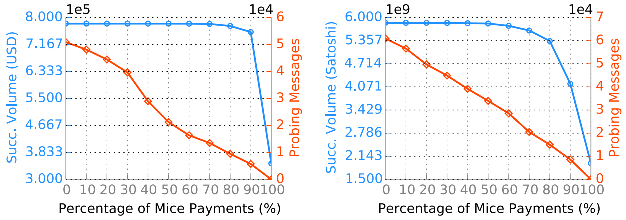
Impact of transaction fee optimization. As mentioned in §3.2, Flash splits an elephant payment over multiple paths to minimize



(a) Lightning

(b) Ripple

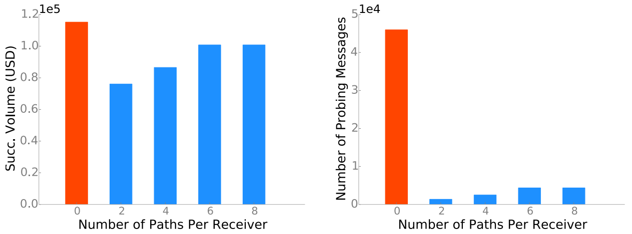
Figure 9: Impact of transaction fee optimization in Flash.



(a) Ripple

(b) Lightning

Figure 10: Impact of threshold value in Flash.



(a) Success volume of mice payments

(b) Probing overhead of mice payments

Figure 11: Impact of number of paths per receiver m for mice payment routing in Flash. Here Flash routes mice payments in the same way as elephant payments when $m = 0$.

the total transaction fees. We now validate the effectiveness of this design. To perform a fair comparison, we realize Flash without transaction fees minimization as the baseline, where the paths are used sequentially as they are found by our modified Edmonds-Karp algorithm until the demand is met. We compare the unit transaction fees (in percentage) to avoid the impact of volume on the result. Note the unit fee is obtained over all payments, not just elephant payments. We set 90% channels with random fees from 0.1% to 1% of the volume and 10% channels from 1% to 10%. Observe from Figure 9 that Flash reduces the transaction fees by around 40% on average in both Ripple and Lighting compared to not performing fee minimization.

Impact of threshold. We first show how the choice of threshold impacts the performance, i.e. success volume of payments. Here we vary the threshold value such that the percentage of mice payments varies from 0% to 100%. Obviously a higher percentage with a larger threshold results in more payments classified as mice. Observe from Figure 10 that the success volume of mice payments remains stable until the percentage of mice reaches 80%–90%. That is, when most payments are treated as mice with Flash's simple routing algorithm, their success volume is only marginally smaller than when everyone is treated by the elephant routing algorithm. However, the probing overhead increases as the percentage of mice payments decreases and probing is more aggressively used. This clearly demonstrates that our design choice of differentiating mice and elephant is effective: it significantly reduces the probing overhead without much performance degradation for most mice payments. This also justifies our setting of threshold with 90% mice flows which achieves a good performance-overhead tradeoff.

Impact of number of paths per receiver for mice routing. We now investigate the benefit of using just a few shortest paths per receiver in routing mice payments. We only show results with the Ripple trace for brevity since results with Lightning trace shows similar trends. Here we vary m, the number of paths for a receiver in the routing table for mice payments. The case with $m = 0$ represents the performance upper bound when we route mice payments in the same way as elephant payments in §3.2, which clearly offers the best performance in success volume. Figure 11(a) shows that just a few paths per receiver leads to fairly good performance compared to routing them as elephants: the gap is within 15% with $m = 6$. The performance of Flash stabilizes when m exceeds 6. Figure 11(b) shows that using a few routes achieves at least ∼12x less probing overhead. These results confirm that our mice payment routing design is effective in reducing probing overhead while ensuring satisfactory performance.

5 TESTBED EVALUATION

We conduct testbed evaluation to further investigate Flash's design.

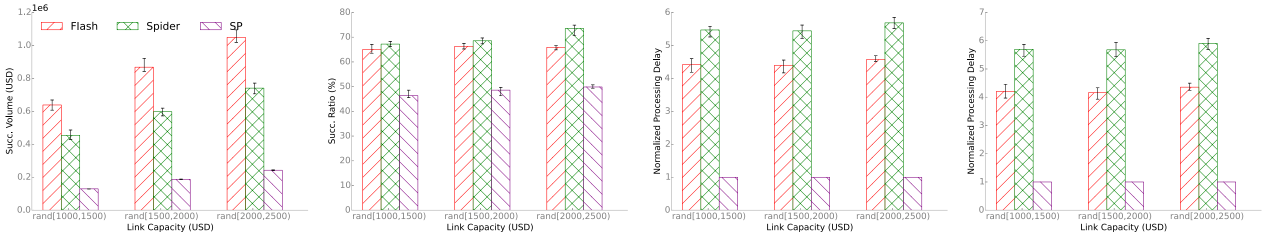
5.1 Implementation

We start by describing our prototype implementation.

Overview. Since we focus on routing, we take a minimalist approach and build a simplified prototype offchain routing system without mechanisms such as gossiping protocols for topology maintenance and HTLC for security. We use Golang to implement the prototype with TCP for network communication. The prototype reads the network topology from a local file at launch time. Upon seeing a new transaction, it runs the routing algorithm and sends it out accordingly.

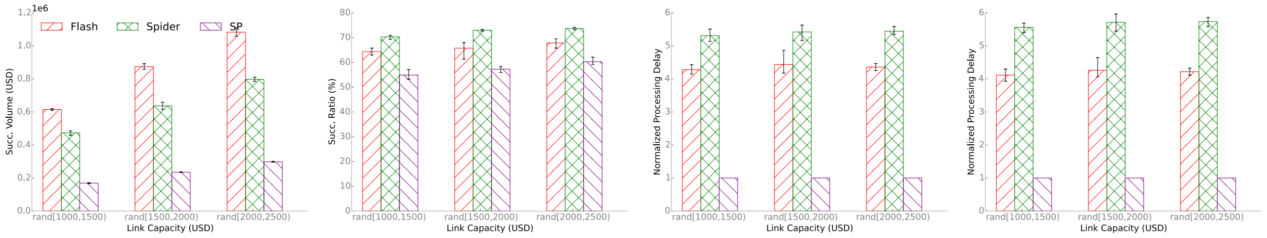
Most importantly, we implement an offchain routing protocol in our prototype that realizes three essential functions required by any routing algorithm: source routing, probing, and atomic payment processing. We describe their details in the following.

Source routing is the basic service of offchain networks since the probing process and payment routing happen over a specified path of multiple hops in the overlay network. We implement a simple source routing scheme by embedding the complete path into every message a sender initiates. Table 1 shows the message format used in our prototype, where the `Path` field contains the path information. Upon receiving a message, a node parses this field and sends it to the next-hop after necessary processing as indicated in the `Type` field.



(a) Success volume **(b) Success ratio** **(c) Overall processing delay** **(d) Mice processing delay**

Figure 12: Testbed experiment results of the 50-node network.



(a) Success volume **(b) Success ratio** **(c) Overall processing delay** **(d) Mice processing delay**

Figure 13: Testbed experiment results of the 100-node network.

Field	Description
`TransID`	A unique ID of a (partial) payment
`Type`	Message type
`Path`	Path of this message
`Capacity`	Probed channel capacity
`Commit`	Committed amount of funds for this payment

Table 1: Message format for source routing in our prototype.

Probing is needed for offchain routing algorithms to collect the ever-changing channel balance. A node initiates probing by constructing a `PROBE` message for the path it is interested in. The intermediate nodes append the `Capacity` field in the message with their current balance. To return the probed information, the receiver modifies the message type to `PROBE_ACK`, replaces the `Path` field with the reversed version of the forward path, and sends it back all the way to the sender.

Atomic payment processing. Last but not least, we implement a two-phase commit protocol to realize atomic payment processing without complex security mechanisms like HTLC [27]. This is necessary for two reasons. First, due to network dynamics, it is possible that a payment fails on its path because the balance of some channel has changed after it was last probed by the sender. Thus confirmation is required for the sender to know the status of the payment and ensure the atomicity of balance update on the path. Second and more interestingly, with multipath routing, a payment is successful if and only if all its sub-payments are successful [10, 11]. This necessitates the need for two-phase commit from distributed systems, where the protocol only commits the payment when all its sub-payments have been confirmed on their paths.

Our two-phase commit protocol works as follows for the general case of multipath routing. In the first phase, the sender prepares a `COMMIT` message for each sub-payment and sends them out. An intermediate node determines if its current balance can handle this sub-payment. If yes, it decreases its balance by the volume specified in the `COMMIT` message and forwards the message to the next hop. The receiver constructs a `COMMIT_ACK` message by adding the success information in the payload and reversing the path. The sender recognizes this sub-payment to be successful upon receiving the `COMMIT_ACK`. In case an intermediate node does not have enough balance, it constructs a `COMMIT_NACK` with the reversed path and immediately sends it back to its previous hop. The sender recognizes the sub-payment to be failed afterward.

After the results of all sub-payments are back, the protocol enters the second phase. When all sub-payments are successful, the sender sends a `CONFIRM` message for each sub-payment along their paths. The intermediate nodes simply relay the `CONFIRM` message. The receiver would send a `CONFIRM_ACK` along the reverse path back to the sender. Now each intermediate node processes `CONFIRM_ACK` by adding the committed funds of this sub-payment to the channel in the reverse direction, in order to make the bidirectional channel balances consistent. With all `CONFIRM_ACK` received, the sender considers this payment successful. In case at least one sub-payment is unsuccessful in the first commit phase, the sender sends a `REVERSE` message for each sub-payment. All intermediate nodes then add back the committed funds to the channel in the forward path, and the receiver sends a corresponding `REVERSE_ACK` to indicate that everyone has been informed.

5.2 Experiment Setup

Our evaluation is conducted on a server machine with a 10-core Intel E5-2640v4 CPU and 64GB DDR4-2400 memory. For simplicity, we represent each node of an offchain network as a single process running our Golang prototype. Each process is bound to a unique IP address and port number tuple.

We implement our routing algorithms described in §3 in our prototype. We also implement two baseline routing algorithms: Spider as in [30] and a simple shortest path scheme (denoted as SP) as described in §4.1.

The network topology follows the Watts Strogatz graph [34]. The Watts Strogatz graph exhibits short path lengths and high clustering coefficients which is a good fit to generate topologies for offchain networks. This graph was also used for evaluation in previous work on offchain routing like Flare [28]. We generate two topologies with 50 and 100 nodes, respectively. The capacity of each channel is set randomly from an interval which varies from [\$1000, \$1500), [\$1500, \$2000), to [\$2000, \$2500). We generate 10,000 transactions whose volume follows the Ripple trace and randomly select the sender-receiver pairs.[3] For Flash the payment size threshold is set such that 90% of transactions are mice, the number of paths for elephant routing k is 20, and the number of shortest paths for mice routing m is 4. Spider uses 4 edge-disjoint shortest paths as proposed in [30]. Each scheme is evaluated in 5 independent runs. Results are shown with min-mean-max bars.

5.3 Results

With experiments under different network scales (50-node and 100-node), we demonstrate the consistent performance gains from Flash. We can observe from Figures 12a and 13a that the success volume of Flash is much larger than Spider, 42.5% and 34.4% on average for the 50-node and 100-node topologies, respectively. This demonstrates the effectiveness of our routing algorithms which select a good set of paths to improve throughput. As shown in Figures 12b and 13b, Flash's success ratio is slightly worse (5.6% and 8.8% on average, respectively) than Spider and is better (36.3% and 14.8% on average, respectively) than SP. The reason Flash has lower success ratio than Spider is that Spider uses waterfilling to balance the utilization of multiple paths and creates better chances for mice payments to go through. Flash does not consider load balance for mice payments in the design in order to achieve low probing overhead.

Next, we investigate overhead. Instead of messaging overhead, we measure the average processing delay of a transaction in our prototype as the metric of overhead. We normalize the results by the average processing delay of SP, the simplest baseline algorithm. From Figures 12c and 13c, we can see that Flash's processing delay is on average 19.4% and 19.2% smaller than Spider for the 50-node and 100-node topologies, respectively. Further, we look at the processing delay of mice payments that generally require faster settlement time. As plotted in Figures 12d and 13d, Flash is on average 26.4% and 26% faster than Spider in the two topologies, respectively. This confirms that our mice payment routing algorithm reduces the probing overhead and thus the processing delay significantly.

6 RELATED WORK

Offchain routing emerges only recently in 2016. The first offchain routing algorithm is proposed in the design draft of Lightning network [27]. It routes payments to paths using a BGP-like system and maintains a global routing table. To minimize the routing state, Flare [28] proposes that nodes only maintain neighbors within a certain hop distance. When routing a payment, the sender exchanges the neighbor information with the receiver to construct complete paths. Besides, each node finds some random beacon nodes to supplement its view of the network.

To further reduce the message overhead in path finding, SilentWhispers [24] utilizes landmark-centered routing. It performs periodic Breadth-First-Search to find the shortest path from the landmarks to the sender and receiver. All paths need to go through the landmarks, which makes some paths unnecessarily long. SpeedyMurmurs [29] proposes embedding-based routing to assign coordinates to nodes and find shortcuts that reduce the average path lengths.

The above routing algorithms fall into static routing, which does not consider payment channel dynamics and leads to poor throughput performance. Revive [22] and Spider [30] take the dynamic channel balances into consideration and propose centralized offline routing algorithms to maximize the throughput or success volume of payments. As we discussed centralized schemes have high probing overhead and do not work for decentralized offchain networks.

Compared to existing work, Flash is the first solution that considers the characteristics of payments in offchain network in order to achieve a better balance between the path optimality and probing overhead. Flash's approach of differentiating elephant and mice payments are akin to past work on flow scheduling in datacenter networks (DCNs), such as Hedera [13] and DevoFlow [16]. Other effective approaches for DCNs, such as congestion aware load balancing [14, 33] and fine-grained routing [19, 26, 31], may also provide insights for offchain routing solutions. The key differences are that, an offchain network topology is highly irregular while a DCN topology is usually a Clos, and the channel balance is highly fluctuating while the link capacity is fixed and abundant in a DCN. We believe how to learn from these proven ideas in DCN for better offchain routing designs would be an interesting direction of future work with much potential.

7 CONCLUSION

We presented Flash, a new routing solution that efficiently delivers payments over offchain networks. By studying the characteristics of payments in real offchain networks, we find that payment sizes are heavy-tailed, and most payments are recurring. Flash thus differentiates the treatment of elephant and mice payments. It uses a modified max-flow algorithm to provide elephant payments with sufficient path capacity, and routes mice payments by a routing table with just a few shortest paths to reduce probing overhead. Through trace-driven simulations and prototype implementation, we demonstrated that Flash significantly outperforms existing solutions especially on success volume, while maintaining low probing overhead.

ACKNOWLEDGMENTS

We thank our shepherd Matteo Varvello and the anonymous reviewers for their comments that improved the paper. The work is supported in part by the Hong Kong RGC GRF (CityU 11210818).

---

[3] We ensure there exists at least one path from sender to receiver.

REFERENCES

[1] 2018. Atomic Multi-path Payment. https://lists.linuxfoundation.org/pipermail/lightning-dev/2018-February/000993.html.
[2] 2018. Ripple transaction trace. https://crysp.uwaterloo.ca/software/speedymurmurs/.
[3] 2019. Bitcoin. https://bitcoin.org/en/.
[4] 2019. c-lightning Daemon. https://github.com/ElementsProject/lightning/tree/master/lightningd.
[5] 2019. Lightning Network Daemon. https://github.com/lightningnetwork/lnd.
[6] 2019. NetworkX. https://networkx.github.io/.
[7] 2019. Raiden Network Daemon. https://github.com/raiden-network/raiden.
[8] 2019. Real-Time Lightning Network Statistics. https://1ml.com/statistics.
[9] 2019. Ripple. https://ripple.com/.
[10] 2019. The Lightning Network. https://lightning.network/.
[11] 2019. The Raiden Network. https://raiden.network/.
[12] 2019. Transaction Rate of Bitcoin. https://www.blockchain.com/en/charts/transactions-per-second.
[13] Mohammad Al-Fares, Sivasankar Radhakrishnan, Barath Raghavan, Nelson Huang, and Amin Vahdat. 2010. Hedera: Dynamic Flow Scheduling for Data Center Networks. In *Proc. USENIX NSDI*.
[14] Mohammad Alizadeh, Tom Edsall, Sarang Dharmapurikar, Ramanan Vaidyanathan, Kevin Chu, Andy Fingerhut, Vinh The Lam, Francis Matus, Rong Pan, Navindra Yadav, and George Varghese. 2014. CONGA: Distributed Congestion-Aware Load Balancing for Datacenters. In *Proc. ACM SIGCOMM*.
[15] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. 2009. *Introduction to Algorithms*. MIT Press.
[16] Andrew R. Curtis, Jeffrey C. Mogul, Jean Tourrilhes, Praveen Yalagandula, Puneet Sharma, and Sujata Banerjee. 2011. DevoFlow: Scaling Flow Management for High-performance Networks. In *Proc. ACM SIGCOMM*.
[17] Lester Randolph Ford and Delbert R Fulkerson. 1956. Maximal flow through a network. *Canadian Journal of Mathematics* 8 (1956), 399–404.
[18] Yossi Gilad, Rotem Hemo, Silvio Micali, Georgios Vlachos, and Nickolai Zeldovich. 2017. Algorand: Scaling byzantine agreements for cryptocurrencies. In *Proc. ACM SOSP*.
[19] Keqiang He, Eric Rozner, Kanak Agarwal, Wes Felter, John Carter, and Aditya Akella. 2015. Presto: Edge-based Load Balancing for Fast Datacenter Networks. In *Proc. ACM SIGCOMM*.
[20] Chi-Yao Hong, Srikanth Kandula, Ratul Mahajan, Ming Zhang, Vijay Gill, Mohan Nanduri, and Roger Wattenhofer. 2013. Achieving High Utilization with Software-Driven WAN. In *Proc. ACM SIGCOMM*.
[21] Sushant Jain, Alok Kumar, Subhasree Mandal, Joon Ong, Leon Poutievski, Arjun Singh, Subbaiah Venkata, Jim Wanderer, Junlan Zhou, Min Zhu, Jon Zolla, Urs Hölzle, Stephen Stuart, and Amin Vahdat. 2013. B4: Experience with a Globally-Deployed Software Defined WAN. In *Proc. ACM SIGCOMM*.
[22] Rami Khalil and Arthur Gervais. 2017. Revive: Rebalancing off-blockchain payment networks. In *Proc. ACM CCS*.
[23] Loi Luu, Viswesh Narayanan, Chaodong Zheng, Kunal Baweja, Seth Gilbert, and Prateek Saxena. 2016. A secure sharding protocol for open blockchains. In *Proc. ACM CCS*.
[24] Pedro Moreno-Sanchez, Aniket Kate, and Matteo Maffei. 2017. SilentWhispers: Enforcing Security and Privacy in Decentralized Credit Networks. In *Proc. NDSS*.
[25] Satoshi Nakamoto. 2008. Bitcoin: A peer-to-peer electronic cash system. *Technical Report* (2008). https://bitcoin.org/bitcoin.pdf
[26] J. Perry, H. Balakrishnan, and D. Shah. 2017. Flowtune: Flowlet Control for Datacenter Networks. In *Proc. USENIX NSDI*.
[27] Joseph Poon and Thaddeus Dryja. 2016. The bitcoin lightning network: Scalable off-chain instant payments. *Technical Report* (2016). https://lightning.network/lightning-network-paper.pdf
[28] Pavel Prihodko, Slava Zhigulin, Mykola Sahno, Aleksei Ostrovskiy, and Olaoluwa Osuntokun. 2016. Flare: An approach to routing in lightning network. *White Paper* (2016). https://bitfury.com/content/downloads/whitepaper_flare_an_approach_to_routing_in_lightning_network_7_7_2016.pdf
[29] Stefanie Roos, Pedro Moreno-Sanchez, Aniket Kate, and Ian Goldberg. 2018. Settling Payments Fast and Private: Efficient Decentralized Routing for Path-Based Transactions. In *Proc. NDSS*.
[30] Vibhaalakshmi Sivaraman, Shaileshh Bojja Venkatakrishnan, Mohammad Alizadeh, Giulia Fanti, and Pramod Viswanath. 2018. Routing Cryptocurrency with the Spider Network. In *Proc. ACM HotNets*.
[31] Erico Vanini, Rong Pan, Mohammad Alizadeh, Parvin Taheri, and Tom Edsall. 2017. Let It Flow: Resilient Asymmetric Load Balancing with Flowlet Switching. In *Proc. USENIX NSDI*.
[32] Jiaping Wang and Hao Wang. 2019. Monoxide: Scale Out Blockchain with Asynchronized Consensus Zones. In *Proc. USENIX NSDI*.
[33] Peng Wang, Hong Xu, Zhixiong Niu, Dongsu Han, and Yongqiang Xiong. 2016. Expeditus: Congestion-aware Load Balancing in Clos Data Center Networks. In *Proc. ACM SoCC*.
[34] Duncan J Watts and Steven H Strogatz. 1998. Collective dynamics of 'small-world'networks. *Nature* 393, 6684 (1998), 440.
[35] Gavin Wood. 2014. Ethereum: a secure decentralized transaction ledger. http://gavwood.com/paper.pdf.
[36] Jin Y Yen. 1971. Finding the k shortest loopless paths in a network. *Management Science* 17, 11 (1971), 712–716.
[37] Mahdi Zamani, Mahnush Movahedi, and Mariana Raykova. 2018. RapidChain: scaling blockchain via full sharding. In *Proc. ACM CCS*.