Deep Reinforcement Learning-based Rebalancing Policies for Profit Maximization of Relay Nodes in Payment Channel Networks

Anonymous authors

Department
Institute
email

Abstract. Payment channel networks (PCNs) are a layer-2 blockchain scalability solution, with its main entity, the payment channel, enabling transactions between pairs of nodes "off-chain," thus reducing the burden on the layer-1 network. Nodes with multiple channels can serve as relays for multihop payments by providing their liquidity and withholding part of the payment amount as a fee. Relay nodes might after a while end up with one or more unbalanced channels, and thus need to trigger a rebalancing operation. In this paper, we study how a relay node can maximize its profits from fees by using the rebalancing method of submarine swaps. We introduce a stochastic model to capture the dynamics of a relay node observing random transaction arrivals and performing occasional rebalancing operations, and express the system evolution as a Markov Decision Process. We formulate the problem of the maximization of the node's fortune over time over all rebalancing policies, and approximate the optimal solution by designing a Deep Reinforcement Learning (DRL)-based rebalancing policy. We build a discrete event simulator of the system and use it to demonstrate the DRL policy's superior performance under most conditions by conducting a comparative study of different policies and parameterizations. Our work is the first to introduce DRL for liquidity management in the complex world of PCNs.

Keywords: Payment Channel Networks · Lightning Network · rebalancing · submarine swaps · Deep Reinforcement Learning · Soft Actor-Critic · optimization · discrete event simulation · control.

1 Introduction

Blockchain technology enables trusted interactions between untrusted parties, with financial applications like Bitcoin and beyond, but with also known scalability issues [8]. Payment channels are a layer-2 development towards avoiding the long confirmation times and high costs of the layer-1 network: they enable nodes that want to transact quickly, cheaply and privately to do by depositing some balances to open a payment channel between themselves, and then trustlessly shifting the same total balance between the two sides without broadcasting

their transactions and burdening the network. Connected channels create a Payment Channel Network (PCN), via which two nodes not sharing a channel can still pay one another via a sequence of existing channels. Intermediate nodes in the PCN function as relays: they forward the payment along its path and collect relay fees in return. As transactions flow through the PCN, some channels get depleted, causing incoming transactions to fail because of insufficient liquidity on their path. Thus, the need for channel rebalancing arises.

In this paper, we study the rebalancing mechanism of submarine swaps, which allows a blockchain node to exchange funds from on- to off-chain and vice versa. Since a swap involves an on-chain transaction, it takes some time to complete. Taking this into account, we formulate the following optimal rebalancing problem as a Markov Decision Process (MDP): For a node relaying traffic across multiple channels, determine an optimal rebalancing strategy over time (i.e. when and how much to rebalance) as a function of the transaction arrival rates observed from an unknown distribution and the confirmation time of an on-chain transaction, so that the node can keep its channels liquid and its profit from relay fees can be maximized.

More specifically, our *contributions* are the following:

- We develop a stochastic model that captures the dynamics of a relay node with two payment channels under two timescales: a continuous one for random discrete transaction arrivals in both directions from distributions unknown to the node, and a discrete one for dispatching rebalancing operations.
- We express the system evolution in our model as an MDP with continuous state and action spaces and time-dependent constraints on the actions, and formulate the problem of relay node profit maximization.
- We approximate the optimal policy of the MDP using Deep Reinforcement Learning (DRL) by appropriately engineering the states, actions and rewards and tuning a version of the Soft Actor-Critic algorithm.
- We develop a discrete event simulator of the system, and use it to evaluate the performance of the learning-based as well as other heuristic rebalancing policies under various transaction arrival conditions and demonstrate the superiority of our policy in a range of regimes.

In summary, our paper is the first to formally study the submarine swap rebalancing mechanism and to introduce a DRL-based method for channel rebalancing in particular, and for PCN liquidity management in general.

2 Background

2.1 Payment Channel Networks and the need for rebalancing

A payment channel (Fig. 1) is created between two nodes N_1 and N_2 after they deposit some capital to a channel-opening on-chain transaction. After this transaction is confirmed, the nodes can transact completely *off-chain* (i.e. in the channel) without broadcasting their interactions to the layer-1 network, and

without the risk of losing funds, thanks to a cryptographic safety mechanism. The sum of their two balances in the channel remains constant and is called the channel capacity. A transaction of amount α from N_1 to N_2 will succeed if the balance of N_1 at that moment suffices to cover it. In this case, the balance of N_1 is reduced by α and the balance of N_2 is increased by α.

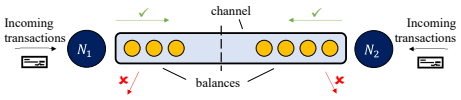


Fig. 1. A payment channel between nodes N_1 and N_2 and current balances of 3 and 4

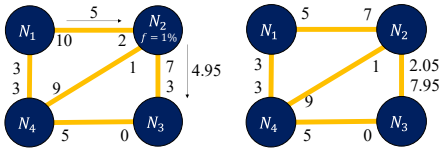


Fig. 2. Processing of a transaction in a payment channel network: before (left) and after (right)

As pairs of nodes create channels, a payment channel network (Fig. 2) is formed, over which multihop payments are possible: Consider a transaction of amount 5 from N_1 to N_3 via N_2. Note that the amount 5 includes the fees that will have be paid on the way, e.g. 1% at each intermediate node. In the N_1N_2 channel, N_1's local balance is reduced by 5 and N_2's local balance is increased by 5. In the N_2N_3 channel, N_2's local balance is reduced by $5-\text{fees} = 4.95$ and N_3's local balance is increased by 4.95. N_2's total capital in all its channels before the transaction was $2+1+7=10$, while after it is $7+1+2.05=10.05$, so N_2 made a profit of 0.05 by acting as a relay. If one of the outgoing balances did not suffice, then the transaction would fail end-to-end, thanks to a smart contract mechanism, the Hashed Time-Lock Contract (HTLC). The role of relay nodes is fundamental for the continuous operation of a PCN. The most prominent PCN currently is the Lightning Network [26] built on top of Bitcoin. More details on PCN operation can be found in [23].

Depending on the demand a payment channel is facing in its two directions, funds might accumulate on one side and deplete on the other, due to a combination of factors (see Appendix A for details). The resulting imbalance is undesirable, as it leads to transaction failures and loss of profit from relay fees, thus creating the need for rebalancing mechanisms.

2.2 The submarine swap rebalancing mechanism

In this work, we study submarine swaps, introduced in [4] and used commercially by Boltz[1] and Loop[2]. At a high level, a *submarine swap* works as follows (Fig. 3): Node N_1 owns some funds in its channel with node N_2, and some funds on-chain. At time t_0, the channel N_1N_2 is almost depleted on N_1's side (balance = 5). N_1 can start a *swap-in* by paying an amount (50) to a Liquidity Service Provider (LSP) – a wealthy node with access to both layers – via an *on-chain* transaction, and the LSP will give this amount back (reduced by a 10% swap

[1] https://boltz.exchange
[2] https://lightning.engineering/loop

fee, so 45) to N_1 *off-chain* via a path that goes through N_2. The final amount that is added at N_1 (and subtracted at N_2) is $45 - \varepsilon$ due to the relay fees spent on its way from the LSP. Thus, at time t_1 the channel will be almost perfectly balanced. The reverse procedure is also possible (a *reverse submarine swap* or *swap-out*) in order for a node to offload funds from its channel, by paying the server off-chain and receiving funds on-chain. More details on the submarine swap technical protocol can be found in Appendix B.

The node has to make an important tradeoff: not rebalance a lot to avoid paying swap fees but forfeit relay fees of transactions dropped due to imbalance, or vice versa. This motivates the problem of demand-aware, timely dispatching of swaps by a node aiming to maximize its total fortune.

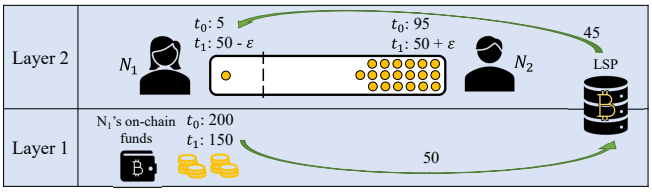


Fig. 3. A submarine swap (swap-in)

3 Problem formulation

3.1 System evolution

In this section, we introduce a stochastic model of a PCN relay node N that has two channels, one with node L and one with node R, and wishes to maximize its profits from relaying payments from L to R and vice versa (Fig. 4). Let $b_{LN}(\tau), b_{NL}(\tau), b_{NR}(\tau), b_{RN}(\tau)$ be the channel balances and $B_N(\tau)$ be the on-chain amount of N at time τ. Let C_n be the total capacity of the channel Nn, $n \in \mathcal{N} \triangleq \{L, R\}$. Events happen at two timescales: a continuous one for arriving transactions, and a discrete one for times when the node is allowed to rebalance.

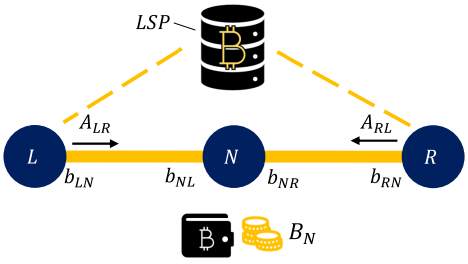


Fig. 4. System model

The transaction timescale Transactions arrive as a marked point process and are characterized by their direction (L-to-R or R-to-L), time of arrival and amount. We consider node N to not be the source or destination of any transactions itself, but rather to only act as a relay. At each moment in continuous time (denoted by τ), (at most) one transaction arrives in the system. All transactions are admitted, but some fail due to insufficient balances.

Let $f(\alpha)$ be the fee that a transaction of amount α pays to a node that relays it. We assume all nodes charge the same fees. f can be any fixed function with $f(0) = 0$. In practice, $f(\alpha) = f_{\text{base}} + f_{\text{prop}} \cdot \alpha$, where the base fee f_{base} and the proportional fee f_{prop} are constants. Let $A_{LR}(\tau)$, $A_{RL}(\tau)$ be the externally arriving amounts coming from node L in the L-to-R direction and from node R in the R-to-L direction at time τ respectively, each drawn from a distribution that is fixed but unknown to node N. An arriving transaction of amount $A_{LR}(\tau) = \alpha$ is feasible if and only if there is enough balance in the L-to-R direction in both channels, i.e. $b_{LN}(\tau) \geq \alpha$ and $b_{NR} \geq \alpha - f(\alpha)$, and similarly for the R-to-L direction. The successfully processed amounts by N at time τ are[3]:

$$S_{LR}(\tau) = \begin{cases} A_{LR}(\tau) & \text{, if } A_{LR}(\tau) \leq b_{LN}(\tau) \text{ and } A_{LR}(\tau) - f(A_{LR}(\tau)) \leq b_{NR}(\tau) \\ 0 & \text{, otherwise} \end{cases}$$

and symmetrically for $S_{RL}(\tau)$.

The profit of node N at time τ is $f(S_{LR}(\tau)) + f(S_{RL}(\tau))$, and the lost fees (from transactions that potentially failed to process) are $f\big(A_{LR}(\tau) - S_{LR}(\tau)\big) + f\big(A_{RL}(\tau) - S_{RL}(\tau)\big)$. The balance processes at time τ evolve as follows (the on-chain amount $B_N(\tau)$ is not affected by the processing of off-chain transactions; channel NR behaves symmetrically):

$$b_{LN}(\tau) \to b_{LN}(\tau) + (S_{RL}(\tau) - f(S_{RL}(\tau))) - S_{LR}(\tau)$$
$$b_{NL}(\tau) \to b_{NL}(\tau) + S_{LR}(\tau) - (S_{RL}(\tau) - f(S_{RL}(\tau)))$$

The rebalancing decision (control) timescale The evolution of the system can be controlled by node N using submarine swap rebalancing operations. Rebalancing may start at times $t_i = i \cdot T_{\text{check}}$, $i = 0, 1, ...$, and takes a (fixed) time T_{conf} to complete (on average 10 minutes for Bitcoin). We consider the case where $T_{\text{check}} \geq T_{\text{conf}}$ (to avoid having concurrent rebalancing operations in the same channel that could be combined into one). The system state is defined only for the discrete timescale as the collection of the off- and on-chain balances:

$$S(t_i) = \big(b_{LN}(t_i), b_{NL}(t_i), b_{NR}(t_i), b_{RN}(t_i), B_N(t_i)\big) \tag{1}$$

At each time t_i, node N can decide to request a swap-in or a swap-out in each channel. Call the respective amounts $r_L^{\text{in}}(t_i), r_L^{\text{out}}(t_i), r_R^{\text{in}}(t_i), r_R^{\text{out}}(t_i)$. At any time t_i, in a given channel, either a swap-in or a swap-out or nothing will be requested, but not both a swap-in and a swap-out[4].

Let $F_{\text{swap}}^{\text{in}}(\alpha)$ and $F_{\text{swap}}^{\text{out}}(\alpha)$ be the swap fees that the LSP charges for an amount α for a swap-in and a swap-out respectively, where $F_{\text{swap}}^{\text{in}}(\cdot)$ and $F_{\text{swap}}^{\text{out}}(\cdot)$

[3] Since in the sequel we focus on the discrete and sparse time scale of the periodic times at which the node rebalances, we make the fair assumption (as e.g. in [2]) that off-chain transactions are processed instantaneously across their entire path and do not fail in their subsequent steps after they cross the two channels (if a transaction were to fail outside the two channels, it can be viewed as of zero value by the system).

[4] Nodes L and R are considered passive: they perform no swap operations themselves.

are any functions with $F_{\text{swap}}(0) = 0$. For ease of exposition, we let all types of fees the node will have to pay (relay fees for the off-chain part, on-chain miner fees, server fees) be both part of the above swap fees, and be the same for both swap-ins and swap-outs when a net amount r_{net} is transferred from on- to off-chain or vice versa: $F^{\text{in}}_{\text{swap}}(r_{\text{net}}) = F^{\text{out}}_{\text{swap}}(r_{\text{net}}) = F_{\text{swap}}(r_{\text{net}}) \triangleq r_{\text{net}}F + M$, where the proportional part F includes the server fee and off-chain relay fees, and M includes the miner fee and potential base fees.

Note that the semantics of the swap amounts r are such that they represent the amount that will move *in the channel* (and not necessarily the net change in the node's fortune). As a result of this convention and based on the swap operation as described in the following paragraph, the amount r^{in} of a swap-in coincides with the net amount $r^{\text{in}}_{\text{net}}$ by which the node's fortune decreases (as r^{in} does not include the swap fee), while the amount r^{out} of a swap-out includes the swap fee and the net amount by which the node's fortune decreases is $r^{\text{out}}_{\text{net}} = \phi^{-1}(r^{\text{out}})$, where $\phi(r_{\text{net}}) \triangleq r_{\text{net}} + F_{\text{swap}}(r_{\text{net}})$, and ϕ^{-1} is the generalized inverse function of $\phi(\cdot)$ (it always exists: $\phi^{-1}(y) = \min\{x \in \mathbb{N} : \phi(x) = y\}$). For our $F_{\text{swap}}(\cdot)$ it is $\phi(r_{\text{net}}) = r_{\text{net}}(1 + F) + M$ for $r_{\text{net}} > 0$, $\phi(0) = 0$, so $\phi^{-1}(y) = (y - M)/(1 + F)$ for $y > 0$ and $\phi^{-1}(0) = 0$.

A submarine swap step-by-step We now describe how a rebalancing operation on the Nn channel is affecting the system state. First, a swap-in of amount r^{in}_n initiated by node N to refill N's local balance in the Nn channel:

- At time t_i, node N locks the net rebalancing amount plus fees and subtracts it from its on-chain funds: $B_N \to B_N - (r^{\text{in}}_n + F^{\text{in}}_{\text{swap}}(r^{\text{in}}_n))$
- At time $t_i + T_{\text{conf}}$, the on-chain transaction is confirmed, so the LSP sends a payment of r^{in}_n to node N off-chain[5]. The payment reaches node n:
  - If $b_{nN} \leq r^{\text{in}}_n$ (i.e. n does not have enough balance to forward it), then the off-chain payment fails. The on-chain funds are unlocked and refunded back to the on-chain amount: $B_N \to B_N + (r^{\text{in}}_n + F^{\text{in}}_{\text{swap}}(r^{\text{in}}_n))$
  - Otherwise (if the transaction is feasible), n forwards the payment to N: $b_{nN} \to b_{nN} - r^{\text{in}}_n$ and $b_{Nn} = b_{Nn} + r^{\text{in}}_n$

A swap-out of amount r^{out}_n, initiated by node N to offload some of its local balance to the chain, works as follows:

- At time t_i, node N locks the net rebalancing amount plus fees and sends it to the LSP via the off-chain network: $b_{Nn} \to b_{Nn} - r^{\text{out}}_n$. Note that r^{out}_n includes the fees.
- At time $t_i + T_{\text{conf}}$, the on-chain transaction is confirmed, so node N receives the funds on-chain: $B_N \to B_N + \phi^{-1}(r^{\text{out}}_n)$, and the funds are also unlocked in the channel and pushed towards the remote balance: $b_{nN} \to b_{nN} + r^{\text{out}}_n$

[5] The LSP is a well-connected node owning large amounts of liquidity, so we reasonably assume that it can always find a route from itself to N, possibly via splitting the amount across multiple paths.

Rebalancing constraints Based on the steps just described, swap operations will succeed if and only if their amounts satisfy the following constraints:

- Rebalancing amounts must be non-negative:

$$r_n^{\text{in}}(t_i), r_n^{\text{out}}(t_i) \geq 0 \text{ for all } i \in \mathbb{N}, n \in \mathcal{N} \tag{2}$$

- A swap-in and a swap-out cannot be requested in the same channel at the same time:

$$r_n^{\text{in}}(t_i) \cdot r_n^{\text{out}}(t_i) = 0 \text{ for all } i \in \mathbb{N}, n \in \mathcal{N} \tag{3}$$

- The swap-out amounts (which already include the swap fees) must be greater than the fees themselves:

$$r_n^{\text{out}}(t_i) - F_{\text{swap}}^{\text{out}}(r_n^{\text{out}}(t_i)) \geq 0 \text{ for all } i \in \mathbb{N}, n \in \mathcal{N} \tag{4}$$

- The respective channel balances must suffice to cover the swap-out amounts (which already include the swap fees):

$$r_n^{\text{out}}(t_i) \leq b_{Nn}(t_i) \text{ for all } i \in \mathbb{N}, n \in \mathcal{N} \tag{5}$$

- The on-chain balance must suffice to cover the total swap-in amount plus fees:

$$\sum_{n \in \mathcal{N}} \left(r_n^{\text{in}}(t_i) + F_{\text{swap}}^{\text{in}}(r_n^{\text{in}}(t_i)) \right) \leq B_N(t_i) \text{ for all } i \in \mathbb{N} \tag{6}$$

State evolution equations The amounts added to each balance due to successful transactions during the interval (t_i, t_{i+1}) are:

$$d_{NL}^{(t_i,t_{i+1})} \triangleq \int_{\tau \in (t_i,t_{i+1})} \Big(S_{LR}(\tau) - (S_{RL}(\tau) - f(S_{RL}(\tau))) \Big) d\tau,$$

similarly for $d_{NR}^{(t_i,t_{i+1})}$, and $d_{nN}^{(t_i,t_{i+1})} \triangleq -d_{Nn}^{(t_i,t_{i+1})}$. Then for actions taken subject to the constraints (2)–(6), the state evolves as follows:

$$b_{nN}(t_{i+1}) = b_{nN}(t_i) + d_{nN}^{(t_i,t_{i+1})} - (r_n^{\text{in}}(t_i) - z_n(t_i)) + r_n^{\text{out}}(t_i)$$
$$b_{Nn}(t_{i+1}) = b_{Nn}(t_i) + d_{Nn}^{(t_i,t_{i+1})} + (r_n^{\text{in}}(t_i) - z_n(t_i)) - r_n^{\text{out}}(t_i)$$
$$B_N(t_{i+1}) = B_N(t_i) - \sum_{n \in \mathcal{N}} \left(r_n^{\text{in}}(t_i) - F_{\text{swap}}^{\text{in}}(r_n^{\text{in}}(t_i)) \right) + \sum_{n \in \mathcal{N}} \phi^{-1}(r_n^{\text{out}}(t_i)) + \sum_{n \in \mathcal{N}} w_n(t_i)$$

where $z_n(t_i)$ and $w_n(t_i)$ are the refunds of the swap-in amount off- and on-chain respectively in case a swap-in operation fails:

$$z_n(t_i) = r_n^{\text{in}}(t_i) \mathbb{1}\{b_{nN}(t_i) + d_{nN}^{(t_i,t_i+T_{\text{conf}})} < r_n^{\text{in}}(t_i)\} \tag{7}$$
$$w_n(t_i) = z_n(t_i) + F_{\text{swap}}^{\text{in}}(z_n(t_i)) \tag{8}$$

3.2 Writing the problem as a Markov Decision Process

The objective function the node wishes to maximize in the real world is its total fortune both in the channels and on-chain (another equivalent objective is discussed in Appendix C). The fortune increase due to the action (the 4-tuple) $r(t_i)$ taken at step t_i is:

$$D(t_i, r(t_i)) \triangleq \left(\sum_{n \in \mathcal{N}} b_{Nn}(t_{i+1}) + B_N(t_{i+1}) \right) - \left(\sum_{n \in \mathcal{N}} b_{Nn}(t_i) + B_N(t_i) \right)$$

A control policy $\pi = \{(t_i, r^\pi(t_i))\}_{i \in \mathbb{N}}$ consists of the times t_i and the corresponding actions $r^\pi(t_i) = \left(r_L^{\text{in}}(t_i), r_L^{\text{out}}(t_i), r_R^{\text{in}}(t_i), r_R^{\text{out}}(t_i)\right)$ taken from the set of allowed actions $\mathcal{R} = [0, C_L]^2 \times [0, C_R]^2$, and belongs to the set of admissible policies

$$\mathit{\Pi} = \left\{ \{(t_i, r(t_i))\}_{i \in \mathbb{N}} \text{ such that } r(t_i) \in \mathcal{R} \text{ for all } i \in \mathbb{N} \right\}$$

Ultimately, the goal of node N is to find a rebalancing policy that maximizes the long-term average expected fortune increase D over all admissible policies:

$$\underset{\pi \in \mathit{\Pi}}{\text{maximize}} \lim_{H \to \infty} \frac{1}{t_H} \sum_{i=0}^{H} \mathbb{E}\left[D(t_i, r^\pi(t_i))\right] \quad \text{subject to (2)–(6)}$$

4 Heuristic and Reinforcement Learning-based policies

In this section, we describe the steps we took in order to apply DRL to approximately solve the formulated MDP. We first outline two heuristic policies, which we will use later to benchmark our DRL-based solution.

4.1 Heuristic policies

Autoloop [18,7] is a policy that allows a node to schedule automatic swap-ins (resp. swap-outs) if its local balance falls below a minimum (resp. rises above a maximum) threshold expressed as a percentage of the channel's capacity. The initiated swap is of amount equal to the difference of the local balance from the midpoint, i.e. the average of the two thresholds. The pseudocode can be found in Alg. 1. We expect Autoloop to be suboptimal with respect to profit maximization in certain cases, as it does not take the expected demand into account and thus possibly performs rebalancing at times when it is not necessary.

This motivates us to define another heuristic policy that incorporates the empirical demand information. We call this policy Loopmax, as its goal is to rebalance with the maximum possible amount and as infrequently as possible (without sacrificing transactions), based on the demand. Loopmax keeps track of the total arriving amounts, and estimates the net change of each balance per unit time using the difference of the total amounts that arrived in each direction (the NR channel is symmetric):

$$\hat{A}_{LN}^{\text{net}}(\tau) = -\hat{A}_{NL}^{\text{net}}(\tau) \triangleq \frac{1}{\tau} \int_{t \in [0,\tau]} \Big(A_{RL}(t) - f(A_{RL}(t)) - A_{LR}(t) \Big) dt \qquad (9)$$

For each channel, we first calculate its estimated time to depletion ($ETTD$) or saturation ($ETTS$), depending on the direction of the net demand and the current balances, and using this time we dispatch a swap of the appropriate type not earlier than $T_{\text{check}} + T_{\text{conf}}$ before depletion/saturation, and of the maximum possible amount. The rationale is that if e.g. $ETTD \geq T_{\text{check}} + T_{\text{conf}}$, the policy can leverage this fact to postpone starting a swap until the next check time, since until then no transactions will have been dropped. If $ETTD < T_{\text{check}} + T_{\text{conf}}$ though, the policy should act now, as otherwise it will end up dropping transactions during the following interval of $T_{\text{check}}+T_{\text{conf}}$. The maximum possible swap-out is constrained by the local balance at that time, while the maximum possible swap-in is constrained by the remote balance at that time[6] and the on-chain amount: an on-chain amount of B_N can support (by including fees) a net swap-in amount of at most $\phi^{-1}(B_N)$. The pseudocode can be found in Alg. 2. Compared to Autoloop, Loopmax has the advantage that it rebalances only when it is absolutely necessary and can thus achieve savings in swap fees. On the other hand, Loopmax's aggressiveness can lead it to extreme rebalancing decisions when traffic is quite skewed in a particular direction (e.g. it can do a swap-in of almost the full capacity, which is very likely to fail due to randomness in the transaction arrivals). A small modification we can use on top of Alg. 2 to alleviate this is to define certain safety margins of liquidity that Loopmax should always leave intact on each side of the channel, so that incoming transactions do not find it depleted due to a large pending swap.

4.2 Deep reinforcement learning algorithm design

Having formulated the problem as an MDP, we now need to find an (approximately) optimal policy. The problem is challenging for a number of reasons:

- The problem dynamics are not linear.
- The state and action spaces are continuous and thus tabular approaches are not applicable.
- There are time-dependent constraints on the actions.
- Choosing to not rebalance at a specific time requires special treatment, as otherwise the zero action will be sampled from a continuous action space with zero probability.

To tackle these challenges, we resort to approximate methods, and specifically Reinforcement Learning (RL). In the standard RL framework, an agent makes decisions based on a policy represented as a distribution $p : p(s, a) \to [0, 1]$, with $p(s, a)$ being the probability that action a will be taken when the environment is in state s. Since our problem has continuous state and action spaces and the policy cannot be stored in tabular form, we need to use function approximation techniques. Neural networks serve well the role of function approximators

[6] Actually, it is constrained by the remote balance at the time of the swap-in's completion. We will improve this later using estimates of future balances.

in many applications [20,5]. Some algorithms appropriate for this type of problems are Deep Deterministic Policy Gradient (DDPG) [19] and Soft Actor-Critic (SAC) [15]. We decided to use the latter as DDPG is known to exhibit extreme brittleness and hyperparameter sensitivity [11]. We now describe our methodology around how we engineer our DRL algorithm based on the vanilla SAC in order to arrive at a solution that deals with all the above challenges.

For the RL agent's environment, we use as state the five balances (off- and on-chain) and the estimates of the remote balances at the time of the swap completion, each normalized appropriately: by the respective channel's capacity, or by a total target fortune in the on-chain amount's case. Thus, our state space is $[0,1]^7$. As actions, instead of the 4-tuple of Sec. 3, we use an equivalent (due to (3)) 2-tuple (r_L, r_R), i.e. a single variable for each channel that can take both positive (swap-in) and negative (swap-out) values. Before the raw sampled action is applied, it undergoes some processing described in the sequel.

Raw actions are sampled from the entire continuous action space; thus the zero action will be selected with zero probability. In reality, though, performing zero rebalancing in a channel when a swap is not necessary is important for minimizing the costs, and an action the agent should learn to apply. To make the zero action selectable with positive probability, and at the same time prevent the agent from performing swaps too small in size, we force the respective applied action to be zero if the raw action coordinate is less than a threshold ρ_0 (e.g. 20%) of the channel capacity. Moreover, the vanilla SAC algorithm [15] operates on an action space that is a compact subset of $\mathbb{R}^k$ for all decision times. In our case, though, the allowed actions vary due to the time-dependent constraints (2)–(6). We therefore define the action space to be $[-1,1]^2$, where each coordinate denotes the percentage not of the entire channel capacity, but of the maximum amount available for the respective type of swap at that moment.

All constraints are already decoupled per channel, with the exception of (6), which though can also be decoupled as explained in Appendix E.1. Another useful observation is that when a swap-in is about to complete time T_{conf} after it was requested, the remote balance in the respective channel needs to suffice (otherwise the swap-in will fail and a refund will be triggered as in (8)):

$$r_n^{\text{in}}(t_i) \leq b_{nN}(t_i) + d_{nN}^{(t_i, t_i+T_{\text{conf}})} \text{ for all } i \in \mathbb{N}, n \in \mathcal{N} \tag{10}$$

We calculate an estimate $\hat{b}_{nN}(t_i + T_{\text{conf}})$ of the right-hand side of (10) based on the past history, with the details of the calculation given in Appendix E.2. Let $\rho_{min}^{\text{out}} \triangleq M/(1-F)$ be the minimum solution of (4). As long as $\rho_0 C_n \gg \rho_{min}^{\text{out}}$, which should hold in practice as ρ_{min}^{out} is very small, we can write all constraints (2)–(6), (10) in terms of the 2-tuple (r_L, r_R) as follows:

$$r_n \in \left[-b_{Nn}, \min\{\hat{b}_{nN}(t_i + T_{\text{conf}}), \phi^{-1}(B_N(t_i)), C_n\}\right], n \in \mathcal{N}$$

The described mapping of raw actions (sampled from the distribution on the entire action space) to the finally applied actions is summarized in Table 1.

We craft the reward signal to guide the agent towards optimizing the objective: we add the node's fortune increase (3.2) until the next check time, subtract

the fee losses from transactions dropped until the next check time, and also subtract a fixed penalty for every swap the algorithm initiates and which eventually fails. A high-level sketch of the most important components of the final learning process described in this section is given in Alg. 3. We call the emerging policy "RebEL": Rebalancing Enabled by Learning.

5 Evaluation

In order to evaluate the performance of different rebalancing policies, we build a discrete event simulator of a relay node with two payment channels and rebalancing capabilities using Python SimPy[7]. The simulator treats each channel as a resource allowed to undergo at most one active swap at a time, and allows for parameterization of the initial balances, the transaction generation distributions (frequency, amount, number) in both directions, the different fees, the swap check and confirmation times, the rebalancing policy and its parameters[8].

We simulate a relay node with two payment channels, each of capacity \$1000 split equally between the nodes. Transactions arrive from both sides as Poisson processes. We evaluate policies Autoloop, Loopmax and RebEL defined in Sec. 4, as well as the *None* policy that never performs any rebalancing. We use $T_{\text{check}} = T_{\text{conf}} = 10$ minutes, miner fee $M = \$2$/on-chain transaction (tx), base fee[9] $f_{\text{base}} = 0$, swap fee $F = 0.5\%$, 0.3 and 0.7 as the low and high liquidity thresholds of Autoloop, and 2 minutes worth of estimated traffic as safety margins for Loopmax. We run all experiments on a regular consumer laptop.

We experimented with different hyperparameters for the original SAC algorithm[10] as well as for RebEL parameters and reward shapes, and settled with the ones shown in Appendix F. We performed experiments for the transaction amount distribution being Uniform in [0, 50] and Gaussian with mean 25 and standard deviation 20, and the results were very similar. Therefore, all plots shown below are for the Gaussian amounts.

The role of fees Current median fee rates for transaction forwarding are in the order of $3 \cdot 10^{-5}$ (\$/\$) or 0.003%, while swap server fees are in the order of 0.5% and miner fees are in the order of 2 \$/tx[11]. In order to see if a relay node can make a profit with such fees, we perform the following back-of-the-envelope calculation: A swap-in of amount r will cost the node $rF + M$ in fees and will enable traffic of at most value r to be processed, which will yield profits rf_{prop} from relay fees. Therefore, the swap-in cannot be profitable if $rF + M \geq rf_{\text{prop}}$.

[7] https://simpy.readthedocs.io

[8] The code is publicly available at https://anonymous.4open.science/r/payment-channel-rebalancing-5C8F.

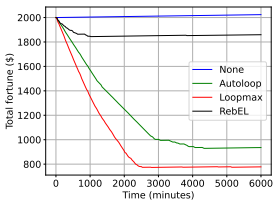
[9] Currently, according to https://lnrouter.app/graph/zero-base-fee, almost 50% of the Lightning Network uses $f_{\text{base}} = 0$.

[10] We used the PyTorch implementation in https://github.com/pranz24/pytorch-soft-actor-critic.

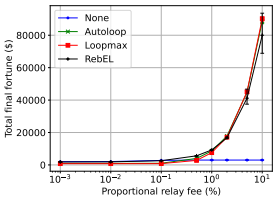
[11] Fee value sources: https://1ml.com/statistics, https://lightning.engineering/loop, https://ycharts.com/indicators/bitcoin_average_transaction_fee

Solving this inequality, we see that no positive amount r can be profitable if $f_{\text{prop}} \leq F$, while if $f_{\text{prop}} > F$ a necessary (but not sufficient) condition for profitability is $r > M/(f_{\text{prop}} - F)$. The respective inequality for a swap-out of amount r is $r - \frac{r-M}{1+F} \geq r f_{\text{prop}}$, which shows that for $f_{\text{prop}} \leq \frac{F}{1+F}$ no amount can be profitable and for $f_{\text{prop}} > \frac{F}{1+F}$ a necessary condition for profitability is that $r > \frac{M}{f_{\text{prop}}(1+F)-F}$. *With the current fees, we are in the non-profitable regime.* Although the above inequalities are short-sighted in that they focus only on a specific action time, they do confirm the observation made by both the Lightning and the academic communities [6] that in order for relay nodes to be a profitable business, relay fees have to increase.

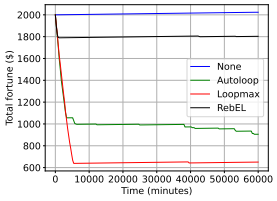
We now perform an experiment confirming this finding with the currently used fee values. We simulate a workload of demand in the L-to-R direction: 60000 L-to-R and 15000 R-to-L transactions under a high (10 tx/minute L-to-R, 2.5 tx/minute R-to-L) and a low (1 tx/minute L-to-R, 0.25 tx/minute R-to-L) intensity. The node's total fortune over time for high and low intensity are shown in Figs. 5(a) and 5(c) respectively. We see that regardless of the (non-*None*) rebalancing policy, the node's fortune decreases over time, because rebalancing fees surpass any relay profits, which are small because of the small f_{prop} compared to F. In this regime, the node is better off not rebalancing at all. Still, our RebEL policy manages to learn this fact and after some point exhibits the desired behavior and stops rebalancing as well. Autoloop and Loopmax keep trying to rebalance and end up exhausting their entire on-chain balance, so the total fortune under them gets stuck after some point.



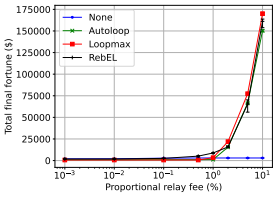
(a) Total fortune over time under high skewed demand



(b) Total final fortune under high skewed demand for different values of f_{prop}



(c) Total fortune over time under low skewed demand

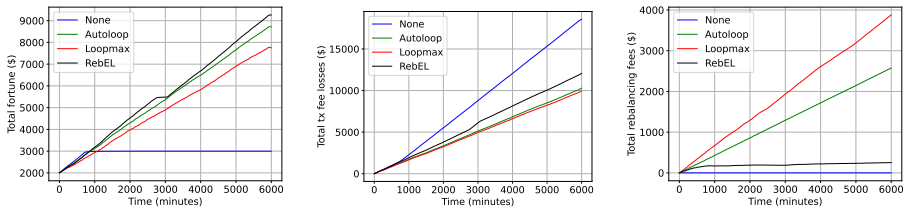


(d) Total final fortune under low skewed demand for different values of f_{prop}

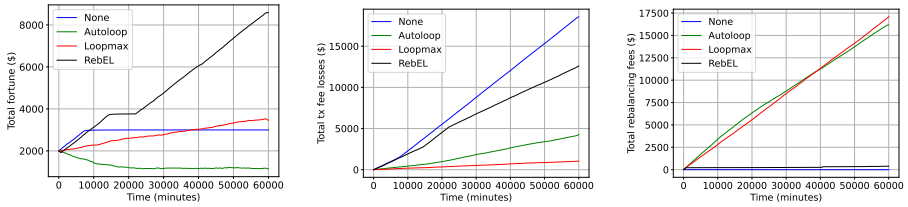
Fig. 5. Experiments with different relay fee f_{prop}

Taking a higher level view, we also conduct multiple experiments with the same demand as before but now while varying f_{prop}. The results of the total final fortune of each experiment (run for the same total time and averaged over 10 runs; error bars show the maximum and minimum values) are shown in Fig. 5(b) under high demand and in Fig. 5(d) under low demand. We see that no rebalancing policy is profitable (i.e. better than *None*) as long as $f_{\text{prop}} < 0.5\% =$

F, which confirms our back-of-the-envelope calculation. For higher values of f_{prop}, the node is able to make a profit (see Appendix G.1 for more remarks).



(a) Total fortune over time (high demand) (b) Transaction fee losses over time (high demand) (c) Rebalancing fees over time (high demand)



(d) Total fortune over time (low demand) (e) Transaction fee losses over time (low demand) (f) Rebalancing fees over time (low demand)

Fig. 6. Total fortune, transaction fee losses and rebalancing fees over time under demand skewed in the L-to-R direction

The role of the demand We now stay in the fee regime of possible profitability by keeping $f_{\text{prop}} = 1\%$, and study the role of the demand (and indirectly of the depletion frequency) on the performance of the different policies. The results for the same high and low workload of skewed demand in the L-to-R direction as before are shown in Fig. 6. RebEL outperforms all other policies under both demand regimes (Figs. 6(a), 6(d)), as it manages to strike a balance in terms of frequency and amount of rebalancing and transaction fee profits. This happens in a few 10-minute iterations under high demand (a few hours in real time), because balance changes are more pronounced in this case and help RebEL learn faster, while it takes about 1200 iterations under low demand, translating in 8.3 days of training. Both these training times are reasonable for a relay node investing its capital to make a profit. We see that under both regimes the system without rebalancing (*None* policy) at some point reaches a state where almost all the balances are accumulated locally and no transactions can be processed anymore (hence the flattening in the *None* curve). Under high demand, Autoloop and Loopmax rebalance a lot (Fig. 6(c)) in order to minimize transaction fee losses (Fig. 6(b)), while RebEL sacrifices some transactions to achieve higher total fortune. Under low demand, RebEL rebalances only when necessary (Fig. 6(f)), even if this means sacrificing many more transactions (Fig. 6(e)), simply because rebalancing is not worth it at that low demand regime, in the sense that the potential profits during the 10-minute rebalancing check times

are too low to justify the frequent rebalancing operations that the other policies apply. Loopmax eventually achieves a profit (although much lower than RebEL) because it tends to rebalance with higher amounts. On the contrary, Autoloop rebalances with small amounts, thus incurring significant costs from constant miner fees and eventually even making a loss compared to the initial node's fortune (Fig. 6(d)). For more remarks and experiments, the interested reader is referred to Appendix G. In the special case of equal demands from both sides, RebEL does not perform as well. However, although possible, this scenario is much less likely to occur in practice, as usually the traffic follows some patterns, e.g. from clients to merchants. The skewed demand scenario, where RebEL is superior, is also the most natural. We also examined how the initial conditions (capacities, initial balances) affect the performance, and see that under skewed demand RebEL continues to perform well in all cases.

6 Related work

Rebalancing via payments from a node to itself via a circular path of channels has been studied by several works (e.g. [17,25,1]), with some taking relay fees into account (as we did) and some not. [10,13] describe fee strategies that incentivize the balanced use of payment channels. [2] uses a game-theoretic lens to study the extent to which nodes can pay lower transaction fees by waiting patiently and reordering transactions instead of pursuing maximum efficiency. Perhaps the only work on submarine swaps is [12], which considers the problem of the appropriate fee design by liquidity providers according to usage patterns. A recent development similar to submarine swaps is PeerSwap [28]: instead of buying funds from an LSP, a node can exchange funds on-/off-chain with its channel neighbor directly. *Splicing* is another mechanism that replaces a channel with a new one with a different capacity while allowing transactions to flow in the meantime [27].

Stochastic modeling and optimization in the blockchain space has been used both in layer-1 [9,14,22,21] for performance characterization, and in layer-2 for routing [29] and scheduling [24] of payments. Deep Reinforcement Learning has been broadly applied to approximately solve challenging optimization problems from various areas and to build systems that learn to manage resources directly from experience. For example, [20] applies DRL to the resource allocation problem of packing tasks under multiple resource demands, while [3] uses DRL to solve the MDP modeling selfish mining attacks in Bitcoin-like blockchains.

7 Conclusion

In this paper, we studied the problem of relay node profit maximization using submarine swaps, and demonstrated the feasibility of applying state-of-the-art DRL techniques for solving it. Future work can explore swap-based rebalancing in a network setting, the comparison of different rebalancing methods, and the incentive problems arising. We hope that this research will inspire further interest

in designing capital management strategies in the complex world of PCNs based on learning from experience as an alternative to currently applied heuristics, and will be a step towards guaranteeing the profitability of the relay nodes and, consequently, the viability and scalability of the PCNs they sustain.

References

1. Avarikioti, Z., Pietrzak, K., Salem, I., Schmid, S., Tiwari, S., Yeo, M.: Hide & Seek: Privacy-preserving rebalancing on Payment Channel Networks. In: Eyal, I., Garay, J. (eds.) Financial Cryptography and Data Security. pp. 358–373. Springer International Publishing, Cham (2022)
2. Bai, Q., Xu, Y., Wang, X.: Understanding the benefit of being patient in payment channel networks. IEEE Transactions on Network Science and Engineering **9**(3), 1895–1908 (2022)
3. Bar-Zur, R., Abu-Hanna, A., Eyal, I., Tamar, A.: WeRLman: To tackle whale (transactions), go deep (RL). In: Proceedings of the 15th ACM International Conference on Systems and Storage. p. 148. SYSTOR '22, Association for Computing Machinery, New York, NY, USA (2022)
4. Bosworth, A.: Submarine swaps on the Lightning Network (2018), https://submarineswaps.github.io
5. Boute, R.N., Gijsbrechts, J., van Jaarsveld, W., Vanvuchelen, N.: Deep reinforcement learning for inventory control: A roadmap. Eur. J. Oper. Res. **298**(2), 401–412 (2022)
6. Béres, F., Seres, I.A., Benczúr, A.A.: A cryptoeconomic traffic analysis of Bitcoin's Lightning Network. Cryptoeconomic Systems (6 2020), https://cryptoeconomicsystems.pubpub.org/pub/b8rb0ywn
7. Carla Kirk-Cohen: Autoloop: Lightning Liquidity You Can Set and Forget! (2020), https://lightning.engineering/posts/2020-11-24-autoloop
8. Croman, K., Decker, C., Eyal, I., Gencer, A.E., Juels, A., Kosba, A., Miller, A., Saxena, P., Shi, E., Sirer, E.G., Song, D., Wattenhofer, R.: On scaling decentralized blockchains. In: Clark, J., Meiklejohn, S., Ryan, P.Y., Wallach, D., Brenner, M., Rohloff, K. (eds.) Financial Cryptography and Data Security. pp. 106–125. Springer Berlin Heidelberg, Berlin, Heidelberg (2016)
9. Dembo, A., Kannan, S., Tas, E.N., Tse, D., Viswanath, P., Wang, X., Zeitouni, O.: Everything is a race and Nakamoto always wins. In: Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security. p. 859–878. CCS '20, ACM (2020)
10. Di Stasi, G., Avallone, S., Canonico, R., Ventre, G.: Routing payments on the Lightning Network. In: iThings/GreenCom/CPSCom/SmartData. pp. 1161–1170. IEEE (2018)
11. Duan, Y., Chen, X., Houthooft, R., Schulman, J., Abbeel, P.: Benchmarking deep reinforcement learning for continuous control. In: Proceedings of the 33rd International Conference on International Conference on Machine Learning - Volume 48. p. 1329–1338. ICML'16, JMLR.org (2016)
12. Echenique, J.I.R., Burtey, N.: Pricing liquidity for Lightning wallets (2022), https://github.com/GaloyMoney/liquidity-fees-paper
13. van Engelshoven, Y., Roos, S.: The merchant: Avoiding payment channel depletion through incentives. In: IEEE International Conference on Decentralized Applications and Infrastructures, DAPPS 2021, Online Event, August 23-26, 2021. pp. 59–68. IEEE (2021)

14. Gaži, P., Kiayias, A., Russell, A.: Tight consistency bounds for Bitcoin. In: Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security. p. 819–838. CCS '20, Association for Computing Machinery, New York, NY, USA (2020)
15. Haarnoja, T., Zhou, A., Hartikainen, K., Tucker, G., Ha, S., Tan, J., Kumar, V., Zhu, H., Gupta, A., Abbeel, P., Levine, S.: Soft actor-critic algorithms and applications (2018), http://arxiv.org/abs/1812.05905
16. Jager, J.: Loop Out in-depth (2019), https://blog.lightning.engineering/technical/posts/2019/04/15/loop-out-in-depth.html
17. Khalil, R., Gervais, A.: Revive: Rebalancing off-blockchain payment networks. In: Thuraisingham, B.M., Evans, D., Malkin, T., Xu, D. (eds.) Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security, CCS 2017, Dallas, TX, USA, October 30 - November 03, 2017. pp. 439–453. ACM (2017)
18. Lightning Labs: Autoloop (2022), https://github.com/lightninglabs/loop/blob/master/docs/autoloop.md
19. Lillicrap, T.P., Hunt, J.J., Pritzel, A., Heess, N., Erez, T., Tassa, Y., Silver, D., Wierstra, D.: Continuous control with deep reinforcement learning. In: Bengio, Y., LeCun, Y. (eds.) 4th International Conference on Learning Representations, ICLR 2016, San Juan, Puerto Rico, May 2-4, 2016, Conference Track Proceedings (2016)
20. Mao, H., Alizadeh, M., Menache, I., Kandula, S.: Resource management with deep reinforcement learning. In: Proceedings of the 15th ACM Workshop on Hot Topics in Networks. p. 50–56. HotNets '16, Association for Computing Machinery, New York, NY, USA (2016)
21. Mišić, J., Mišić, V.B., Chang, X., Motlagh, S.G., Ali, M.Z.: Modeling of Bitcoin's blockchain delivery network. IEEE Transactions on Network Science and Engineering **7**(3), 1368–1381 (2020)
22. Papadis, N., Borst, S., Walid, A., Grissa, M., Tassiulas, L.: Stochastic models and wide-area network measurements for blockchain design and analysis. In: IEEE INFOCOM 2018 - IEEE Conference on Computer Communications. pp. 2546–2554. IEEE (2018)
23. Papadis, N., Tassiulas, L.: Blockchain-based payment channel networks: Challenges and recent advances. IEEE Access **8**, 227596–227609 (2020)
24. Papadis, N., Tassiulas, L.: Payment Channel Networks: Single-hop scheduling for throughput maximization. In: IEEE INFOCOM 2022 - IEEE Conference on Computer Communications. pp. 900–909 (2022)
25. Pickhardt, R., Nowostawski, M.: Imbalance measure and proactive channel rebalancing algorithm for the Lightning Network. In: IEEE International Conference on Blockchain and Cryptocurrency, ICBC 2020, Toronto, ON, Canada, May 2-6, 2020. pp. 1–5. IEEE (2020)
26. Poon, J., Dryja, T.: The Bitcoin Lightning Network: scalable off-chain instant payments (2016), https://lightning.network/lightning-network-paper.pdf
27. Russell, R.: Splicing Proposal (2018), https://lists.linuxfoundation.org/pipermail/lightning-dev/2018-October/001434.html
28. Togami, W., Nick, K.: PeerSwap: Decentralized P2P LN Balancing Protocol (2021), https://blockstream.com/assets/downloads/2021-11-16-PeerSwap_Announcement.pdf
29. Varma, S.M., Maguluri, S.T.: Throughput optimal routing in blockchain-based payment systems. IEEE Transactions on Control of Network Systems **8**(4), 1859–1868 (2021)

A Causes of channel depletion

Channel depletion might happen due to a number of reasons:

- Asymmetric demand inside single channels
- The random nature of arrivals causing temporary depletions at specific times (e.g. when a large transaction arrives)
- Symmetric demand between two endpoints of a multihop path which can cause imbalance due to fees withheld by intermediate nodes

An example of the third and more subtle case is given in Fig. 7, which shows the evolution over time of a subnetwork of three channels with symmetric demand of amount 20 arriving alternately from either side of the path. When each transaction is relayed by node B, a 50% fee is withheld and the remaining amount of 10 is forwarded to the next channel in the path. We see that even though the end-to-end path demand is symmetric, after a few steps the channels get unbalanced and stop being able to process any more transactions[12].

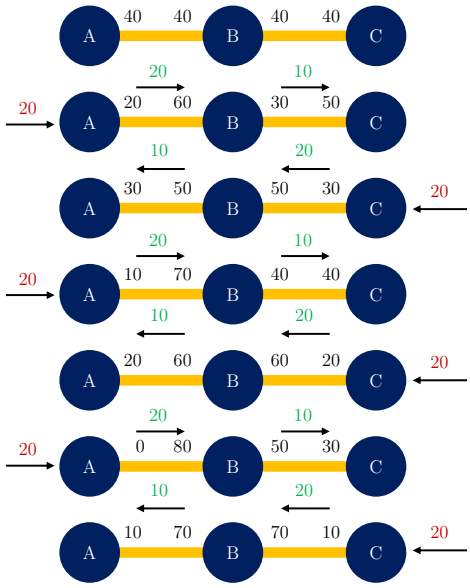


Fig. 7. An example of a PCN getting stuck even though the demand is symmetric. Demand is shown in red, forwarded amounts after a 50% fee withholding are shown in green, and channel balances are shown in black.

[12] The 50% fee is not realistic and is only used for the purposes of this example. With the real much lower fees the channels will similarly get stuck after a larger number of steps.

B The submarine swap protocol

A sketch of the technical protocol followed during a successful swap-in, which is the basis for the modeling of Sec. 3.1, is shown in Fig. 8 (a swap-out is similar). First, a node-client initiates the swap by generating a hash preimage, creating an invoice of the desired swap amount r tied to this hash and with a certain expiration time T_{exp}, and sends it to an LSP that is willing to make the exchange. The LSP then quotes what it wants to be paid on-chain in exchange for paying the client's invoice off-chain, say $\alpha + F_{\text{swap}}(\alpha)$, where $F_{\text{swap}}(\alpha)$ is the LSP's swap service fee. If the client accepts the exchange rate, it creates a conditional on-chain payment of amount $\alpha + F_{\text{swap}}(\alpha)$ to the LSP based on an HTLC with the same preimage as before and broadcasts the payment to the blockchain network. The payment can only be redeemed if the LSP knows the preimage, and the client will only reveal the preimage once it has received the LSP's funds on-chain. Thus, the LSP pays the off-chain invoice. This forces the client to reveal the preimage, and now the LSP can redeem the on-chain funds and the swap is complete. The entire process happens trustlessly thanks to the HTLC mechanics. More technical details can be found in [16].

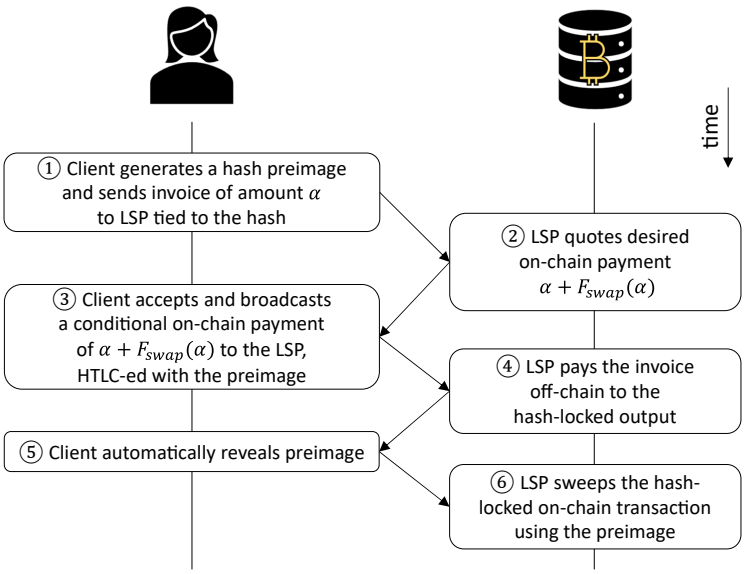


Fig. 8. A swap-in step-by-step

C An equivalent objective

In Sec. 3.2, we formulated the optimal rebalancing problem as a maximization of the total fortune increase. Equivalently, the node can minimize the total fee cost, which comes from two sources: from lost fees because of dropped transactions[13], and from fees paid for rebalancing operations:

$$L(t_i, r(t_i)) = \int_{\tau\in(t_i,t_{i+1})} \big(f(A_{LR}(\tau) - S_{LR}(\tau)) + f(A_{RL}(\tau) - S_{RL}(\tau))\big)d\tau + \sum_{n\in\mathcal{N}} \big(F^{\text{in}}_{\text{swap}}(r^{\text{in}}_n(t_i)) + F^{\text{out}}_{\text{swap}}(r^{\text{out}}_n(t_i))\big)$$

Under this objective, the ultimate goal of node N is to find a rebalancing policy that minimizes the long-term average expected fee cost L over all admissible policies:

$$\underset{\pi\in\Pi}{\text{minimize}} \lim_{H\to\infty} \frac{1}{t_H} \sum_{i=0}^{H} \mathbb{E}\left[L(t_i, r^{\pi}(t_i))\right]$$

subject to the constraints (2)–(6).

The two objectives at each timestep sum to $\int_{\tau\in(t_i,t_{i+1})}(f(A_{LR}(\tau)+f(A_{RL}(\tau))d\tau$ (the fees that would be collected by the node if the total arriving amount had been processed), a quantity independent of the control action, and therefore maximizing the total fortune and minimizing the total fee cost are equivalent.

D Pseudocode of heuristic policies

Algorithm 1: Autoloop rebalancing policy

Input: *state* as in (1)
Parameters: T_{check}, *low, high*

```
every T_check do
    foreach neighbor n ∈ 𝒩 do
        midpoint = C_n · (low + high)/2
        if b_Nn < low · C_n then
            Swap-in amount = midpoint − b_Nn
        else if b_Nn > high · C_n then
            Swap-out amount = b_Nn − midpoint
        else
            Perform no action
```

[13] Note that we assume the node knows not only about the transactions that reach it, but also about the transactions that are supposed to reach it but never do because of insufficient remote balances. This is not strictly true in practice, but the node can approximate it by observing the transactions during an interval in which the remote balances are both big enough so that no incoming transaction would fail, and create an estimate based on this observation.

Note: the original Autoloop algorithm defines the thresholds in terms of the node's inbound liquidity in a channel. We adopt an equivalent balance-centric view instead.

Algorithm 2: Loopmax rebalancing policy

Input: *state* as in (1)
Parameters: T_{check}
every T_{check} **do**
  Update $\{\hat{A}^{\text{net}}_{Nn}\}_{n\in\mathcal{N}}$ according to (9)
  foreach *neighbor* $n \in \mathcal{N}$ **do**
    if $\hat{A}^{\text{net}}_{Nn} < 0$ **then**
      $ETTD = b_{Nn}/|\hat{A}^{\text{net}}_{Nn}|$ /* estimated time to depletion */
      if $ETTD < T_{\text{check}} + T_{\text{conf}}$ **then**
        Swap-in amount $= \max\{\phi^{-1}(B_N), b_{nN}\}$ /* maximum possible swap in */
      else
        Perform no action
    else if $\hat{A}^{\text{net}}_{Nn} > 0$ **then**
      $ETTS = b_{nN}/\hat{A}^{\text{net}}_{Nn}$ /* estimated time to saturation */
      if $ETTS < T_{\text{check}} + T_{\text{conf}}$ **then**
        Swap-out amount $= b_{Nn}$ /* maximum possible swap out */
      else
        Perform no action
    else
      Perform no action

E Deep reinforcement learning algorithm design details

E.1 Decoupling the coupled constraint

All constraints are decoupled per channel, except for (6). However, we observe that given some traffic, mostly in the L-to-R direction or mostly in the R-to-L direction or equal in both directions, the local balances of node N will either deplete in one channel and accumulate in the other, or accumulate in both, but never both deplete. Thus, a swap-in in both channels in general will not be a good action. Therefore, for the RL solution's purposes we can split (6) into two constraints, one for each channel, with the right-hand side of each being the entire amount $B_N(t_i)$. In case the agent does take the not advisable decision of swap-ins in both channels and their sum exceeds the on-chain amount, one of the two will simply fail.

E.2 Helping a swap-in succeed

When a swap-in is about to complete time T_{conf} after it was requested, the remote balance in the respective channel needs to suffice[14] (otherwise the swap-in will fail and a refund will be triggered as in (8)):

$$r_n^{\text{in}}(t_i) \leq b_{nN}(t_i) + d_{nN}^{(t_i, t_i+T_{\text{conf}})} \text{ for all } i \in \mathbb{N}, n \in \mathcal{N} \tag{10}$$

Although (10) are not hard constraints when the decision is being made like the ones of Sec. 3.1, we would like to guide the agent to respect them. An obstacle is that the swap-in decision is made at time t_i, when the node does not yet know the arriving amount $d_{nN}^{(t_i, t_i+T_{\text{conf}})}$. To approximate the right-hand side of (10) in terms of quantities known at time t_i, we can use the difference of the total (and not the successful as in d_{nN}'s definition) amounts that arrived in each direction from (9):

$$\begin{aligned} b_{nN}(t_i) + d_{nN}^{(t_i, t_i+T_{\text{conf}})} &\approx \hat{b}_{nN}(t_i + T_{\text{conf}}) \\ &\triangleq \left(\min\{b_{nN}(t_i) + \hat{A}_{nN}^{\text{net}} \cdot T_{\text{conf}}, C_n\}\right)^+ \end{aligned} \tag{11}$$

A better estimate can be obtained by using the empirical amounts that succeeded in either direction:

$$\hat{S}_{LR}(\tau) \triangleq \frac{1}{\tau} \int_{t \in [0,\tau]} S_{LR}(t) dt \qquad \text{and} \qquad \hat{S}_{RL}(\tau) \triangleq \frac{1}{\tau} \int_{t \in [0,\tau]} S_{RL}(t) dt \tag{12}$$

Then the amount $\hat{S}_{LR}$ (resp. $\hat{S}_{RL}$) will be flowing in the L-to-R (resp. R-to-L) direction for either the entire duration of T_{conf}, or until one of the balances in the respective direction is depleted:

$$\begin{aligned} \hat{b}_{LN}(t_i+T_{\text{conf}}) \triangleq \Bigg(\min \Bigg\{ & b_{LN}(t_i) - \hat{S}_{LR}(t_i) \min\left\{T_{\text{conf}}, \frac{b_{LN}}{\hat{S}_{LR}(t_i)}, \frac{b_{NR}}{\hat{S}_{LR}(t_i)}\right\} \\ & + (1 - f_{\text{prop}}) \hat{S}_{RL}(t_i) \min\left\{T_{\text{conf}}, \frac{b_{RN}}{\hat{S}_{RL}(t_i)}, \frac{b_{NL}}{\hat{S}_{RL}(t_i)}\right\}, C_L \Bigg\} \Bigg)^+ \end{aligned} \tag{13}$$

and symmetrically for the NR channel. Thus, the approximate version of (10) becomes:

$$r_n^{\text{in}}(t_i) \leq \hat{b}_{nN}(t_i + T_{\text{conf}}) \text{ for all } i \in \mathbb{N}, n \in \mathcal{N} \tag{14}$$

[14] Note that we have given the RL agent more flexibility compared to Autoloop and Loopmax: it can perform swap-ins of amount bigger than the current remote balance under the expectation that by the time of completion the balance will be adequate.

E.3 Mapping of sampled to implemented actions

Table 1. Mapping of raw actions sampled from the learned distribution to final swap amounts requested for channel Nn, $n \in \mathcal{N}$

Raw action $r_n \in [-1, 1]$	Corresponding absolute amount $\tilde{r}_n$	Final requested swap amount
$r_n < 0$	$\lvert r_n \rvert b_{Nn}$	swap out $\tilde{r}_n \mathbb{1}\{\tilde{r}_n \geq \rho_0 C_n\}$
$r_n \geq 0$	$r_n \min\{\hat{b}_{nN}(t_i + T_{\text{conf}}), \phi^{-1}(B_N(t_i)), C_n\}$	swap in $\tilde{r}_n \mathbb{1}\{\tilde{r}_n > \rho_0 C_n\}$

E.4 Deep reinforcement learning algorithm for the RebEL policy

Algorithm 3: RL algorithm for RebEL policy

Input: *state* as in (1)
Parameters: T_{check}, various learning parameters, penalty
every T_{check} **do**
  Update estimates $\hat{S}_{LR}$, $\hat{S}_{RL}$ and $\hat{b}_{LN}$, $\hat{b}_{RN}$ according to (12)–(13)
  Perform SAC gradient step to update policy distribution as in [15] based on replay memory
  Fetch $state \in [0, 1]^7$
  Sample $rawAction$ from $[-1, 1]^2$ according to policy distribution
  $processedAction = process(rawAction)$ where $process(\cdot)$ is described in Table 1
  Apply $processedAction$ and wait for its completion
  $reward$ = fortuneAfter − fortuneBefore − lostFees − penalty · numberOfFailedSwaps
  Fetch $nextState \in [0, 1]^7$
  Store transition $(state, rawAction, reward, nextState)$ to replay memory

E.5 Design choices

In our model in Sec. 3.1, we have considered the time for on-chain transaction confirmation and thus also rebalancing completion to be constant. In practice, completion happens when the miners solve the random puzzle and produce the Proof-of-Work for the next block that includes the rebalancing transaction. The time for this to happen fluctuates, though only slightly, so we use a fixed value for the sake of tractability.

Also, in practice, the on-chain funds used in a swap are unlocked after a time T_{exp} to prevent malicious clients from requesting many swaps from an LSP and

then defaulting. However, since we are concerned with online and cooperative clients with on-chain amounts usually quite larger than the amounts in their channels (and thus than their swaps), and also there is currently a community effort to reduce or even eliminate T_{exp}, we ignore it.

The objective in Sec. 3.2 was defined as a long-term expected average in order to match what a relay node would intuitively want to optimize, while the SAC algorithm works for long-term discounted objectives with a discount factor (usually set very close to 1), and including a maximum entropy term to enhance exploration[15]. We expect this difference to not be significant, and indeed the results show that the SAC-based policy performs well in practice.

In Sec. 5, we presented results for specific parameters and rewards for the RL algorithm. Further tuning specific to the demand regime might lead to even higher returns for the RebEL policy. Additionally, improving the estimates of future balances by having the agent perform a "mini-simulation" of the transactions arriving in the following time interval based on past statistics could help the policy produce more informed decisions. Techniques from Model Predictive Control could also be applied.

Theoretically, a class of policies that could result in even higher fortune than the class (3.2) would be one that would allow rebalancing to happen at any point in continuous time instead of periodically. Optimization in such a model however would be extremely difficult, as an action taken now would affect the state both now and in the future (when rebalancing completes). Considering that practical policies like Autoloop applied today only check for rebalancing periodically, we follow the same path for the sake of tractability.

E.6 Practical applicability

An actual PCN node could use our simulator with samples from its past demand, and try to tune the RL parameters and the reward to get better performance than the heuristic policies we defined or the one it is currently using; then, it will apply the policy learned in the simulator environment to the real node. Alternatively, a node may not use a simulator at all and directly learn a pre-parameterized policy on the fly from the empirical transaction data. In either case, the node can do occasional retraining with updated data to account for time-variance in the distribution of the arriving demand.

[15] The exact formula for the SAC objective can be found in Appendix A of [15].

F Hyperparameters and rewards

Table 2. SAC hyperparameters used for the different experiments of Sec. 5

SAC hyperparameter	**Parameter value for skewed demand experiments**	**Parameter value for even demand experiments**
policy	Gaussian	
optimizer	Adam	
learning rate	0.0003	0.006
discount	0.99	
replay buffer size	10^5	
number of hidden layers (all neural networks)	2	
number of hidden units per layer	256	
number of samples per minibatch	10	
temperature	0.05	0.005
nonlinearity	ReLU	
target smoothing coefficient	0.005	
target update interval	1	
gradient steps	1	
automatic entropy tuning	False	True
initial random steps	10	

Table 3. Parameters used in RebEL's representation or processing of the states, actions, and rewards

RebEL parameter	**Parameter value for skewed demand experiments**	**Parameter value for even demand experiments**
on-chain amount normalization constant	60	
minimum swap threshold ρ_0	0.2	
penalty per swap failure	0	10

G Additional experimental results

G.1 Remarks on the varying fees case

In Figs. 5(b) and 5(d), although RebEL performs better for $f_{\text{prop}} = 1\%$, Autoloop and Loopmax sometimes perform better for even higher (and thus even farther from the current) fees, because the RebEL policy used in this experiment is the one we tuned to operate best for the experiments of the next section that use $f_{\text{prop}} = 1\%$. In principle though, with different tuning, RebEL could outperform the other policies for higher values of f_{prop} as well.

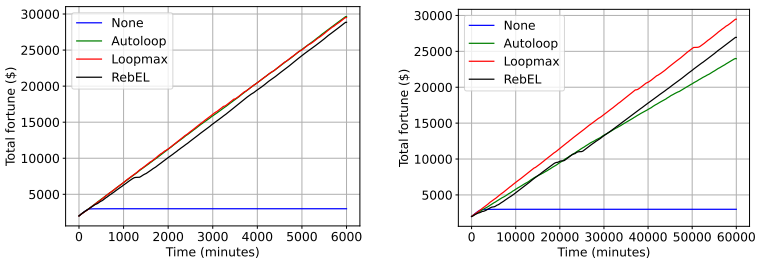
G.2 Remarks on the skewed demand case (Fig. 6)

Under high demand, there is a point in Fig. 6(a) around time 2700 where RebEL stalls for a bit, and the same happens under low demand in Fig. 6(d) between times 14000-22000. Upon more detailed inspection, this happens because all balances temporarily accumulate on the local sides of the channels. RebEL takes some steps to again bring the channels to some balance (either actively by making a swap or passively by letting transactions flow) and subsequently completely recovers.

Note that the figures over time are for a single experiment so that we can assess the actual evolution with time, but conducting multiple experiments with different samples from the same arrival distributions and averaging the results should eliminate these small plateaus.

G.3 The RebEL policy under even demand

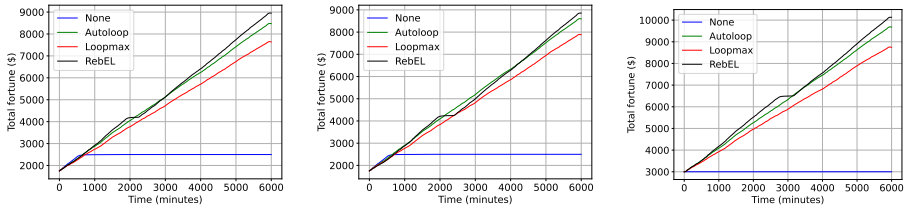
In this section, we explore the special case of equal demands in the two directions, by applying 60000 transactions arriving on each side in high (10 tx/minute) and low (1 tx/minute) intensity. Tuning some hyperparameters and making the penalty for failed swaps non-zero as shown in Appendix F gave better results for even demand specifically, so we use this configuration for the results of Fig. 9.



(a) High demand (b) Low demand

Fig. 9. Total fortune over time under equal demand intensity from both sides

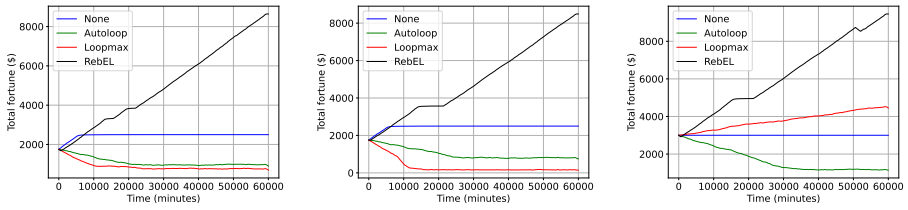
We observe that all policies (except *None*) achieve higher total fortunes than before. This happens because the almost even traffic automatically rebalances the channel to some extent and therefore more fees can be collected in both directions and for larger amounts of time before the channels get stuck. RebEL is not as good for even traffic, because the net demand constantly oscillates around zero and this does not allow the agent to learn a good policy. It still manages though to surpass Autoloop pretty quickly under low demand, while if we run the simulation for longer times (not shown in the figure), we see that after time 78000 RebEL surpasses Loopmax as well. This translates to about 54 days of operation, which is a big time interval in practice, but is justified by the fact that the traffic is low and therefore more time is needed in order for the node to make a profit. However, even demand from both sides is a special case that is not likely to occur in practice, as usually the traffic follows some patterns, e.g. from clients to merchants. So the skewed demand scenario, where RebEL is superior, is also the most natural.



(a) Total fortune over time when $C_L = 1000$, $C_R = 500$ (high demand)

(b) Total fortune over time when $C_L = 500$, $C_R = 1000$ (high demand)

(c) Total fortune over time when initial balances are only local (high demand)



(d) Total fortune over time when $C_L = 1000$, $C_R = 500$ (low demand)

(e) Total fortune over time when $C_L = 500$, $C_R = 1000$ (low demand)

(f) Total fortune over time when initial balances are only local (low demand)

Fig. 10. Total fortune, transaction fee losses and rebalancing fees over time under demand skewed in the L-to-R direction for different initial conditions

G.4 The role of the initial conditions

In this section, we examine how the initial conditions (capacities, initial balances) affect the performance. We evaluate all rebalancing policies for the skewed demand in the L-to-R direction scenario as before, but this time for channels of uneven capacities or initial balances. The results for high and low demand are shown in Figs. 10(a) and 10(d) respectively for $C_L = 1000$, $C_R = 500$ and the initial balances evenly distributed, in Figs. 10(b) and 10(e) respectively for $C_L = 500$, $C_R = 1000$ and the initial balances evenly distributed, and in Figs. 10(c) and 10(f) respectively for $C_L = C_R = 1000$ but $b_{NL} = b_{NR} = 1000$ (and so $b_{LN} = b_{RN} = 0$). We see that RebEL performs well in all these cases as well. Depending on the exact arriving transactions, the little plateaus of RebEL happen at different points in time for the same reason as explained in Appendix G.2, but in the end the learning algorithm recovers.